Deep Into Pharo
Esug 2013 Edition

한국어 번역 초판(2014.05.19)

License

본 저서는 아래 링크에서 다운로드 가능하다:

http://deepintopharo.com

출판: 스위스의 Square Bracket Associations. http://SquareBracketAssociations.org
ISBN 978-3-9523341-6-4
제 1판, 2013년 8월. 표지 제작 Jerome Bergel 담당 (jeromebergel@gmail.com).

차 례

제 1 장

서문

"스몰토크는 유연하고 탐구적인 프로그래밍의 훌륭한 도구로 잘 알려져 있다. 본 저서에서 저자들은 스몰토크의 새로운 dialect 이자 특히 독창적인 개발자들을 위해 고안된 Pharo를 제시한다. 저자들은 Pharo 팀의 핵심 구성원들로서 다재다능한 OO 교육인들, 연구원들, 디자이너들이다. 저자들을 포함해 많은 이들이 만든 다양한 스몰토크 프로젝트를 Pharo로 이식하였다. Deep into Pharo를 즐기길 바란다"

- Dave Thomas[1]

프로그래밍 언어는 인간이 컴퓨터에게 할 일을 알려주는 데에 사용하는 가장 간편한 방법에 해당한다. Pharo는 객체 지향 프로그래밍 언어로, 스몰토크의 영향을 많이 받았다. Pharo는 구문과 여러 의미론적 규칙으로 이루어진 대부분의 프로그래밍 언어와는 달리 그 이상이다. Pharo는 광범위하고 유연한 프로그래밍 환경이 함께 온다. 수많은 객체 지향 라이브러리와 프레임워크 덕분에 Pharo는 데이터의 모델링과 시각화, 스크립팅, 네트워킹, 다른 많은 애플리케이션 범위를 특징으로 한다.

흔히 Pharo는 매우 가벼운 구문과 가변성 있는 객체 모델로 유명하다. 초보 학습자와 숙련된 프로그래머 모두 "모든 것은 객체다"라는 패러다임을 즐긴다. 생활환경을 비롯해 Pharo의 단순성과 표현력은 프로그래머에게 경이롭고 유일한 감각을 즐길 수 있는 권한을 부여한다.

Deep into Pharo는 Pharo by Example[2]로 시작된 시리즈에서 두 번째 책에 해당한다. 당신이 읽고 있는 Deep into Pharo는 독자를 흥미로운 Pharo의 부분으로 데려다주는 멋진 여정을 선사한다. FileSystem과 같은 새 라이브러리, Rossal과 Glamour와 같은 프레임워크, 예외나 블록과 같은 시스템 측면의 복합을 다룬다.

[1] David (http://www.davethomas.net) 현대 소프트웨어 개발 및 객체 기법에서 유명한 인물이다. Thomas는 아마도 현재 IBM OTI Labs가 된 Object Technology International, Inc. 의 창립자이자 전 CEO로 가장 잘 알려져 있을 것이다. OTI는 Eclipse 오픈 소스 IDE의 초기 개발과 Visual Age Java 개발 환경을 책임졌다.

[2] http://pharobyexample.org 무료로 이용 가능하다.

책은 5부, 17장으로 나뉜다. 제 1부는 진정한 객체 지향 라이브러리를 다룬다. 제 2부는 소스 코드 관리, 제 3부는 향상된 프레임워크, 제 4장은 언어의 고급 주제, 특히 예외, 블록, 숫자를 다룬다. 마지막 5부는 프로파일링과 파싱을 포함해 툴링 (tooling) 에 관한 부분이다.

Pharo 는 나날이 성장해가는 강력한 공동체의 지원을 받는다. Pharo의 공동체는 활동적이고, 혁신적이며, 항상 소프트웨어 공학의 한계를 밀어붙인다. Pharo 공동체는 소프트웨어를 생산하는 기업, 일반 프로그래머들로 구성되나 높은 수준의 컨설턴트, 연구원, 교사들이기도 하다. 이 책은 Pharo 공동체 덕분에 존재하므로 우리들을 제2의 가족으로 여기는 집단에게 바치는 바이다.

감사의 말

이 책에 도움을 준 많은 분들에게 감사의 말을 전하고 싶다. 특히 아래 열거된 분들에게 감사함을 전한다:

- Zero Configuration 장에 참여해 주신 Camillo Bruni.
- Socket 장에 참여해 주신 Noury Bouraqadi와 Luc Fabresse.
- Setting Framework 장에 참여하고 그 내용을 Pharo에 통합하기 위해 노력해 주신 Alain Plantec.
- Regex와 Monticello와 같이 몇몇 장의 내용을 작성하고 공동 편집해 주신 Oscar Nierstrasz.
- Metacello 장에 참여해 주신 Dale Henrichs와 Mariano Martinez Peck.
- Glamour 장과 첫 문서에 도움을 주신 Tudor Doru Girba.
- Exception 장에 도움을 주신 Clement Bera.
- Floats 장과 Fun 에 참여해 주신 Nicolas Cellier.
- PetitParser에 도움을 주시고 재팩토링 엔진과 smallLint 규칙에 수고해 주신 Lukas Renggli.
- PetitParser 장에 참여해 주신 Jan Kurs와 Guillaume Larcheveque.
- FileSystem의 초기 버전에 참여해 주신 Colin Putney와 FileSystem을 검토하고 Pharo Core를 재작성 해주신 Camillo Bruni.
- Roassal과 Mondrian 장에 참여해 주신 Vanessa Pena.
- 교정에 도움을 주신 Renato Cerro.
- 질문, 지원, 버그 수정, 기여, 격려를 해주신 독자들.

내용을 검토해 주신 Hernan Wilkinson과 Carlos Ferro, 숫자와 관련된 장에서 피드백을 주신 Nicolas Cellier, Regex 문서를 수정할 수 있는 권한을 제공해 주신 Vassili Bykov에게 감사의 말을 전한다.

그리고 이 오픈 소스 프로젝트를 지원하고 서적의 웹 사이트를 운영하는 Inria Lille Nord Europe에게도 감사하다고 말하고 싶다. 커버를 후원해준 Object Profile 또한 감사하다.

마지막으로 언급하지만 모든 이들만큼 감사의 말을 전하고 싶은 분은 프로젝트를 열정적으로 지원해주시고 제 1판에서 발견된 오류를 알려주신 Pharo 공동체이다.

또한 각 관련 기관과 국가연구기관의 지원과 시설을 제공해주심에 감사의 말씀을 전한다. 특히 칠레 대학교의 프로그램 U-INICIA 11/06 VID 2011, FONDECYT 프로젝트 1120094에도 고마움을 전한다. 끝으로 Plomo Equipe Associee도 감사의 말을 전한다.

제 I 부

Libraries

제 2 장

zero 설정 스크립트와 명령행 핸들러

***Camillo Bruni** 참여 (camilobruni@gmail.com)*

단일 명령행으로부터 Pharo를 설치하거나 명령행으로 인자를 전달할 수 없다는 사실에 신물난 적이 없는가? 훌륭한 디버거와 대화형 환경 개발을 이용한다고 해서 Pharo 개발자들이 자동 스크립트를 중요하게 생각하지 않고 명령행을 사랑한다는 의미는 아니다. 우리도 물론 그것을 중요하게 생각하며, 두 가지 모두에게 최선을 원한다! 우린 임의의 정보를 유지해야 한다는 생각에서 자유롭길 원했다. zero 설정은 당신이 시작하는 데에 필요한 모든 것을 자동으로 다운로드하는 스크립트다. Pharo 2.0 이상 버전부터는 명령행 인자를 정의하고 처리하는 방식 또한 지원하고 있다.

본 장에서는 명령행으로부터 환경으로 인자를 전달하는 방법 뿐만 아니라 Pharo를 위한 zeroconf 스크립트를 얻는 방법도 제시하고자 한다.

VM과 Image 얻기

최신 2.0 Pharo 이미지와 vm을 다운로드하기에 앞서 zero 설정을 다운로드 하는 방법은 다음과 같다.

```
wget get.pharo.org/20+vm
```

wget가 설치되어 있지 않다면 curl -L 을 대신 사용해도 좋다.

방금 다운로드한 스크립트를 실행하기 위해서는 chmod a+x 를 이용해 권한을 변경하거나 아래와 같이 bash를 통해 호출하는 방법을 택해야 한다.

설정하기. 이용 가능한 설정의 종류는 어마어마하다. 각 스크립트에 대한 URL은 다음 표현식에 따라 vm과 이미지 버전으로부터 쉽게 빌드가 가능하다: get.pharo.org/$IMAGE_$VM

> $IMAGE에 가능한 값은 다음과 같다: 12 13 14 20 30 stable alpha
> $VM에 가능한 값은 다음과 같다: vm vmS vmLatest vmSLatest

물론 get.pharo.org/$IMAGE 에서 이미지를 다운로드하거나 get.pharo.org/$VM에서 VM을 다운로드할 수도 있다.

help 살펴보기. 스크립트 도움말(help)을 살펴보자.

```
bash 20+vm --help
```

help를 살펴보면 20+vm 명령이 현재 가상 머신을 다운로드하여 pharo-vm 폴더로 넣는다고 말하고 있다. 또한 몇 가지 스크립트를 생성한다: 시스템을 시작하기 위한 pharo, 이미지를 UI 모드에서 시작하기 위한 스크립트인 pharo-ui. 마지막으로 최신 이미지를 다운로드하고 파일을 변경하기도 한다.

```
This script downloads the latest Pharo 20 Image.
This script downloads the latest Pharo VM.

The following artifacts are created:
 Pharo.changes A changes file for the Pharo Image
 Pharo.image A Pharo image, to be opened with the Pharo VM
 pharo Script to run the downloaded VM in headless mode
 pharo-ui Script to run the downloaded VM in UI mode
 pharo-vm/ Directory containing the VM
```

잡고(grab) 실행하기. 접 스크립트를 실행하길 원한다면 아래를 실행할 수도 있다.

```
wget -O -- get.pharo.org/20+vm | bash
```

옵션 – O – 는 다운로드된 bash 파일을 standard out으로 출력할 것이므로 이를 bash로 pipe할 수 있다. 웹의 로그가 마음에 들지 않을 시 –quiet를 사용하라.

```
wget --quiet -O -- get.pharo.org/20+vm | bash
```

자동화 업무를 신뢰하는 사람들은 참고하기 바란다. 스크립트는 우리 Jenkins 서버 (https://ci.inria.fr/pharo/job/Scripts-download/)와 gitorious 서버 (https://gitorious.org/pharo-build/pharo-build)로부터 자동으로 인출된다. 그렇다. 우리는 에너지를 아껴주는 자동화 업무를 신뢰한다.

VM만 얻기

당신은 다른 스크립트도 사용할 수 있다. 가령 get.pharo.org/vm은 최신 vm만 다운로드한다.

```
wget -O -- get.pharo.org/vm | bash
```

다시 말하지만 어떤 스크립트와 마찬가지로 help 메시지를 언제든 확인할 수 있다.
해당 스크립트는 최신 Pharo VM을 다운로드한다.

```
This script downloads the latest Pharo VM.
The following artifacts are created:
    pharo       Script to run the downloaded VM in headless mode
    pharo-ui    Script to run the downloaded VM in UI mode
    pharo-vm/   Directory containing the VM
```

그림 2.1은 http://get.pharo.org 에서 받을 수 있는 스크립트 목록을 보여준다.

명령행 옵션 처리하기

이제 명령행 인자를 처리하는 새롭고도 훌륭한 방법이 생겼다. 이는 자체 문서화되고 쉽게 확장 가능하다. 명령행이 어떻게 처리되는지 살펴보자. 앞과 마찬가지로 결과에 도달하는 과정을 보여주면서 시작하겠다.

결과에 도달하기

앞에서 언급했다시피 자체 문서화를 좋아하고 중요하게 생각하므로 설명을 얻기 위해 –help 옵션을 사용하라.

```
./pharo Pharo.image --help
```

이는 아래와 같은 출력을 생성할 것이다.

```
Usage: [<subcommand>] [--help] [--copyright] [--version] [--list]
 --help print this help message
 --copyright print the copyrights
 --version print the version for the image and the vm
 --list list a description of all active command line handlers
 <subcommand> a valid subcommand in --list


Documentation:
A DefaultCommandLineHandler handles default command line arguments and options.
The DefaultCommandLineHandler is activated before all other handlers.
It first checks if another handler is available. If so it will activate the found handler.
```

시스템 버전과 핸들러 목록

기본 옵션 중 두 가지가 중요한데, 바로 versions와 list다. 이를 먼저 살펴보도록 하자.

시스템 버전 얻기. 일반적이면서 중요한 명령행 옵션은 –version이다. 버그와 이상(deviant) 행위를 전달할 때 이용하길 바란다.

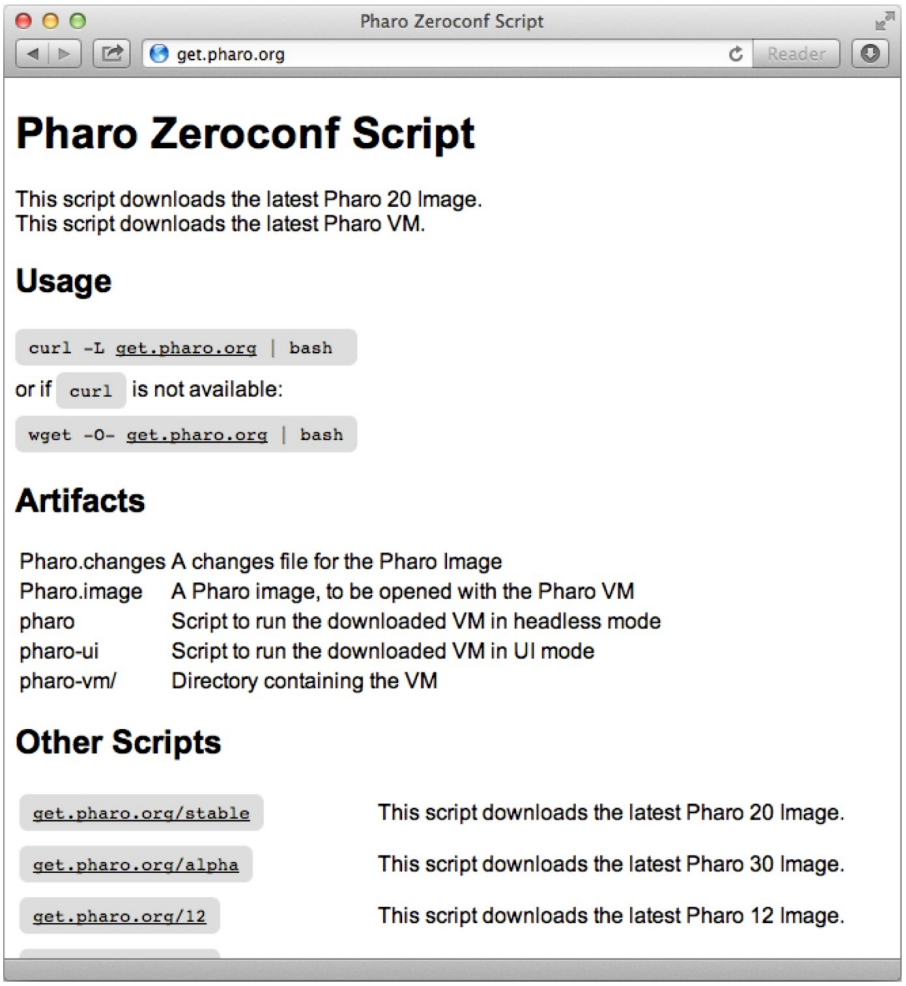


그림 2.1: 모든 스크립트는 http://get.pharo.org 에서 이용 가능하다.

```
./pharo Pharo.image --version
M: NBCoInterpreter NativeBoost-CogPlugin-IgorStasenko.15 uuid: 44b6b681-38f1-4a9e- b6ee-8769b
    499576a Dec 18 2012
NBCogit NativeBoost-CogPlugin-IgorStasenko.15 uuid: 44b6b681-38f1-4a9e-b6ee-8769b499576a
    Dec 18 2012
[git://gitorious.org/cogvm/blessed.git] Commit: 452863bdfba2ba0b188e7b172e9bc597a2caa928 Dat
   e: 2012-12-07 16:49:46 +0100 By:Esteban Lorenzano <estebanlm@gmail.com> Jenkins build \#5922
```

–version 인자는 가상 머신의 버전을 제공한다. 이미지의 버전을 얻고자 한다면 이미지를 열고 World 메뉴를 이용해 About을 선택하면 된다.

이용 가능한 핸들러의 목록. 명령행 옵션 –list 는 현재 옵션 핸들러의 목록을 제공한다는 면에서 흥미롭다. 목록은 현재 시스템에 로딩된 핸들러에 따라 좌우된다. 특히 자신의 상황이나 바람에 맞게 핸들러를 단순히 추가할 수 있음을 의미한다.

아래 목록은 우리가 본 장을 작성할 당시 사용한 시스템에서 이용 가능하다.

```
./pharo Pharo.image --list

Currently installed Command Line Handlers:
 st      Loads and executes .st source files
 Fuel    Loads fuel files
 config  Install and inspect Metacello Configurations from the command line
 save    Rename the image and changes file
 test    A command line test runner
 update  Load updates
 printVersion Print image version
 eval    Directly evaluates passed in one line scripts
```

이제 특정 옵션을 어떻게 사용해야 하는지 궁금할 것이다. 사실 Metacello configurations의 로딩을 처리하는 config 옵션을 이용하는 것이 분명해 보이지 않는다. 하지만 짐작하였듯 핸들러 또한 자체 설명적이다. 한 번 살펴보도록 하자.

Metacello Configuration 로딩하기. Config 옵션의 사용에 관한 설명을 가져오기 위해서는 아래와 같이 연관된 도움말(help)을 요청하면 된다:

```
./pharo Pharo.image config --help
```

이러한 도움말은 명령행 일반적인 시스템에 해당하지 않고 관련 핸들러 중 하나에 해당한다.

```
Usage: config [--help] <repository url> [<configuration>] [--install[=<versi
    on>]] [--group=<group>] [--username=<username>] [--password=<password>]
--help              show this help message
 <repository url>    A Monticello repository name
 <configuration>     A valid Metacello Configuration name
 <version>           A valid version for the given configuration
 <group>             A valid Metacello group name
 <username>          An optional username to access the configuration's repository
 <password>          An optional password to access the configuration's repository


Examples:
 # display this help message
 pharo Pharo.image config


 # list all configurations of a repository
 pharo Pharo.image config $MC_REPOS_URL


 # list all the available versions of a confgurtation
 pharo Pharo.image config $MC_REPOS_URL ConfigurationOfFoo


 # install the stable version
 pharo Pharo.image config $MC_REPOS_URL ConfigurationOfFoo --install


 #install a specific version '1.5'
 pharo Pharo.image config $MC_REPOS_URL ConfigurationOfFoo --install=1.5


 #install a specific version '1.5' and only a specific group 'Tests'
 pharo Pharo.image config $MC_REPOS_URL ConfigurationOfFoo --install=1.5 --group=Tests
```

핸들러의 구조

앞에서 언급하였듯 명령행 메커니즘은 개방형이고 확장이 가능하다. 이제 eval(평가) 옵션에 대한 핸들러가 어떻게 정의되는지 살펴볼 것이다.

Pharo 표현식 평가하기. 아래와 같이 명령행을 이용해 표현식을 평가할 수도 있다:

```
./pharo Pharo.image eval --help

Usage: eval [--help] <smalltalk expression>
  --help list this help message
  <smallltalk expression> a valid Smalltalk expression which is evaluated and
                the result is printed on stdout

Documentation:
A CommandLineHandler that reads a string from the command line, outputs the
        evaluated result and quits the image.

This handler either evaluates the arguments passed to the image:
  $PHARO_VM my.image eval 1 + 2

or it can read directly from stdin:

  echo "1+2" | $PHARO_VM my.image eval
```

이제 다음과 같이 핸들러가 정의된다: 먼저 우리가 CommandLineHandler의 서브클래스를 정의한다. 여기서 BasicCodeLoader는 CommandLineHandler의 서브클래스이고, EvaluateCommandLineHandler 는 BasicCodeLoader 의 서브클래스이다.

```
BasicCodeLoader subclass: \#EvaluateCommandLineHandler
instanceVariableNames: \textit{
classVariableNames: }
poolDictionaries: \textit{
category: 'System-CommandLine'
```

다음으로 클래스 측면에서 commandName을 정의하고 메서드 isResponsibleFor: 도 함께 정의한다:

```
EvaluateCommandLineHandler class>>commandName
    ^ 'eval'

EvaluateCommandLineHandler class>>isResponsibleFor: commandLineArguments
    "directly handle top-level -e and --evaluate options"
    commandLineArguments withFirstArgument: [ :arg|
        (#('-e' '--evaluate') includes: arg)
            ifTrue: [ ^ true ]].

    ^ commandLineArguments includesSubCommand: self commandName

EvaluateCommandLineHandler class>>description
    ^ 'Directly evaluates passed in one line scripts'
```

이제 옵션이 일치하면 실행될 activate 메서드를 정의한다.

```
EvaluateCommandLineHandler>>activate
    self activateHelp.
    self arguments ifEmpty: [ ^ self evaluateStdIn ].
    self evaluateArguments.
    self quit.
```

특히 우리는 클래스 주석을 정의하는데, help를 요청 시 인쇄되는 것이 클래스 주석이기 때문이다.

평가 스크립트의 끝에 이미지가 저장되길 원한다면 –save 옵션을 eval 직후에 전달하라.

Jenkins와 함께 ZeroConf 스크립트 이용하기

우리에겐 그러한 스크립트가 있고 옵션을 명시할 가능성이 있으므로 가능한 한 bash에 적게 의존하는 Jenkins 스크립트를 작성할 수 있겠다.

예를 들어, 프로젝트 XMLWriter(PharoExtras에 host하는)에 대해 jenkins에서 사용하는 명령은 다음과 같다.

```
# jenkins puts all the params after a / in the job name as well :(
export JOB_NAME=‘dirname $JOB_NAME‘

wget --quiet -O - get.pharo.org/$PHARO+$VM | bash

./pharo Pharo.image save $JOB_NAME --delete-old
./pharo $JOB_NAME.image --version > version.txt

REPO=http://smalltalkhub.com/mc/PharoExtras/$JOB_NAME/main
./pharo $JOB_NAME.image config $REPO ConfigurationOf$JOB_NAME --install=$VERSION --group='Tests'
./pharo $JOB_NAME.image test --junit-xml-output "XML-Writer-.*"

zip -r $JOB_NAME.zip $JOB_NAME.image $JOB_NAME.changes
```

요약

이제 쉽게 Pharo의 최신 버전에 접근하고 스크립트를 빌드할 수 있게 되었다. 뿐만 아니라 명령행 핸들러는 셸 스크립트에서 사용될 새로운 지평을 연다.

제 3 장

FileSystem 과 File

***Max Leske** 참여 (maxleske@gmail.com)*

Pharo에서 파일을 처리하기 위한 라이브러리를 FileSystem이라 부른다. 이는 표현적이고 우아한 객체 지향 디자인을 제공한다. 본 장은 사용자의 요구 대부분을 충족시킬 수 있는 API의 주요 측면들을 제시한다.

FileSystem은 많은 사람들이 오랜 시간 열심히 일한 결과물이다. FileSystem은 처음에 Colin Putney에 의해 개발되었으며, 라이브러리는 Pharo의 구성요소(component) 대부분과 관련해 MIT 라이센스에 따라 배포되었다. Camillo Bruni는 원본 디자인 또는 API에 몇 가지 변경을 적용하였다. Camillo Bruni는 이를 Esteban Lorenzano와 Guillermo Polito의 도움을 받아 Pharo로 통합했다. FileSystem에 도움을 주신 분들의 기존 작업이 없었다면 이번 장은 존재하지 않을 것이므로, 도움을 주신 모든 분들께 감사하는 바이다.

시작하기

프레임워크는 서로 대체가 가능하고 투명하게 서로 작업이 가능한 여러 "종류"의 파일시스템(filesystem)을 지원한다. 아마 가장 흔히 사용되는 FileSystem은 자신이 하드 디스크에 보관된 파일과 의 직접 작업과 관련될 것이다. 예로 하나를 살펴보겠다.

FileSystem 클래스는 팩토리 클래스-메서드를 제공하여 여러 filesystem으로 접근성을 제공한다. disk 메시지를 FileSystem으로 전송하면 당신의 물리적 하드 드라이브에 파일 시스템이 리턴된다. 그리고 memory를 전송하면 메모리 이미지에 보관된 새 파일 시스템이 생성된다.

```
| working |
working := FileSystem disk workingDirectory.
   → /Users/ducasse/Workspace/FirstCircle/Pharo/20

working := FileSystem disk workingDirectory class
   → FileReference
```

위의 WorkingDirectory 메시지는 당신의 Pharo 이미지를 포함하는 디렉터리로의 참조를 리턴한다. 참조는 클래스 FileReference의 인스턴스다. 참조는 프레임워크의 중심(central) 객체들로서, 파일 및 디렉터리와 작업을 위한 주요 메커니즘을 제공한다.

FileSystem은 최종 사용자에게 중요한 클래스 네 가지를 정의한다: FileSystem, FileReference, FileLocator, FileSystemDirectoryEntry. FileSystem은 새 파일 시스템을 생성하기 위한 팩토리 메서드를 제공한다. FileReference는 폴더나 파일로의 참조로서, 연산을 검색하고 실행하기 위한 메서드를 제공한다. FileLocator는 늦은 바인딩(late binding) 참조이다. 파일 로케이터(file locator)로 구체적 연산을 실행하도록 요청하면 origin의 현재 위치를 살펴보고 그에 대한 경로를 resolve한다. FileSystemDirectoryEntry는 파일이나 디렉터리에 관한 추가 정보를 받을 수 있도록 해준다. 이러한 클래스들은 'FileSystemCore' 패키지에 속하며, 이번 장의 아래 부분에서 설명하고 있다.

UnixStore 또는 WindowsStore와 같은 플랫폼 특정적 클래스는 내부적 클래스이므로 사용해선 안 된다. 아래 제시한 코드 조각은 모두 FileReference 인스턴스에서 작동한다.

파일 시스템 찾아가기(navigation)

이제 FileSystem을 갖고 놀아보자.

직계 자손(Immediate children). 작업 디렉터리의 직계 자손을 열거하려면 아래 표현식을 실행하라:

```
| working |
working := FileSystem disk workingDirectory.
working children.
   → anArray(File @ /Users/ducasse/Workspace/FirstCircle/Pharo/20/.DS_Store File
       @ /Users/ducasse/Workspace/FirstCircle/Pharo/20/ASAnimation.st ...)
```

자식들은 직접 파일과 폴더를 리턴함을 주목하라. 현재 디렉터리의 모든 자식들을 재귀적으로 접근하기 위해서는 다음과 같이 allChildren 메시지를 사용해야 한다:

```
working allChildren.
```

문자열(string character)을 파일 참조로 변환하는 것은 흔히 사용되는 유용한 연산이다. 단순히 문자열로 asFileReference를 전송하면 그에 상응하는 파일 참조를 얻을 수 있다.

```
'/Users/ducasse/Workspace/FirstCircle/Pharo/20' asFileReference
```

문자열이 기존 파일을 가리키지 않는 경우 오류가 발생됨을 명심한다. 하지만 파일의 존재 여부도 확인이 가능하다:

```
'foobarzork' asFileReference exists
   → false
```

모든 '.st' 파일. 필터링은 파일명에 일치하는 표준 패턴을 이용해 실현된다. 작업 디렉터리 내 모든 st 파일을 찾기 위해서는 아래를 실행하라:

```
working allChildren select: [:each | each basename endsWith: 'st']
```

basename 메시지는 전체명으로부터 파일명을 리턴한다 (예: /foo/gloops.taz 의 파일명은 'gloops.taz'다).

주어진 파일이나 디렉터리로 접근하기. 슬래시 연산자를 이용해 작업 디렉터리 내에서 특정 파일이나 디렉터리로의 참조를 얻는다:

```
| working cache |
working := FileSystem disk workingDirectory.
cache := working / 'package-cache'.
```

부모 폴더로 접근하기. 부모 폴더를 다시 찾아가는 일은 parent 메시지를 이용하면 쉽다:

```
| working cache |
working := FileSystem disk workingDirectory.
cache := working / 'package-cache'.
parent := cache parent.
parent = working
→ true
```

디렉터리 프로퍼티로 접근하기. 당신은 요소의 다양한 프로퍼티를 확인할 수 있다. 다음과 같은 표현식을 실행하여 캐시 디렉터리를 예로 들어보겠다.

```
cache exists. → true
cache isSymLink. "ask if it is a symbolic link" → false
cache isFile. → false
cache isDirectory. → true
cache basename. → 'package-cache'
cache fullName
    → '/Users/ducasse/Workspace/FirstCircle/Pharo/20/package-cache'
cache parent fullName
    → '/Users/ducasse/Workspace/FirstCircle/Pharo/20/'
```

exists, isFile, isDirectory, basename 메서드들은 FileReference 클래스 상에서 정의된다. 파일명(basename) 없이 경로를 얻는 메시지는 존재하지 않으며, 그것을 얻기 위해선 parent fullName을 사용해야 함을 주목하라. path 메시지는 FileSystem이 내부적으로 사용하고 공개적으로 사용되어선 안 되는 Path 객체를 리턴한다.

FileSystem은 파일과 폴더를 사실상 구별하지 않기 때문에 코드는 종종 더 명확해지고 Composite 디자인 패턴의 애플리케이션으로 간주될 수 있다는 사실을 주목하라.

파일 엔트리 상태 질의하기. 파일시스템 엔트리에 관한 추가 정보를 얻기 위해서는 entry 메시지를 이용하는 FileSystemDirectoryEntry를 얻어야 한다. 파일 권한으로 접근할 수도 있음을 명심하라. 몇 가지 예를 소개하겠다:

```
cache entry creation. → 2012-04-25T15:11:36+02:00
cache entry creationTime → 2012-04-25T15:11:36+02:00
cache entry creationSeconds → 3512812296 2012-08-02T14:23:29+02:00
cache entry modificationTime → 2012-08-02T14:23:29+02:00
cache entry size. → 0 (directories have size 0)
cache entry permissions → rwxr-xr-x
cache entry permissions class → FileSystemPermission
cache entry permissions isWritable → true
cache entry isFile → false
cache entry isDirectory → true
```

위치. 프레임워크는 위치, 즉 파일이나 디렉터리를 가리키는 늦게 바인딩된(late-bound) 참조를 지원하기도 한다. 위치(location)에게 구체적 연산을 실행하도록 요청하면 참조와 같은 방식으로 행동한다. 위치를 몇 가지 소개하겠다.

```
FileLocator desktop.
FileLocator home.
FileLocator imageDirectory.
FileLocator vmDirectory.
```

이미지로 위치를 저장하고 이미지를 다른 머신이나 운영체제로 이동할 경우, 위치는 여전히 예상 디렉터리 또는 파일로 resolve할 것이다. 몇몇 파일 위치는 가상 머신에 특정적임을 주목하라.

읽기 및 쓰기 Streams 열기

파일 상에서 스트림을 열기 위해서는 다음과 같이 단순히 writeStream이나 readStream 메시지를 이용해 읽기- 또는 쓰기-스트림에 대한 참조를 요청하기만 하면 된다:

```
| working stream |
working := FileSystem disk workingDirectory.
stream := (working / 'foo.txt') writeStream.
stream nextPutAll: 'Hello World'.
stream close.
stream := (working / 'foo.txt') readStream.
stream contents. → 'Hello World'
stream close.
```

writeStream은 기존의 어떤 파일이든 오버라이드하고, readStream은 파일이 존재하지 않을 경우 예외를 던짐을 기억하길 바란다. 심지어 숙련된 프로그래머들조차 스트림을 닫는 것을 잊어버리는 실수를 흔히 범한다. 스트림을 닫는 행위는 저수준 리소스를 해제(free)시키기 때문에 훌륭한 일이다. readStreamDo: 와 writeStreamDo: 메시지는 close 를 사용하지 않아도 된다. 아래를 살펴보라:

```
| working |
working := FileSystem disk workingDirectory.
working / 'foo.txt' writeStreamDo: [:stream | stream nextPutAll: 'Hello World'].
working / 'foo.txt' readStreamDo: [:stream | stream contents].
```

어떤 경고도 제공하지 않고 파일을 쉽게 오버라이드할 수 있음을 명심하라. 아래와 같은 상황을 고려해보자:

```
| working |
working := FileSystem disk workingDirectory.
working / 'authors.txt' readStreamDo: [:stream | stream contents].
   → 'stephane alexandre damien jannik'
```

파일 authors.txt 는 아래와 같은 내용으로 쉽게 오버라이드가 가능하다:

```
FileSystem disk workingDirectory / 'authors.txt'
   writeStreamDo: [ :stream | stream nextPutAll: 'bob joe'].
```

파일을 다시 읽어오면 희한한 결과를 제공할지 모른다:

```
| working |
working := FileSystem disk workingDirectory.
working / 'authors.txt' readStreamDo: [:stream | stream contents].
   → 'bob joee alexandre damien jannik'
```

openFilestream: aString writable: aBoolean 메시지를 이용해 그에 상응하는 쓰기 상태의 스트림을 얻을 수도 있다.

```
| stream |
stream := FileSystem disk openFileStream: 'authors.txt' writable: true.
stream nextPutAll: 'stephane alexandre damien jannik'.
```

다른 편리한 메서드(convenience methods)를 위한 FileReference의 스트림 프로토콜을 살펴보라.

Files와 Directories 재명명, 복사, 삭제하기

파일은 copyTo: 와 renameTo: 메시지를 이용해 복사 및 재명명이 가능하다. CopyTo: 는 다른 fileReference를 예상하는 반면 renameTo: 는 경로, 경로명, 또는 참조를 예상한다.

```
| working |
working := FileSystem disk workingDirectory.
working / 'foo.txt' writeStreamDo: [:stream | stream nextPutAll: 'Hello World'].
working / 'foo.txt' copyTo: (working / 'bar.txt').

| working |
working := FileSystem disk workingDirectory.
working / 'bar.txt' readStreamDo: [:stream | stream contents].
   → 'Hello World'

| working |
working := FileSystem disk workingDirectory.
working / 'foo.txt' renameTo: 'skweek.txt'.

| working |
working := FileSystem disk workingDirectory.
working / 'skweek.txt' readStreamDo: [:stream | stream contents].
   → 'Hello World'
```

디렉터리 생성. 디렉터리를 생성하기 위해서는 아래와 같이 createDirectory 메시지를 사용하라:

```
| working |
working := FileSystem disk workingDirectory.
backup := working / 'cache-backup'.
backup createDirectory.
backup isDirectory.
   → true

backup children.
   → #()
```

모두 복사하기. copyAllTo: 메시지를 이용하면 디렉터리의 내용을 모두 복사할 수 있다. 아래는 copyAllTo: 를 이용해 완전한 패키지-캐시를 해당 백업 디렉터리로 복사한다:

```
cache copyAllTo: backup.
```

대상 디렉터리가 존재하지 않을 경우 복사 이전에 생성됨을 주목한다.

삭제하기. 하나의 파일을 삭제하려면 delete: 메시지를 이용하라:

```
(working / 'bar.txt') delete.
```

전체적인 디렉터리 트리를 (수신자 포함) 삭제하려면 deleteAll을 이용하되, 이를 사용 시에는 주의를 기울여야 한다.

```
backup deleteAll.
```

주 입구점: FileReference

FileSystem은 다수의 개념과 FileSystem, Path, FileReference 등의 클래스를 기반으로 하는데, 그 중에서 최종 사용자에게 가장 중요한 것은 FileReference이다. FileReference는 파일을 조작하기 위한 연산 집합을 제공한다. 우리는 지금까지 몇 가지 기본 연산만 살펴보았다. 이번에는 좀 더 정교한 연산을 다루도록 하겠다.

디자인 수준에서 파일 참조(FileReference)는 두 가지 저수준 엔티티를 결합한다: 경로(Path)와 파일시스템(FileSystem)을 하나의 객체로 결합하여 파일을 조작하고 처리하기 위한 간단한 프로토콜을 제공한다. FileReference는 FileSystem의 많은 연산을 구현하지만 (둘 다 주로 다형적), 경로와 파일시스템을 따로 추적할 필요는 없다.

Filereference 정보 접근

먼저 basename, base, extensions와 같은 메시지를 이용해 일반 정보로 접근할 수 있는 파일 참조를 예로 들었다.

```
| pf |
pf := (FileSystem disk workingDirectory / 'package-cache' ) children second.
   → /Users/ducasse/Pharo/PharoHarvestingFixes/20/package-cache/AsmJitIgorStasenko.66.mcz

pf fullName
  → '/Users/ducasse/Pharo/PharoHarvestingFixes/20/package-cache/AsmJitIgorStasenko.66.mcz'

pf basename
   → 'AsmJit-IgorStasenko.66.mcz'

pf basenameWithoutExtension
   → 'AsmJit-IgorStasenko.66'

pf base
   → 'AsmJit-IgorStasenko'

pf extension
   → 'mcz'

pf extensions
   → an OrderedCollection('66' 'mcz')
```

Indicator(지시자). FileSystem은 파일 참조 지시자의 개념을 소개한다. 지시자는 참조의 타입을 전달하는 시각적 단서다. 현재로선 세 가지 종류의 지시자가 구현되는데, 존재하지 않는 참조에 대해 '?', 디렉터리에 대해 '/', 파일에 대해서는 빈 문자열이 있다. FileReference는 지시자를 활용하는 basenameWithIndicator 메시지를 정의한다. 아래 표현식을 통해 그 사용을 살펴보자.

```
pf basenameWithIndicator
   → 'AsmJit-IgorStasenko.66.mcz'

pf parent basename
   → 'package-cache'

pf parent basenameWithIndicator
   → 'package-cache/'
```

Path(경로). 경로의 일부로 접근할 필요가 있는 경우, pathSegments 메시지를 이용하면 전체 명을 경로 요소로 줄여 문자열로서 리턴한다. 디자인 측면에서 보면 문자열은 "죽은(dead)" 객체로 간주되므로 가령 path 메시지를 이용하는 실제 객체를 다룰 때에 사용하는 편이 더 낫다.

```
pf pathSegments
  → #('Users' 'ducasse' 'Pharo' 'PharoHarvestingFixes' '20' 'package-cache' 'AsmJit-IgorStasenko.66.mcz')

pf path
  → Path / 'Users' / 'ducasse' / 'Pharo' / 'PharoHarvestingFixes' / '20' / 'packagecache' /'AsmJit-IgorStasenko.66.mcz'
```

크기. FileReference는 파일의 크기로 접근하는 방식 또한 제공한다.

```
pf humanReadableSize
   → '182.78 kB'

pf size
   → 182778
```

파일 정보. creationTime과 permissions를 이용하면 파일 엔트리 자체에 관해 제한된 정보를 얻을 수 있다. 전체 정보를 얻으려면 entry 메시지를 이용해 엔트리 자체로 접근해야 한다.

```
| pf |
pf := (FileSystem disk workingDirectory / 'package-cache' ) children second.
pf creationTime.
   → 2012-06-10T10:43:19+02:00

pf modificationTime.
   → 2012-06-10T10:43:19+02:00

pf permissions
   → rw-r--r--
```

엔트리는 단일 파일의 모든 메타데이터를 표현하는 객체들이다.

```
| pf |
pf := (FileSystem disk workingDirectory / 'package-cache' ) children second.
pf entry

pf parent entries
   "returns all the entries of the children of the receiver"
```

파일 작업하기

파일에 실행하는 연산이 몇 가지 있다.

삭제하기. delete, deleteAll, deleteAllChildren 모두 수신자를 삭제하고, 수신자가 존재하지 않을 시 오류를 발생시킨다. delete는 파일을 삭제하고, deleteAll은 디렉터리와 그 내용을 삭제하며, deleteAllChildren는 디렉터리의 자식만 삭제한다. 게다가 deleteIfAbsent: 는 파일이 존재하지 않을 시 블록을 실행한다.

마지막으로 ensureDelete는 파일을 삭제하지만 파일이 존재하지 않더라도 오류를 발생시키지 않는다. 이와 유사하게 ensureDeleteAllChildren, ensureDeleteAll도 수신자가 존재하지 않더라도 예외를 발생시키지 않는다.

```
(FileSystem disk workingDirectory / 'paf') delete.
   → error

(FileSystem disk workingDirectory / 'fooFolder') deleteAll.
   → error

(FileSystem disk workingDirectory / 'fooFolder') ensureCreateDirectory.
(FileSystem disk workingDirectory / 'fooFolder') deleteAll.

(FileSystem disk workingDirectory / 'paf') deleteIfAbsent: [Warning signal: 'File did not exist'].
(FileSystem disk workingDirectory / 'fooFolder2') deleteAllChildren.
   → error


(FileSystem disk workingDirectory / 'fooFolder2') ensureCreateDirectory.
(FileSystem disk workingDirectory / 'fooFolder2') deleteAllChildren.
```

디렉터리 생성하기. createDirectory는 새 디렉터리를 생성하는데 디렉터리가 이미 존재할 경우 오류를 발생시킨다. ensureCreateDirectory는 디렉터리가 존재하지 않는지를 검사하고, 필요 시에만 생성한다. ensureCreateFile은 필요 시 파일을 생성한다.

```
(FileSystem disk workingDirectory / 'paf' ) createDirectory.
[(FileSystem disk workingDirectory / 'paf' ) createDirectory] on: DirectoryExists do: [:ex|true].
   → true

(FileSystem disk workingDirectory / 'paf' ) delete.
(FileSystem disk workingDirectory / 'paf' ) ensureCreateDirectory.
(FileSystem disk workingDirectory / 'paf' ) ensureCreateDirectory.
(FileSystem disk workingDirectory / 'paf' ) isDirectory.
   → true
```

파일 이동/복사하기. moveTo: 메시지를 이용해 파일의 이동이 가능하며 파일 참조가 예상된다.

```
(FileSystem disk workingDirectory / 'targetFolder') exist
   → false

(FileSystem disk workingDirectory / 'paf') exist
   → false

(FileSystem disk workingDirectory / 'paf' ) moveTo: (FileSystem disk workingDirectory / 'targetFolder')
   → Error

(FileSystem disk workingDirectory / 'paf' ) ensureCreateFile.
(FileSystem disk workingDirectory / 'targetFolder') ensureCreateDirectory.
(FileSystem disk workingDirectory / 'paf' ) moveTo: (FileSystem disk workingDirectory / 'targetFolder' / 'paf').
(FileSystem disk workingDirectory / 'paf' ) exists.
   → false

(FileSystem disk workingDirectory / 'targetFolder' / 'paf') exists.
   → true
```

파일의 이동 외에 복사 또한 가능하다. 파일의 복사에는 copyAllTo: 를 이용할 수 있다. 여기서는 소스 폴더에 포함된 파일을 대상 폴더로 복사하고자 한다.

copyAllTo: 메시지는 수신자의 깊을 복사를 수행하는데, 인자(argument)가 명시한 위치로 복사된다. 수신자가 파일일 경우 파일이 복사된다. 수신자가 디렉터리일 경우 디렉터리와 그 내용이 재귀적으로 복사된다. 인자는 존재하지 않는 참조여야 한다; 인자는 복사로 인해 생성될 것이다.

```
(FileSystem disk workingDirectory / 'sourceFolder') createDirectory.
(FileSystem disk workingDirectory / 'sourceFolder' / 'pif') ensureCreateFile.
(FileSystem disk workingDirectory / 'sourceFolder' / 'paf') ensureCreateFile.
(FileSystem disk workingDirectory / 'targetFolder') createDirectory.
(FileSystem disk workingDirectory / 'sourceFolder') copyAllTo: (FileSystem diskworkingDirectory / 'targetFolder').
(FileSystem disk workingDirectory / 'targetFolder' / 'pif') exists.
   → true

(FileSystem disk workingDirectory / 'targetFolder' / 'paf') exists.
   → true
```

copyAllTo: 메시지는 단일 파일의 복사에도 사용 가능하다:

```
(FileSystem disk workingDirectory / 'sourceFolder') ensureCreateDirectory.
(FileSystem disk workingDirectory / 'sourceFolder' / 'pif') ensureCreateFile.
(FileSystem disk workingDirectory / 'sourceFolder' / 'paf') ensureCreateFile.
(FileSystem disk workingDirectory / 'targetFolder') ensureCreateDirectory.
(FileSystem disk workingDirectory / 'sourceFolder' / 'paf') copyAllTo: (FileSystem diskworkingDirectory / 'targetFolder' / 'paf').
(FileSystem disk workingDirectory / 'targetFolder' / 'paf') exists.
   → true.

(FileSystem disk workingDirectory / 'targetFolder' / 'pif' ) exists.
   → false
```

Locator(로케이터)

로케이터는 늦게 바인딩된 참조이다. 이들은 의도적으로 모호(fuzzy)하게 남겨지는데, 일부 파일 연산이 실행될 때 구체적 참조로만 resolve된다. 로케이터는 파일시스템과 경로 대신 origin과 경로로 이루어진다. origin은 추상적 파일시스템 위치로서, 사용자의 홈 디렉터리, 이미지 파일, 또는 VM 실행파일(executable)이 이에 해당된다. origin이 isFile과 같은 메시지를 수신하면 로케이터는 그 origin을 먼저 resolve한 다음 origin에 대한 경로를 resolve한다.

로케이터는 "이미지 파일과 동일한 디렉터리에 'package-cache'로 명명된 항목"과 같은 사항들을 명시할 수 있도록 만들며, 이미지가 저장된 후 가령 다른 컴퓨터 상의 디렉터리로 이동하더라도 명세를 유효하게 남겨두도록 해준다.

```
locator := FileLocator imageDirectory / 'package-cache'.
locator printString. → ' \{imageDirectory\}/package-cache'
locator resolve. → /Users/ducasse/Pharo/PharoHarvestingFixes/20/package-cache
locator isFile. → false
locator isDirectory. → true
```

현재 지원하는 origin은 다음과 같다:

- imageDirectory – 이미지가 상주하는 디렉터리
- image – 이미지 파일
- changes – changes 파일
- vmBinary – 가상 머신을 실행하기 위한 실행파일
- vmDirectory – VM 애플리케이션을 포함하는 디렉터리 (vmBinary의 부모는 아닐 것이다)
- home – 사용자의 홈 디렉터리
- desktop – 사용자의 데스크톱 내용을 보유하는 디렉터리
- documents – 사용자의 문서가 보관된 디렉터리(예: '/Users/colin/Documents')

애플리케이션은 고유의 origin을 정의하기도 하지만 시스템이 이를 자동으로 resolve할 수는 없을 것이다. 대신 사용자가 디렉터리를 수동으로 선택하도록 요청할 것이다. 사용자의 선택은 캐시 저장되어 향후 해결(resolution)을 요청 시 사용자의 상호작용이 요구되지 않는다.

absolutePath vs. path. absolutePath는 수신자의 절대 경로를 리턴한다. 파일 참조가 가상이 아닌 경우 path와 absolutePath 메시지는 비슷한 결과를 제공한다. 파일이 늦게 바인딩된 참조(FileLocator의 인스턴스)일 경우, absolutePath는 파일을 resolve하고 절대 경로를 리턴하는 반면 path는 아래 보이는 바와 같이 resolve 되지 않은 파일 참조를 리턴한다.

```
(FileLocator image parent / 'package-cache') path
   → {image}/../package-cache

(FileLocator image parent / 'package-cache') absolutePath
   → Path / 'Data' / 'Downloads' / 'Pharo-2.0' / 'package-cache'

(FileLocator image parent / 'package-cache') absolutePath
   → Path / 'Data' / 'Downloads' / 'Pharo-2.0' / 'package-cache'
```

References와 Locators 또한 전체적인 디렉터리 트리를 다루는 간단한 메서드들을 제공한다.

FileSystem 내부 살펴보기

그 단계에서는 파일 조작(file handing)과 관련된 자신의 요구를 충족시키기 위해 FileSystem을 문제 없이 사용할 수 있어야 한다. 이번 절에서는 FileSystem의 내부를 다루고자 한다. 우선 중요한 구현 세부 내용을 살펴볼 것인데, 가령 데이터 베이스나 원격 파일 시스템과 같은 새로운 유형의 파일 시스템을 갖고자 하는 독자들의 관심을 끌 것이다.

FileReference = FileSystem + Path

경로(path)와 파일시스템(filesystems)은 FileSystem API의 최저 수준에 해당한다. FileReference는 경로와 파일시스템을 단일 객체로 결합하는데, 이러한 객체는 앞 절에서 보인 바와 같이 파일로 작업하기 위한 단순한 프로토콜을 제공한다. 참조는 /, parent, resolve: 와 같은 메서드로 경로 프로토콜을 구현한다.

FileSystem(파일시스템)

파일시스템은 디렉터리와 파일의 계층도에 접근하기 위한 인터페이스다. "The filesystem"은 호스트 운영체제에 의해 제공되는데 DiskStore과 그 플랫폼 특정적인 서브클래스에 의해 표현된다. 하지만 사용자는 이로 직접 연결해선 안 되며 앞서 소개한 FileSystem을 이용하여 접근해야 한다. 다른 종류의 파일시스템도 가능하다. 메모리 파일시스템은 모든 파일이 이미지에 ByteArray로서 보관되는 RAM 디스크 파일시스템을 제공한다. zip 파일시스템은 zip 파일의 내용을 나타낸다.

각 파일시스템은 고유의 작업 디렉터리를 갖고 있으며, 그곳으로 전달되는 상대 경로로 resolve하는 데에 사용된다. 예를 몇 가지 들어보겠다:

```
fs := FileSystem memory.
fs workingDirectoryPath: (Path / 'plonk').
griffle := Path / 'plonk' / 'griffle'.
nurp := Path
*
'nurp'.
fs resolve: nurp.
    → Path/plonk/nurp

fs createDirectory: (Path / 'plonk'). → "/plonk created"
(fs writeStreamOn: griffle) close. → "/plonk/griffle created"
fs isFile: griffle. → true
fs isDirectory: griffle. → false
fs copy: griffle to: nurp. → "/plonk/griffle copied to /plonk/nurp"
fs exists: nurp. → true
fs delete: griffle. → "/plonk/griffle" deleted
fs isFile: griffle. → false
fs isDirectory: griffle. → false
```

Path(경로)

경로는 FileSystem API의 가장 기본적인 요소이다. 이들은 매우 추상적인 방식으로 파일시스템 경로를 표현하고, 문자열을 조작할 필요 없이 경로를 작업하기 위해 고수준 프로토콜을 제공한다. 절대 경로(/), 상대 경로(*), 파일 확장자(,), 부모 탐색(parent)을 정의하는 방법을 몇 가지 예로 들겠다. 보통은 Path를 사용하지 않아도 되지만 예를 몇 가지 소개하겠다.

```
| fs griffle nurp |
fs := FileSystem memory.
griffle := fs referenceTo: (Path / 'plonk' / 'griffle').
nurp := fs referenceTo: (Path
*
'nurp').
griffle isFile.
    → false
griffle isDirectory.
    → false
griffle parent ensureCreateDirectory.
griffle ensureCreateFile.
griffle exists & griffle isFile.
    → true
griffle copyTo: nurp.
nurp exists.
    → true
griffle delete
```

```
"absolute path"
Path / 'plonk' / 'feep' → /plonk/feep

"relative path"
Path * 'plonk' / 'feep' → plonk/feep

"relative path with extension"
Path * 'griffle' , 'txt' → griffle.txt

"changing the extension"
Path * 'griffle.txt' , 'jpeg' → griffle.jpeg

"parent directory"
(Path / 'plonk' / 'griffle') parent → /plonk

"resolving a relative path"
(Path / 'plonk' / 'griffle') resolve: (Path * '..' / 'feep')
    → /plonk/feep

"resolving an absolute path"
(Path / 'plonk' / 'griffle') resolve: (Path / 'feep')
    → /feep

"resolving a string"
(Path * 'griffle') resolve: 'plonk' → griffle/plonk

"comparing"
(Path / 'plonk') contains: (Path / 'griffle' / 'nurp')
    →         false
```

경로 프로토콜 중 일부(/, parent, resolve: 와 같은 메시지)는 참조 상에서도 이용 가능함을 명심한다.

Visitors

위의 메서드는 많은 공통 업무에 있어 충분하지만 애플리케이션 개발자는 디렉터리 트리에 좀 더 정교한 연산을 실행해야 하는 경우가 있음을 발견한다.

방문자(Visitor) 프로토콜은 매우 간단하다. 방문자는 visitFile: 와 visitDirectory를 구현할 필요가 있다. 파일시스템의 실제 순회는 가이드(guide)에 의해 처리된다. 가이드는 방문자와 함께 작업하며, 파일시스템을 crawl하고, 방문자에게 그것이 발견하는 파일과 디렉터리를 알려준다. Guide 클래스로는 세 가지가 있는데, PreorderGuide, PostorderGuide, BreathFirstGuide로서, 파일시스템을 각기 다른 순서로 순회한다. 가이드가 특정 방문자로 파일시스템을 순회하도록 만들기란 간단하다. 예를 들어보겠다:

```
BreadthFirstGuide show: aReference to: aVisitor
```

위에 설명된 열거형 메서드들은 방문자들을 이용해 구현된다: 그 예는 CopyVisitor, DeleteVisitor, CollectVisitor를 참고한다.

요약

FileSystem은 파일을 조작하기 위한 강력하고도 명쾌한 라이브러리다. 이는 Pharo에서 기본적인 부분에 해당한다. Pharo 공동체는 계속해서 이를 확장하고 빌드할 것이다. FileReference 클래스는 프레임워크에 가장 중요한 진입점이다.

- FileSystem 은 하드 디스크 상에, 그리고 메모리 내에 파일 시스템을 빌드하기 위한 팩토리 클래스 메서드를 제공한다.
- FileReference 는 프레임워크에서 중심 클래스로서, 파일이나 폴더를 나타낸다. 파일 참조는 파일을 작업하고 파일 시스템 내부를 탐색하기 위한 메서드를 제공한다.
- asFileReference 메시지를 문자열(string character)로 전송하면 그에 상응하는 파일 참조를 리턴한다 (예: '/tmp' asFileReference)
- 새 파일을 생성하여 그 내부에 작성하는 작업은 다음과 같이 간단하다: (FileSystem disk workingDirectory / 'foo.txt') writeStreamDo: [:stream | stream nextPutAll: 'Hello World'].
- FileLocator 는 늦은 바인딩 참조로서, 실행 중인 컨텍스트에 따라 하드 디스크 내의 파일 위치가 좌우될 때 유용하다.
- FileSystemDirectoryEntry 는 주어진 파일에 대한 저수준 세부 내용의 큰 집합을 제공한다.

제 4 장

Socket(소켓)

작성:

Noury Bouraqadi *(Noury.Bouraqadi@mines-douai.fr)*
Luc Fabresse *(Luc.Fabresse@mines-douai.fr)*

현대 과학은 네트워크를 통해 협력하는 여러 장치를 수반하곤 한다. 그러한 협력을 마련하는 기본 접근법은 소켓을 이용하는 방법이다. 일반적인 사용의 예로 World Wide Web을 들 수 있다. 브라우저와 서버는 HTTP 요청과 응답을 전달하는 소켓을 통해 상호작용한다.

소켓의 개념은 1960년대에 버클리 대학교 연구원들에 의해 처음으로 소개되었다. 그들은 Unix 운영체제를 배경으로 C 프로그래밍 언어를 대상으로 첫 번째 소켓 API를 정의하였다. 이후 소켓의 개념은 다른 운영체제로 확산되었다. 그리고 소켓의 API는 거의 모든 프로그래밍 언어로 이식되었다.

이번 장에서는 Pharo를 배경으로 소켓의 API를 제시하고자 한다. 가장 먼저 클라이언트와 서버를 빌드하는 데 소켓을 사용하는 방법에 대한 예제를 제시하겠다. 클라이언트와 소켓은 소켓에 고유한 개념이다: 서버는 클라이언트가 발생한 요청을 기다린다. 그 다음으로 SocketStream와 그 사용 방법을 소개하겠다. 실제로 보통 소켓보다는 SocketStream을 사용하기가 쉽다. 실험에 유용한 Unix 네트워킹 유틸리티를 몇 가지 설명하면서 마무리 짓고자 한다.

기본 개념

Socket(소켓)

원격 통신에는 네트워크를 통해 어느 정도의 데이터 바이트를 교환하는 시스템 프로세스를 최소한 두 개 수반한다. 각 프로세스는 최소 하나의 소켓을 통해 네트워크로 접근한다 (그림 4.1 참조). 소켓은 통신 네트워크 상의 플러그로 정의할 수 있다:

소켓은 양방향 통신을 구축하는 데 사용된다: 소켓은 데이터의 전송과 수신을 모두 허용한다. 그러한 상호작용은 소켓에 의해 캡슐화되는 통신 프로토콜에 따라 이루어질 수 있다. 인터넷

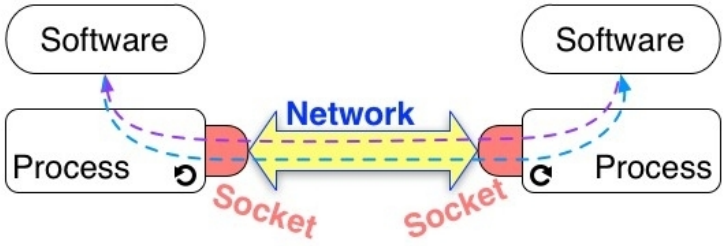


그림 4.1: 소켓을 통한 프로세스간(Inter-Process) 원격 통신.

을 비롯해 이더넷 LAN[1]과 같은 다른 네트워크 상에서 널리 사용되는 두 가지 기본 프로토콜로 TCP/IP와 UDP/IP가 있다.

TCP/IP vs. UDP/IP

TCP/IP는 전송 제어 프로토콜/인터넷 프로토콜을 나타낸다 (줄여서 TCP). TCP는 (데이터 손실 없이) 신뢰성 있는 통신을 보장한다. 이는 통신에 관여한 애플리케이션이 실제로 대화 전에 연결되어 있을 것을 요구한다. 연결이 구축되고 나면 상호작용하는 당사자들은 임의의 바이트 양을 전송하고 수신할 수 있다. 이를 보통 스트림 통신이라고 부른다. 데이터는 전송 시와 같은 순서로 목적지에 도달한다.

UDP/IP는 사용자 데이터그램 프로토콜/인터넷 프로토콜(줄여서 UDP)을 의미한다. 데이터그램은 64KB를 초과할 수 없는 데이터 덩어리를 나타낸다. UDP는 두 가지 이유에서 신뢰성이 없는 프로토콜이다. 첫째, UDP는 데이터그램이 그 목적지에 실제로 도달할 것인지 보장하지 못한다. 두 번째 이유는, 수신자가 하나의 전송자로부터 다수의 데이터그램을 임의의 순서로 수신할 수도 있기 때문이다. 그럼에도 불구하고 UDP는 TCP보다 속도가 빠른데, 데이터를 전송하기 전에 연결을 요구하지 때문이다. 일반적인 UDP의 사용으로는, 클라이언트가 서버에게 그들의 상태를 알려주어야 하는 서버 기반의 소셜 애플리케이션에서 사용되는 "heart-beating"을 들 수 있겠다 (예: Requesting interactions, 또는 Invisible).

본 장의 나머지 부분에서는 TCP 소켓에 전적으로 집중하여 살펴볼 것이다. 가장 먼저 클라이언트 소켓의 생성, 서버로 연결, 데이터 교환, 연결 중단에 관한 방법을 보여줄 것이다. 이러한 라이프사이클은 웹 서버와 상호작용하기 위해 클라이언트 소켓을 이용하는 예제를 통해 설명할 것이다. 다음으로 4.3절은 서버 소켓을 소개한다. 서버 소켓의 라이프사이클, 그리고 이를 이용해 동시연결을 처리 가능한 서버를 구현하는 방법을 설명하겠다. 마지막으로 4.4절에서는 소켓 스트림을 소개하겠다. 그리고 클라이언트와 서버 측에서 소켓 스트림의 사용을 설명함으로써 그들의 이점을 개략적으로 설명하겠다.

[1] 근거리 통신망

TCP 클라이언트

우리는 TCP 클라이언트를 다른 애플리케이션, 즉 서버와 데이터를 교환하기 위해 TCP 연결을 시작하는 애플리케이션이라고 부른다. 클라이언트와 서버는 다른 언어로 개발될 수 있음을 언급하는 것이 중요하겠다. Pharo에서 그러한 클라이언트의 라이프사이클은 4가지 단계로 이루어진다:

1. TCP 소켓을 생성한다.
2. 소켓을 서버로 연결한다.
3. 소켓을 통해 서버와 데이터를 교환한다.
4. 소켓을 닫는다.

TCP 소켓 생성하기

Pharo는 단일 소켓 클래스를 제공한다. 여기에는 소켓 타입(TCP 또는 UDP) 별로 하나의 생성 메서드가 있다. TCP 소켓을 생성하기 위해서는 아래와 같은 표현식을 평가할 필요가 있겠다:

```
Socket newTCP
```

TCP 소켓을 어떤 서버로 연결하기

TCP 소켓을 서버로 연결하기 위해서는 해당 서버의 IP 주소를 나타내는 객체를 가질 필요가 있다. 해당 주소는 SocketAddress의 인스턴스이다. 이를 손쉽게 생성하는 방법은 IP 스타일 네트워크명 검색 및 해석 기능을 제공하는 NetNameResolver를 사용하는 것이다.

스크립트 4.1는 소켓 주소 생성의 두 가지 예제를 제공한다. 첫 번째는 서버명을 설명하는 문자열로부터('www.esug.org') 주소를 생성하는 반면, 나머지 하나는 서버의 IP 주소를 나타내는 문자열로부터 ('127.0.0.1') 생성한다. NetNameResolver를 사용하기 위해서는 DNS[2]를 이용해 네트워크로 연결된 머신이 존재해야 함을 명심한다. 유일한 예외는 로컬 호스트 주소를 검색하는 경우인데, 자신의 소프트웨어(본문에서는 Pharo)를 실행시키는 머신을 참조하는 일반 주소인 127.0.0.1를 예로 들 수 있겠다.

스크립트 4.1: 소켓 주소 생성하기

```
| esugAddress localAddress |
esugAddress := NetNameResolver addressForName: 'www.esug.org'.
localAddress := NetNameResolver addressForName: '127.0.0.1'.
```

이제 스크립트 4.2와 같이 우리 TCP 소켓을 서버로 연결할 수 있다. connectTo:port: 메시지는 매개변수로서 제공된 포트와 서버 주소를 이용해 서버로의 소켓 연결을 시도한다. 서버 주소는 서버가 사용하는 네트워크 인터페이스(예: 이더넷, wifi)의 주소를 나타낸다. 포트는

[2]도메인 네임 시스템: 기본적으로 장치명을 그 IP 주소로 매핑하는 디렉터리

네트워크 인터페이스 상의 통신 끝점을 의미한다. 각 네트워크 인터페이스는 각 IP 전송 프로토콜마다 (예 : TCP, UDP) 0부터 65535까지 숫자로 매겨진 포트의 집합체를 갖고 있다. 주어진 프로토콜에서 인터페이스 상의 포트 번호는 단일 프로세스만이 사용할 수 있다.

스크립트 4.2: TCP 소켓을 ESUG 서버로 연결하기

```
| clientSocket serverAddress |
clientSocket := Socket newTCP.
serverAddress := NetNameResolver addressForName: 'www.esug.org'.
clientSocket
   connectTo: serverAddress port: 80;
   waitForConnectionFor: 10.

clientSocket isConnected
   → true
```

connectTo:port: 메시지는 소켓을 연결하기 위한 요청을 (프리미티브 콜을 통해) 시스템으로 발행한 직후 리턴한다. waitForConnectionFor: 10 메시지는 소켓이 서버에 연결될 때까지 현재 프로세스를 연기한다. 이는 매개변수가 요청한 바와 같이 거의 10 초를 기다린다. 소켓이 10초 이후에도 연결되지 않은 경우 ConnectionTimedOut 예외가 시그널링된다. 그 외의 경우는 true를 응답하는 표현식 clientSociet isConnected를 평가함으로써 실행을 진행할 수 있다.

서버와 데이터 교환하기

연결이 구축되고 나면 클라이언트는 서버와 ByteString의 인스턴스를 교환(전송/수신)할 수 있다. 보통 클라이언트는 서버로 일부 요청을 전송한 후에 응답을 예상한다. 웹 브라우저가 이러한 스키마에 따라 행동하는데, 웹 브라우저는 URL에 의해 식별된 어떤 웹 서버로 요청을 보내는 클라이언트다. 그러한 요청은 서버 상에서 보통 html 파일이나 사진(picture)과 같은 리소스로의 경로에 해당한다. 이후 브라우저는 서버 응답을 기다린다 (예 : html 코드, 사진 바이트 수).

스크립트 4.3: TCP 소켓을 통해 서버와 데이터 교환하기

```
| clientSocket data |
{\dots} "create and connect the TCP clientSocket"
clientSocket sendData: 'Hello server'.
data := clientSocket receiveData.
{\dots} "Process data"
```

스크립트 4.3은 클라이언트 소켓을 통해 데이터를 전송하고 수신하는 프로토콜을 보여준다. 여기서는 sendData: 메시지를 이용해 서버로 'Hello server!' 문자열을 전송한다. 다음으로는, receiveData 메시지를 클라이언트 소켓으로 전송하여 답을 읽는다. 응답을 읽는 것을 블로킹(blocking)이라 부르는데, 응답이 읽힐 때 receiveData가 리턴한다는 의미다. 다음으로 data 변수의 내용이 처리된다.

스크립트 4.4: 최대 데이터 수신 시간을 바운딩하기

```
|clientSocket data|
... "create and connect the TCP clientSocket"
[data := clientSocket receiveDataTimeout: 5.
... "Process data"
    ] on: ConnectionTimedOut
    do: [ :timeOutException |
        self
            crLog: 'No data received!';
            crLog: 'Network connection is too slow or server is down.']
```

클라이언트는 receiveData를 이용함으로써 서버가 더 이상 어떤 데이터도 전송하지 않거나 연결을 종료할 때까지 기다린다는 사실을 명심하라. 다시 말해 클라이언트는 무기한으로 기다릴지도 모른다는 의미다. 이 방법이 아니라면, 스크립트 4.4에서 보이는 바와 같이 클라이언트가 너무 오랜 시간을 기다린 경우 ConnectionTimedOut 예외를 시그널링하는 대안책이 있다. 우리는 클라이언트 소켓에게 5초 간만 기다리도록 요청하기 위해 receiveDataTimeout: 메시지를 이용했다. 데이터가 그 기간 동안 수신되면 조용히 처리된다. 하지만 5초 이내에 데이터가 수신되지 않는 경우, ConnectionTimedOut이 시그널링된다. 예제에서는 어떤 일이 일어났는지 설명을 기록했다.

소켓 닫기

TCP 소켓은 양 끝의 장치들이 연결되어 있는 동안 생존한 채로 남는다. 소켓은 그것으로 close 메시지를 전송하여 닫을 수 있다. 그리고 다른 측에서 그것을 닫을 때까지는 연결된 채로 남는다. 이는 네트워크 실패 또는 다른 측이 다운(down)된 경우 무기한으로 지속될 수도 있다. 이것이 바로 소켓이 destroy 메시지를 수락하는 이유인데, 해당 메시지는 소켓이 요구하는 시스템 리소스를 해제시킨다.

실제로는 closeAndDestroy를 사용한다. 먼저 이는 close 메시지를 전송하여 소켓 끄기를 시도한다. 20초 가 지나서도 소켓이 연결되어 있으면 소켓은 파괴(destroy)된다. closeAndDestroy: seconds 라는 변형체(variant)도 있는데 이는 소켓이 파괴되기 전까지 기간을 매개변수로서 취함을 기억하라.

스크립트 4.5: Web Site 및 Cleanup과 상호작용

```
| clientSocket serverAddress httpQuery htmlText |
httpQuery := 'GET / HTTP/1.1', String crlf,
    'Host: www.esug.org:80', String crlf,
    'Accept: text/html', String crlfcrlf.
serverAddress := NetNameResolver addressForName: 'www.esug.org'.
clientSocket := Socket newTCP.
[ clientSocket
    connectTo: serverAddress port: 80;
    waitForConnectionFor: 10.
clientSocket sendData: httpQuery.
htmlText := clientSocket receiveDataTimeout: 5.
htmlText crLog ] ensure: [clientSocket closeAndDestroy].
```

위에서 설명한 단계를 요약하기 위해 스크립트 4.5의 서버로부터 웹 페이지를 가져오는 예제를 이용했다. 첫째, HTTP[3]쿼리를 가상으로 만들었다. 쿼리에 해당하는 문자열은 GET 키워드로 시작해 서버의 루트 파일을 요청하고 있음을 알리는 슬래시가 추가되었다. 프로토콜 버전 HTTP/1.1을 따른다. 두 번째 행은 웹 서버와 그 포트의 이름을 포함한다. HTTP 쿼리의 3, 4 행은 우리 클라이언트가 수락한 포맷을 나타낸다. 우리는 쿼리의 결과를 Transcript 상에 표시할 것이기 때문에 HTTP 쿼리에 (Accept: 로 시작되는 행 참조) 클라이언트가 html 포맷으로 된 텍스트를 수락함을 명시하였다.

다음으로, www.esug.org 서버의 IP 주소를 검색하였다. 이후 TCP 소켓을 생성하여 서버로 연결하였다. 이전 단계에서 얻은 IP 주소와 웹 브라우저에 대한 기본 포트인 80을 이용했다. 연결은 10초 이내에 구축될 것이며 (waitForConnectionFor: 10), 그렇지 않은 경우 ConnectionTimedOut 예외를 받을 것이다.

http 쿼리를 전송하고 나면 (clientSocket sendData: httpQuery) 우리가 표시하는 수신된 html 텍스트를 소켓으로부터 읽는다. 소켓으로 하여금 서버의 응답을 5초 간 기다리도록 요청함을 명심하라 (clientSocketreceiveDataTimeout: 5). 시간이 만료되면 소켓은 빈 소켓을 응답한다.

마지막으로, 소켓을 닫고 연관된 리소스를 해제시킨다 (clientSocketcloseAndDestroy). 소켓의 연결과 웹 서버와의 데이터 교환을 실행하는 블록으로 ensure: 메시지를 전송시켜 clean up을 확보한다.

TCP 서버

이제 간단한 TCP 서버를 구축해보자. TCP 서버는 TCP 클라이언트들로부터 TCP 연결을 대기하는 애플리케이션이다. 연결이 구축되고 나면 서버와 클라이언트 모두 어떤 순서로든 수신 데이터를 전송할 수 있다. 서버와 클라이언트에 큰 차이점이 있다면 서버는 최소 두 개의 소켓을 사용한다는 점이다. 클라이언트 연결을 처리하는 데에 하나의 소켓이 사용되고, 나머지 하나는 특정 클라이언트와 데이터를 교환하는 데에 사용된다.

TCP 소켓 서버 라이프사이클

TCP 서버의 라이프사이클은 다섯 단계로 나뉜다:

1. connectionSocket 으로 표시된 첫 번째 TCP 소켓을 생성한다.
2. connectionSocket 으로 하여금 포트를 듣도록 만들어 연결을 기다린다.
3. 연결을 위한 클라이언트 요청을 수락한다. 그 결과 connectionSocket 은 interactionSocket 이라는 두 번째 소켓을 빌드할 것이다.
4. interactionSocket를 통해 클라이언트와 데이터를 교환한다. 그 동안 connectionSocket 은 새 연결을 계속해서 기다리는 것이 가능하며, 다른 클라이언트들과 데이터를 교환하기 위해 새 소켓을 생성할지도 모른다.

[3]웹 통신에 사용되는 하이퍼텍스트 전송 규약.

5. interactionSocket을 닫는다.

6. 서버를 끝내고(kill) 클라이언트 연결의 수락을 중단하기로 결정했다면 connectionSocket을 닫는다.

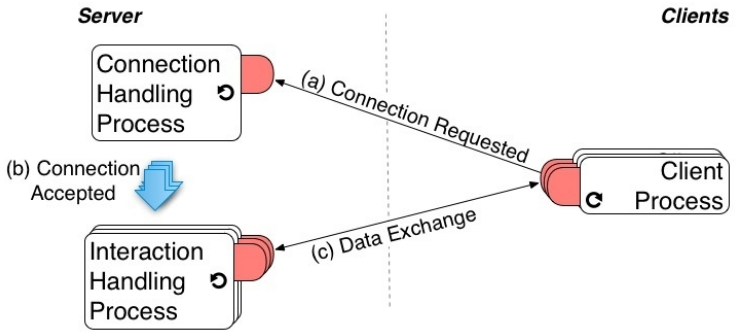


그림 4.2: 다중 클라이언트를 동시에 지원(server)하는 소켓 서버.

이러한 라이프사이클의 동시 실행은 그림 4.2에 잘 표시되어 있다. 서버는 connectionSocket를 통해 들어오는 클라이언트 연결 요청을 듣는데, 어쩌면 다수의 interactionSockets (클라이언트별로 하나)를 통해 다중 클라이언트와 데이터를 교환할지도 모른다. 아래에서는 소켓 서빙 기계(socket serving machinery)를 설명하겠다. 그 다음으로 완전한 서버 클래스를 묘사하고, 서버 라이프사이클 및 관련된 동시성 문제를 설명하고자 한다.

Serving Basic 예제

우리는 단일 클라이언트 요청을 수락하는 에코(echo) TCP 서버의 단순한 예제를 통해 serving basics를 설명하고자 한다. 이는 그것이 수신한 데이터가 무엇이든 클라이언트에게 다시 전송하고 종료한다. 스크립트 4.6에서 코드를 제공한다.

스크립트 4.6: 기본 에코(Echo) 서버.

```
| connectionSocket interactionSocket receivedData |
"Prepare socket for handling client connection requests"
connectionSocket := Socket newTCP.
connectionSocket listenOn: 9999 backlogSize: 10.

"Build a new socket for interaction with a client which connection request is accepted"
interactionSocket := connectionSocket waitForAcceptFor: 60.

"Get rid of the connection socket since it is useless for the rest of this example"
connectionSocket closeAndDestroy.

"Get and display data from the client"
receivedData := interactionSocket receiveData.
receivedData crLog.

"Send echo back to client and finish interaction"
interactionSocket sendData: 'ECHO: ', receivedData.
interactionSocket closeAndDestroy.
```

첫째, 들어오는 연결을 처리하는 데에 사용할 소켓을 생성한다. 소켓이 9999 포트를 듣도록 구성한다. backlogSize는 10으로 설정되는데, 운영체제에게 버퍼를 10회의 연결 요청만큼 할당하도록 요청함을 의미한다. 이러한 백로그는 사실상 우리 예제에선 사용되지 않을 것이다. 하지만 좀 더 현실적인 서버가 다중 연결을 처리한 후 대기 중인 연결 요청은 백로그로 보관해야 할 것이다.

연결 소켓(connectionSocket 변수에 의해 참조되는)이 준비되면 클라이언트 연결을 듣기 시작한다. WaitForAcceptFor: 60 메시지는 소켓이 60초간 연결을 기다리도록 만든다. 60초 동안 어떤 클라이언트도 연결을 시도하지 않을 경우, 메시지는 nil을 응답한다. 그 외의 경우 클라이언트의 소켓으로 연결된 새 소켓 interactionSocket을 얻는다. 이 시점에서 우리는 더 이상 연결 소켓이 필요하지 않으므로 닫을 수 있다 (connectionSocket closeAndDestroy 메시지).

상호작용 소켓은 이미 클라이언트로 연결되어 있으므로 이를 이용해 데이터를 교환할 수 있다. 위에 소개된 receiveData와 sendData: 메시지를 (4.2절 참고) 이용하면 되겠다. 우리 예제에서는 클라이언트로부터 데이터를 기다린 다음 Transcript 에 표시하고, 마지막으로 다시 클라이언트에 보내는데 'ECHO' 문자열을 앞에 붙인다. 끝으로 상호작용 소켓을 닫음으로써 클라이언트와 상호작용을 끝낸다.

스크립트 4.6의 서버를 테스트하는 옵션으로 여러 가지가 있다. 첫 번째 간단한 옵션은 4.5절에 논한 nc (netcat) 유틸리티를 사용하는 것이다. 먼저 워크스페이스에서 서버 스크립트를 실행한다. 다음으로 단말기에서 아래 명령행을 평가한다:

```
echo "Hello Pharo" | nc localhost 9999
```

그 결과 Pharo 이미지의 Transcript 상에 아래와 같은 행이 표시될 것이다:

```
Hello Pharo
```

클라이언트 측, 다시 말해 단말기에는 다음이 표시될 것이다:

```
ECHO: Hello Pharo
```

순수한 Pharo 대안책은 두 가지 다른 이미지의 사용에 의존한다: 하나는 서버 코드를 실행하는 이미지, 나머지 하나는 클라이언트 코드를 실행하는 이미지다. 사실 우리 예제는 사용자 상호작용 프로세스 내에서 실행되기 때문에 Pharo UI는 waitForAcceptFor: 도중이라든가 어떤 특정 시점에 freeze될 것이다. 스크립트 4.7은 클라이언트 이미지에서 실행될 코드를 제공한다. 서버 코드를 먼저 실행해야 함을 주목하라. 이를 어길 경우 클라이언트는 실패할 것이다. 상호작용 이후에는 클라이언트와 서버 모두 종료된다는 사실도 명심하라. 따라서 예제를 두 번째로 실행하길 원한다면 클라이언트와 서버 측 모두에서 실행해야 할 것이다.

스크립트 4.7: 에코 클라이언트.

```
| clientSocket serverAddress echoString |
serverAddress := NetNameResolver addressForName:'127.0.0.1'.
clientSocket := Socket newTCP.
[ clientSocket
        connectTo: serverAddress port: 9999;
        waitForConnectionFor: 10.
    clientSocket sendData: 'Hello Pharo!'.
    echoString := clientSocket receiveDataTimeout: 5.
    echoString crLog.
] ensure: [ clientSocket closeAndDestroy ].
```

에코 서버 클래스

여기서는 동시성 문제를 다루는 EchoServer 클래스를 정의한다. 이 클래스는 동시적 클라이언트 쿼리를 처리하고, UI를 freeze시키지 않는다. 그림 4.3은 EchoServer가 두 개의 클라이언트를 처리하는 방법의 예를 보여준다.

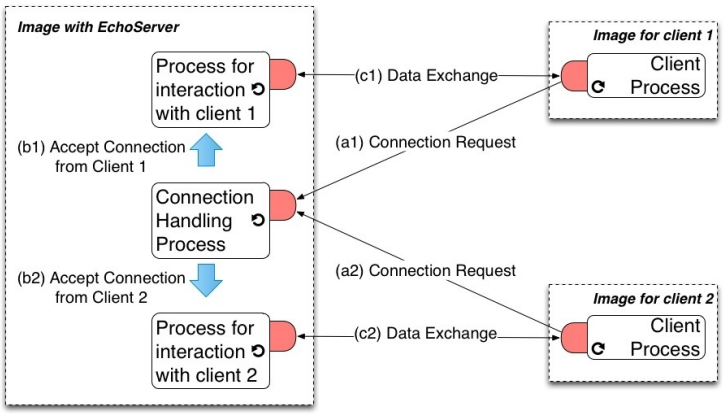


그림 4.3: 두 개의 클라이언트를 동시에 서빙(serving)하는 에코 서버.

클래스 4.8에 라벨로 표시된 정의에서 볼 수 있듯이 EchoServer는 세 개의 인스턴스 변수를 선언한다. 첫 번째(connectionSocket)는 클라이언트 연결을 듣는 데에 사용되는 소켓을 참조

하고, 나머지 두 개의 인스턴스 변수(isRunning은 부울을, isRunningLock은 Mutex를 보유) 동기화 문제를 처리하는 동시 서버 프로세스 라이프사이클을 관리하는 데에 사용된다.

클래스 4.8: EchoServer 클래스 정의

```
Object subclass: #EchoServer
    instanceVariableNames: 'connectionSocket isRunning isRunningLock'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'SimpleSocketServer'
```

isRunning 인스턴스 변수는 서빙이 실행되는 동안에 true로 설정되는 플래그다. 아래에서 알 수 있듯이 여러 프로세스에 의해 접근이 가능하다. 따라서 동시적 쓰기 접근이 존재하더라도 플래그에 대한 읽기 접근이 일관된 값을 얻도록 보장할 필요가 있겠다. 이는 isRunning이 한 번에 하나의 프로세스에 의해서만 접근되도록 보장하는 lock(isRunningLock 인스턴스 변수)을 이용하면 되겠다.

메서드 4.9: EchoServer≫is Running 읽기 접근자

```
EchoServer>>isRunning
    ^ isRunningLock critical: [ isRunning ]
```

메서드 4.10: EchoServer≫isRunning: 쓰기 접근자

```
EchoServer>>isRunning: aBoolean
    isRunningLock critical: [ isRunning := aBoolean ]
```

플래그로의 접근은 접근자 메서드를 통해서만 가능하다 (메서드 4.9와 메서드 4.10). isRunning은 isRunningLock 으로 전송된 critical: 메시지의 인자인 블록 내부에서 읽히고 쓰인다. 이러한 lock은 Mutex의 인스턴스에 해당한다 (메서드 4.11 참조). critical: 메시지를 수신하면 mutex는 인자(블록)을 평가한다. 이러한 평가 도중에 critical: 메시지를 동일한 mutex로 전송하는 다른 프로세스들은 보류된다. 그리고 보류 메시지가 더 이상 없을 때까지 이 과정이 반복된다. 그러므로 mutex는 isRunning 플래그가 순차적으로 읽히고 쓰이도록 보장한다.

메서드 4.11: EchoServer≫initialize 메서드

```
EchoServer>>initialize
    super initialize.
    isRunningLock := Mutex new.
    self isRunning: false
```

우리 서버의 라이프사이클을 관리하기 위해 두 개의 메서드, EchoServer≫start와 EchoServer≫stop를 소개했다. 가장 간단한 형태인 EchoServer≫stop 로 시작하며, 그 정의는 메서드 4.12에서 제공한다. 이는 단순히 isRunning 플래그를 false로 설정한다. EchoServer≫serve 메서드에서 서빙 루프를 중단하는 결과를 야기할 것이다 (메서드 4.13 참조).

메서드 4.12: EchoServer≫stop 메서드

```
EchoServer>>stop
    self isRunning: false
```

메서드 4.13: EchoServer≫serve 메서드

```
EchoServer>>serve
    [ [ self isRunning ]
        whileTrue: [ self interactOnConnection ] ]
        ensure: [ connectionSocket closeAndDestroy ]
```

서빙 프로세스의 활동은 serve 메서드에서 구현된다 (메서드 4.13 참조). 이것은 isRunning 플래그가 true인 동안 연결 상의 클라이언트와 상호작용한다. stop 이후에 서빙 프로세스는 연결 소켓을 파괴함으로써 종료된다. ensure: 메시지는 서빙 프로세스가 이례적으로 종료되더라도 파괴가 실행되도록 보장하는 역할을 한다. 그러한 종료는 예외(예: 네트워크 연결 끊김) 또는 사용자 액션(예: 프로세스 브라우저를 통해) 때문에 발생할 수도 있다.

메서드 4.14: EchoServer≫start 메서드

```
EchoServer>>start
    isRunningLock critical: [
        self isRunning ifTrue: [ ^ self ].
        self isRunning: true].
    connectionSocket := Socket newTCP.
    connectionSocket listenOn: 9999 backlogSize: 10.
    [ self serve ] fork
```

서빙 프로세스의 생성은 EchoServer≫start 가 책임진다 (메서드 4.14 마지막 행 참조). EchoServer≫start 메서드는 먼저 서버가 이미 실행되고 있는지를 확인한다. isRunning 플래그가 true로 설정된 경우 리턴하고, 그 외의 경우 연결 처리에 전념하는 TCP 소켓이 생성되어 포트 9999를 듣도록 만들어진다. 백로그 크기는 위에서 언급한 바와 같이 10으로 설정되므로, 시스템은 보류 중인 클라이언트 연결 요청을 10개까지 보관하도록 버퍼를 할당한다. 해당 값은 서버의 속도와 (VM과 하드웨어에 따라 좌우) 클라이언트 연결 요청의 최대 속도에 따라 좌우되는 상반관계(trade-off)에 해당한다. 백로그 크기는 어떤 연결 요청의 손실도 피하기에 충분히 커야 하지만 메모리의 낭비를 피하려면 너무 커서는 안 된다. 마지막으로 EchoServer≫start 메서드는 [self serve] 블록으로 fork 메시지를 전송함으로써 프로세스를 생성한다. 생성된 프로세스는 생성자(creator) 프로세스(예: 워크스페이스로부터 실행한 경우 UI 프로세스, EchoServer≫start 메서드를 실행하는 프로세스)와 우선순위가 동일하다.

메서드 4.15: EchoServer≫interactOnConnection 메서드

```
EchoServer>>interactOnConnection
    | interactionSocket |
    interactionSocket := connectionSocket waitForAcceptFor: 1 ifTimedOut: [^self].
    [self interactUsing: interactionSocket] fork
```

EchoServer≫serve 메서드(메서드 4.13 참조)는 연결된 클라이언트와 상호작용을 트리거한다. 이러한 상호작용은 EchoServer≫interactOnConnection 메서드(메서드 4.15 참조)에서 처리된다. 첫째로, 연결 소켓은 1초간 클라이언트 연결을 기다린다. 해당 시간 동안 어떤 클라이언트도 연결을 시도하지 않는 경우 단순히 리턴한다. 그 외의 경우 상호작용에 참여하는 다른 소켓을 결과로 받는다. 다른 클라이언트 연결 요청을 처리하기 위해 상호작용이 다른 프로세스에서 실행되므로 마지막 행에 fork가 포함된다.

메서드 4.16: EchoServer≫interactUsing: 메서드

```
EchoServer>>interactUsing: interactionSocket
    | receivedData |
    [ receivedData := interactionSocket receiveDataTimeout: 5.
        receivedData crLog.
        interactionSocket sendData: 'ECHO: ', receivedData
    ] ensure: [
        interactionSocket closeAndDestroy ]
```

EchoServer≫interactUsing: 메서드(메서드 4.16 참조)에서 구현된 바와 같이 클라이언트와의 상호작용은 클라이언트에 의해 제공된 데이터를 읽고 앞에 'ECHO:' 문자열을 붙여 다시 전송하는 것을 핵심으로 한다. 데이터를 교환했든 하지 않았든 (시간 만료) 상호작용 소켓이 파괴되도록 보장한다는 사실을 인지하는 것이 중요하다.

SocketStream

SocketStream은 TCP 소켓을 캡슐화하는 읽기-쓰기 스트림으로서, 기능 메서드 집합과 함께 버퍼링을 제공함으로써 데이터 교환을 수월하게 해준다. 이는 Socket 상에서 사용하기 쉬운 API 를 제공한다.

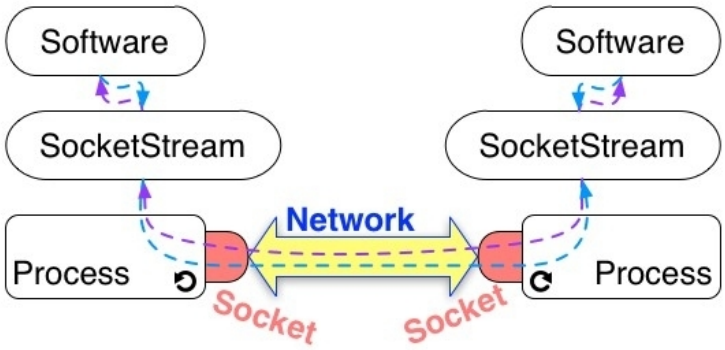


그림 4.4: SocketStream은 Socket을 수월한 방식으로 사용하도록 해준다.

SocketStream은 SocketStream class≫openConnectionToHost:port: 메서드를 이용해 생성이 가능하다. 호스트 주소나 이름, 그리고 포트를 제공함으로써 시스템의 기본 매개변수로 새

Socket을 초기화한다. 하지만 SocketStream class≫on: 메서드로 기존의 Socket 위에 SocketStream을 빌드할 수도 있는데, 이 방법을 선택 시 당신이 이미 설정한 데이터를 소켓으로 전송할 수 있다.

새 SocketStream을 이용해 유용한 메서드들이 있는 데이터를 수신할 수 있는데, 시간만료까지 데이터를 기다리는 receiveData, 데이터를 수신하지만 도달할 때까지 더 이상 기다리지는 않는 receiveAvailableData를 들 수 있겠다. IsDataAvailable 메서드는 데이터를 수신하기 전에 스트림 상에서 이용 가능한지 확인하도록 해준다.

데이터를 스트림에 두는 nextPut:, nextPutAll:, 또는 nextPutAllFlush: 메서드를 이용해 데이터를 전송하는 수도 있다. nextPutAllFlush: 메서드는 스트림에 데이터를 넣기 전에 다른 보류 중인 데이터를 비운다.

마지막으로 SocketStream이 사용을 완료하면 관련된 소켓을 완료하고 닫도록 close 메시지를 전송한다. SocketStream의 다른 유용한 메서드들은 아래에서 설명하겠다.

클래스 측에서의 SocketStream

아래 코드 조각을 이용해 클라이언트 측에서 소켓 스트림의 사용을 설명하겠다 (스크립트 4.17). 소켓 스트림을 이용해 웹페이지 첫 행을 얻는 방법을 보여준다.

스크립트 4.17: SocketStream을 이용해 웹 페이지의 첫 행을 얻기

```
| stream httpQuery result |
stream := SocketStream
    openConnectionToHostNamed: 'www.pharo-project.org'
    port: 80.
httpQuery := 'GET / HTTP/1.1', String crlf,
    'Host: www.pharo-project.org:80', String crlf,
    'Accept: text/html', String crlf.
[ stream sendCommand: httpQuery.
stream nextLine crLog ] ensure: [ stream close ]
```

첫 행은 새로 생성되어 제공된 서버로 연결된 소켓을 캡슐화하는 스트림을 생성한다. 이는 openConnectionToHostNamed:port: 메시지의 책임이다. 해당 메시지는 서버와 연결이 구축될 때까지 실행을 연기시킨다. 서버가 응답하지 않으면 소켓 스트림이 ConnectionTimedOut 예외를 시그널링한다. 이 예제는 사실 기본이 되는 소켓에 의해 시그널링된다. 기본 timeout delay는 45초다 (Socket class≫standardTimeout 메서드에서 정의됨). SocketStream≫timeout: 메서드를 이용해 다른 값을 선택할 수도 있다.

소켓 스트림이 서버로 연결되고 나면 HTTP GET 쿼리를 위조하여 전송한다. 스크립트 4.5(36 페이지)에 비교할 때 이번에는 (스크립트 4.17) 마지막 String crlf 를 건너뛰는데, SocketStream≫sendCommand: 메서드가 행의 끝을 표시하기 위해 전송된 데이터 다음에 CR과 LF 문자를 자동으로 삽입하기 때문이다.

요청한 웹 페이지의 수신은 우리의 소켓 스트림으로 nextLine 메시지를 전송하면 트리거된다. 데이터가 수신될 때까지 몇 초간 기다릴 것이다. 이후 데이터가 transcript에 표시된다. 연결이 안전하게 닫히도록 확보하였다.

이번 예제에서는 서버가 전송하는 응답의 첫 행만 표시했다. 하지만 소켓 스트림으로 upToEnd 메시지를 전송함으로써 html 코드를 포함해 전체 응답을 쉽게 표시할 수도 있다. 하지만 하나의 행을 표시할 때와 비교하면 좀 더 기다려야 함을 명심하라.

서버 측에서의 SocketStream

SocketStreams는 스크립트 4.18에 표시된 바와 같이 상호작용 소켓을 감싸기(wrap) 위해 서버 측에서 사용되기도 한다.

스크립트 4.18: SocketStream을 이용한 간단한 서버.

```
| connectionSocket interactionSocket interactionStream |
connectionSocket := Socket newTCP.
[
    connectionSocket listenOn: 12345 backlogSize: 10.
    interactionSocket := connectionSocket waitForAcceptFor: 30.
    interactionStream := SocketStream on: interactionSocket.
    interactionStream sendCommand: 'Greetings from Pharo Server'.
    interactionStream nextLine crLog.
] ensure: [
    connectionSocket closeAndDestroy.
    interactionStream ifNotNil: [interactionStream close]
]
```

소켓 스트림에 의존하는 서버는 여전히 소켓을 이용해 들어오는 연결 요청을 처리한다. 소켓 스트림은 클라이언트와의 상호작용을 위해 소켓이 생성되어야 실행된다. 소켓은 소켓 스트림에 래핑(wrap)되어 sendCommand: 또는 nextLine과 같은 메시지를 이용한 데이터 교환을 용이하게 만든다. 여기까지 완료되면 연결을 처리하는 소켓을 닫고 파괴하며, 상호작용 소켓 스트림을 닫는다. 상호작용 소켓 스트림은 기본이 되는 상호작용 소켓을 닫고 파괴하는 작업을 책임질 것이다.

Binary 모드 vs. Ascii 모드

교환된 데이터는 바이트 또는 문자로서 취급될 수 있다. 소켓 스트림이 binary를 이용해 바이트를 교환하도록 구성되면 바이트 배열과 같은 데이터를 전송 및 수신한다. 반대로 소켓 스트림이 acsii 메시지를 이용해 문자를 교환하도록 (기본 설정) 구성되면 데이터를 String으로 전송 및 수신한다.

아래 표현식을 이용해 시작되는 EchoServer(4.3 절 참고)의 인스턴스를 갖고 있다고 가정해보자.

```
server := EchoServer new.
server start.
```

소켓 스트림의 기본 행위는 전송과 수신 시 ascii 문자열을 처리하는 것이다. Binary 모드에서 행위는 스크립트 4.19에서 소개하였다. nextPutAllFlush: 메시지는 바이트 배열을 인자로서 수신한다. 이는 모든 바이트를 버퍼로 넣은 직후 전송을 트리거한다 (따라서 메시지명에 Flush가 포함된다). upToEnd 메시지는 서버가 다시 전송한 모든 바이트와 함께 배열을 응답한다. 이러한 메시지는 서버와의 연결이 닫힐 때까지 차단(block)함을 주목하라.

스크립트 4.19: 바이너리 모드에서 상호작용하는 SocketStream

```
interactionStream := SocketStream
46 Sockets
                        openConnectionToHostNamed: 'localhost'
                        port: 9999.
interactionStream binary.
interactionStream nextPutAllFlush: #[65 66 67].
interactionStream upToEnd.
```

클라이언트가 문자열(ascii 모드)을 관리하는지 혹은 바이트 배열(binary 모드)이 서버에 어떤 영향도 미치지 않는지 주목하라. 아스키 모드에서는 사실상 소켓 스트림이 ByteString의 인스턴스를 처리한다. 따라서 각 문자는 단일 바이트로 매핑한다.

데이터 구분하기(Delimiting Data)

SocketStream 은 그저 일부 네트워크에 대한 게이트웨이의 역할을 한다. 이는 어떤 구문도 제공하지 않고, 바이트를 전송하거나 읽는다. 교환된 데이터의 의미와 구성에 해당하는 구문(semantics)은 다른 객체에 의해 처리되어야 한다. 개발자들은 올바른 상호작용을 갖기 위해 사용할 프로토콜을 결정하고 상호작용의 양측에 강요해야 한다.

이를 위한 좋은 연습은 프로토콜을 구체화하는 것으로, 소켓 스트림을 감싸는 객체로서 구체화함을 의미한다. 프로토콜 객체는 교환된 데이터를 분석한 후 그에 따라 소켓 스트림에 보낼 메시지를 결정한다. 여느 대화든 그에 관여된 엔티티는 데이터를 바이트나 문자의 시퀀스로 정의하는 프로토콜이 필요하다. 전송자는 이러한 구성(organization)을 준수하여 수신자가 수신된 바이트 시퀀스로부터 유효한 데이터를 추출하도록 허용해야 한다.

한 가지 가능한 해결책은 각 데이터에 해당하는 바이트 또는 문자 사이에 삽입된 구분자(delimiter) 집합을 갖는 방법이다. 구분자의 예제로 ASCII 문자 CR과 LF의 시퀀스를 들 수 있다. 이 시퀀스는 SocketStream 클래스의 개발자들이 sendCommand: 메시지를 도입하던 때 유용하다고 간주된다. 해당 메서드(스크립트 4.5에 설명된)는 전송된 데이터 다음에 CR와 LF를 덧붙인다. CR 다음에 LF를 읽으면 수신자는 수신된 문자의 시퀀스가 완전하며 유효한 데이터로 안전하게 변환할 수 있음을 안다. 기능 메서드 nextLine(스크립트 4.17에 설명)은 CR+LF 시퀀스가 수신될 때까지 읽기를 실행하도록 SocketStream에 의해 구현된다. 하지만 어떤 문자나 바이트든 구분자로 사용 가능하다. 실제로 우리는 upTo: 메시지를 이용해 특정 문자/바이트까지 포함하는 문자/바이트를 모두 읽도록 소켓 스트림에게 요청할 수도 있다.

구분자를 사용하는 이점은 임의의 크기로 된 데이터를 처리한다는 데에 있다. 반면 한계를 찾기 위해 수신된 바이트나 문자를 분석할 필요가 있으므로 자원을 소모한다는 단점이 있다. 대안적 접근법으로, 고정된 크기의 chunk로 조직된 바이트 또는 문자를 교환하는 방법이 있다. 이러한 접근법은 보통 오디오나 비디오 컨텐트(content)의 스트리밍에 사용된다.

스크립트 4.20: 데이터를 chunk로 전송하는 컨텐트 스트리밍 소스

```
interactionStream := ”create an instance of SocketStream”.
contentFile := FileStream fileNamed: ’/Users/noury/Music/mySong.mp3’.
contentFile binary.
content := contentFile upToEnd.
chunkSize := 3000.
chunkStartIndex := 1.
[chunkStartIndex < content size] whileTrue: [
    interactionStream next: chunkSize putAll: content startingAt: chunkStartIndex.
    chunkStartIndex := chunkStartIndex + chunkSize.
]
interactionStream flush.
```

스크립트 4.20은 mp3 파일을 스트리밍하는 스크립트의 예제이다. 먼저 바이너리(mp3) 파일을 열고 upToEnd: 메시지를 이용해 그 내용을 모두 검색한다. 이후 루프를 실행하여 데이터를 3000 바이트의 chunk로 전송한다. 우리는 세 개의 인자를 취하는 next:putAll:startingAt: 메시지에 의존하는데, 세 가지 인자는 데이터 chunk의 크기(문자 또는 바이트 수), 데이터 소스(시퀀스 가능 컬렉션), chunk의 첫 번째 요소에 대한 색인이 해당한다. 해당 예제에서 우리는 컨텐트 컬렉션의 크기가 chunk 크기의 배수라고 가정한다. 물론 실제 배경에서 이러한 가정은 유지되지 않으며, chunk보다 작은 데이터의 마지막 부분을 처리해야 할 것이다. 이에 가능한 해결책으로는 누락된 바이트를 zero로 대체하는 방법이 있다. 게다가 메모리 내 모든 것을 먼저 로딩하는 방법은 실용적인 해결책이 아니며, 스트리밍 접근법이 대신 주로 사용된다.

스크립트 4.21: SocketStream을 이용해 데이터를 chunk로 읽기

```
| interactionStream chunkSize chunk |
interactionStream := SocketStream
                      openConnectionToHostNamed: ’localhost’
                      port: 9999.

interactionStream isDataAvailable ifFalse: [(Delay forMilliseconds: 100) wait].
chunkSize := 5.
[interactionStream isDataAvailable] whileTrue: [
   chunk := interactionStream next: chunkSize.
   chunk crLog.

].
interactionStream close.
’DONE’ crLog.
```

데이터를 chunk로 읽기 위해 SocketStream은 스크립트 4.21에 설명된 바와 같이 next: 메시지에 응답한다. 우리는 우리 머신의 포트 9999에서 크기가 5의 배수인 문자열을 전송하는 서버가 실행되고 있다고 간주한다. 연결 직후부터 데이터가 수신될 때까지 100 밀리초를 기다린다. 이후 5개 문자의 chunk로 데이터를 읽어와 Transcript 상에 표시한다. 따라서, 서버가 10개 문자로 된 ’HelloWorld’ 문자열을 전송할 경우, Transcript 상에서 첫 행에 Hello가, 두 번째 행에 World가 표시될 것이다.

네트워킹 실험을 위한 조언

클라이언트 측의 소켓과 소켓 스트림에 관련된 여러 절에서 우리는 웹 서버와의 상호작용을 예로 들었다. 따라서 HTTP Get 쿼리를 가상으로 만들어 서버로 전송한다. 간단하고 플랫폼

에 구애 받지 않는(flatform agnostic; 크로스 플랫폼을 의미) 실험을 위해 이러한 예제를 선택하였다. 실제 규모의 애플리케이션에서 HTTP를 수반하는 상호작용은, 기본 Pharo 배포판에 속한 Zinc HTTP Client/Server 라이브러리와 같이 더 높은 수준의 라이브러리를 이용해 코딩되어야 한다[4].

네트워크 프로그래밍에서는 복잡성이 쉽게 증가한다. Pharo 외부의 툴박스를 이용해 이상 행위의 근원을 확인해야 하는 경우가 종종 있다. 이번 절에서는 저수준 네트워크 운용을 다루기 위한 수많은 Unix 유틸리티를 열거하겠다. Unix 머신(Linux, Mac OS X) 또는 Cygwin(Windows)를 이용하는 독자들은 nc(또는 netcat), netstat, lsof를 테스트에 이용할 수 있다.

nc (netcat)

nc는 TCP (기본 프로토콜)과 UDP 모두를 위한 클라이언트 또는 서버를 설정하도록 해준다. 이는 stdin의 내용을 상대측으로 재전송한다. 아래는 포트 9090을 듣는 로컬 머신 상에서 서버로 'Hello from a client'를 전송하는 방법을 보여준다.

```
echo Hello from a client | nc 127.0.0.1 9090
```

아래 명령행은 포트 9090을 듣고, 연결할 첫 번째 클라이언트에게 'Hi from server'를 전송하는 서버를 시작시킨다. 이는 상호작용 후 종료된다.

```
echo Hi from server | nc -l 9090
```

옵션 -k를 이용해 서버의 실행을 유지하는 수도 있다. 하지만 앞에 붙은 echo 문자가 생성한 문자열은 연결해야 할 첫 번째 클라이언트에게만 전송된다. 아니면, nc 서버가 당신이 텍스트를 입력하는 동안 텍스트를 전송하도록 만드는 대안적 방법이 있다. 아래 명령행을 평가하기만 하면 된다:

```
echo nc -lk 9090
```

서버를 시작한 단말기에 텍스트를 입력해보라. 이후 다른 단말기에서 클라이언트를 실행시켜라. 당신이 입력한 텍스트가 클라이언트 측에 표시될 것이다. 이 두 개의 행위는 (서버 측에 텍스트를 입력하고 클라이언트를 시작) 원하는 횟수만큼 반복할 수 있다.

클라이언트와 서버 간 연결을 좀 더 지속적으로 만들어 좀 더 상호작용을 증진시킬 수도 있다. 아래 명령행을 평가함으로써 클라이언트는 모든 행("Enter"로 끝나는)을 전송한다. EOF 신호(ctl-D)를 전송하면 종료될 것이다.

```
echo cat | nc -l 9090
```

[4] http://zn.stfx.eu/zn/index.html

netstat

위의 명령은 자신의 컴퓨터의 소켓과 네트워크 인터페이스에 관한 다양한 정보를 제공한다. 이것은 유용한 정보를 걸러내는 데에 적절한 옵션을 사용하도록 다수의 통계를 제공한다. 아래 명령행은 tcp 소켓의 상태와 그 주소를 표시하도록 해준다. 포트 번호와 주소는 점으로 구분됨을 주목한다.

```
netstat -p tcp -a -n
```

lsof

lsof 명령은 시스템에 열린 파일을 모두 열거한다. Unix에서는 모든 것이 파일이기 때문에 소켓도 물론 포함된다. 이미 netstat가 있다면 lsof 가 왜 유용할까? 그 이유는 lsof가 프로세스와 소켓 간 링크(link)를 표시하기 때문이다. 따라서 자신의 프로그램과 연관된 소켓을 찾을 수 있는 것이다.

아래 명령행이 제공하는 예제는 TCP 소켓을 열거한다. n과 P 옵션은 강제로 lsof로 하여금 호스트 주소와 포트를 숫자로 표시하도록 만든다.

```
lsof -nP -i tcp
```

요약

본 장에서는 TCP 소켓과 소켓 스트림을 이용해 네트워크 클라이언트와 서버를 모두 개발하는 방법을 소개하였다. 그리고 네트워크 프로그래밍의 필수 도구를 검토해보았다:

- 소켓은 Socket 클래스의 저수준 양방향 통신 게이트웨이 인스턴스다.
- 소켓을 기반으로 한 프로그래밍은 항상 하나의 서버와 하나 또는 그 이상의 클라이언트를 수반한다.
- 서버는 클라이언트가 보내는 요청을 기다린다.
- sendData: 와 receiveData 메시지는 데이터를 전송하고 수신하기 위한 소켓 프리미티브다.
- 최대 대기 시간은 receiveDataTimeout: 을 이용해 설정 가능하다.
- SocketStream는 TCP 소켓을 캡슐화하는 buffered read-write 스트림이다.
- 네트워크 DNS는 장치명을 숫자형 인터넷 주소로 변환하는 NetNameResolver 클래스를 통해 접근이 가능하다.
- Unix 도 디버깅과 테스트를 목적으로 유용한 네트워킹 유틸리티를 몇 가지 제공한다.

서론에서 언급하였듯 우리는 수준이 높고 기능 메서드를 제공하는 소켓 스트림의 사용을 권한다. 그들은 각기 AidaWeb과 Seaside 웹 프레임워크에서 사용하는 Swazoo 와 Kom 웹 서버처럼 프로젝트에서 성공적으로 사용된다.

그럼에도 불구하고 만일 통신하는 이미지들에 다른 객체들이 분산되어야 하는 애플리케이션을 갖고 있다면 스켓 스트림은 여전히 저수준으로 유지된다. 그러한 소프트웨어에서는 개발자들이 인자와 결과를 직렬화함으로써 원격 객체들 간 전달되는 메시지를 처리할 필요가 있다. 뿐만 아니라 분산된 쓰레기 수집도 처리해야 한다. 객체가 원격 객체에 의해 참조되는 경우 해당 객체는 파괴되어선 안 된다. 이렇게 반복되지만 사소하지 않은 문제들은 rST[5]와 같은 객체 요구 매개자(ORB)가 해결한다. ORB는 개발자를 네트워크 문제로부터 해방시켜주므로, 원격 객체들 간에 교환되는 메시지를 이용하여 원격 통신을 표현하도록 허용한다.

[5] http://smalltalkhub.com/#!/~CAR/rST/

제 5 장

Settings 프레임워크

***Alain Plantec** 참여 (alain.plantec@univ-brest.fr)*

애플리케이션이 성숙해가면서 기본 선택영역 색상, 기본 폰트 또는 기본 폰트 크기와 같은 변형(variation)을 제공해야 하는 경우가 종종 생긴다. 때때로 그러한 변형은 가능한 소프트웨어 맞춤설정(customization)에 대한 사용자 선호도를 나타낸다. Pharo의 1.1 버전 발매 이후부터 Pharo는 사용자 설정을 관리하기 위해 Settings 프레임워크를 포함하여 사용해오고 있다. 애플리케이션은 Settings를 이용해 그 구성을 제공한다. Settings는 Pharo의 개인설정을 관리하는 것으로 제한되지 않고 어떤 애플리케이션이든 사용을 권장한다. Settings의 장점으로는 개입적(intrusive)이지 않고, 소프트웨어의 모듈식 분해(modular decomposition)를 지원하며, 해당 애플리케이션이 시작된 이후에도 애플리케이션으로 추가가 가능하다는 점을 들 수 있다. Settings 프레임워크를 이제부터 살펴보겠다.

Settings 아키텍처

설정은 개인설정 정의와 조작에 대한 객체 지향 접근법을 지원한다. 그 말은 다음의 의미를 지닌다.

1. 각 패키지나 서브시스템은 그 고유의 맞춤설정 포인트를 정의해야 한다 (보통 변수나 클래스 변수로서 표현). 서브시스템의 코드는 그러한 맞춤설정 값에 자유롭게 접근하고, 그 값을 이용해 개인설정을 반영하도록 그 행위를 변경한다.
2. 서브시스템은 그 개인설정을 설명하여 최종 사용자가 조작할 수 있다. 하지만 서브시스템의 코드는 어떤 경우에도 그 행위를 조정하기 위해 setting 객체를 명시적으로 참조하지 않을 것이다.

서브시스템의 제어 흐름은 Settings를 수반하지 않는다. 이것이 바로 Pharo 1.0에서 개인설정 시스템(preference system)과 Settings 의 주요 차이점이다.

어휘

개인설정(preference)은 변수 값으로서 관리되는 특별한 값이다. 기본적으로 이러한 기본설정 값은 싱글톤의 인스턴스 변수나 클래스 변수에 보관되며, 단순한 접근자의 사용을 통해 직접 관리된다. Pharo는 사용자 인터페이스 테마, 데스크탑 배경색, 또는 부울 플래그와 같은 수많은 개인설정을 포함하여 소리의 사용을 허용하거나 금한다. 5.3 절에서는 개인설정을 정의하는 방법을 보여줄 것이다.

Setting 은 개인설정 값의 선언(설명)이다. 설정 브라우저를 통한 브라우징과 업데이트를 위해서는 개인설정 값이 설정에 의해 설명되어야 한다. 그러한 설정은 구체적 pragma로 태그된 특정 메서드에 의해 빌드된다. 이러한 구체적 pragma는 메서드를 자동으로 설정으로서 식별하는 데에 사용되는 분류 태그 역할을 한다 (그림 5.1 참고). 5.3절은 설정을 선언하는 방법을 설명한다.

Pharo 사용자들은 기존의 개인설정을 살펴보고 특정 사용자 인터페이스를 통해 그 값을 최종적으로 변경한다. 이것이 바로 5.2절에 소개된 Settings Browser의 주요 역할이다.

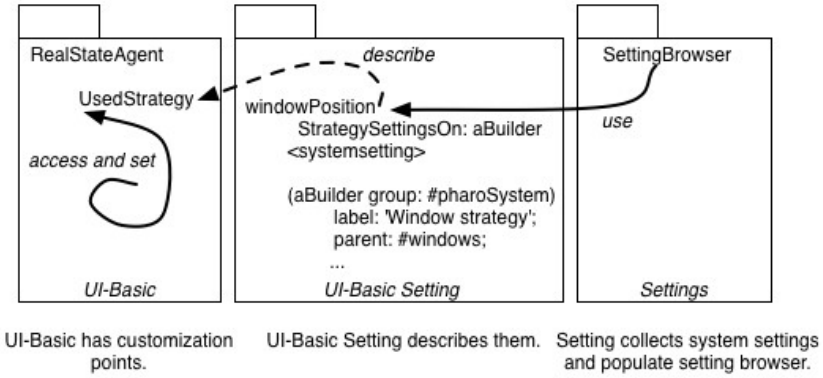


그림 5.1: UI-Basic Setting.

그림 5.1은 Settings가 구축한 아키텍처의 중요한 요점, 즉 Settings 패키지는 언로딩이 가능하다는 점과 개인설정을 정의하는 패키지는 Settings 패키지에 의존하지 않는다는 점을 보여준다. 이러한 아키텍처는 아래의 요점에 따라 지원된다:

맞춤설정 포인트. 각 애플리케이션은 맞춤설정 포인트를 정의한다. 그림 5.1에서 패키지 UI-Basic의 RealStateAgent 클래스는 창(windows)이 나타나는 장소를 정의하는 클래스 변수 UsedStrategy를 정의한다. 패키지 UI-Basic 의 흐름은 모듈식(modular)이며 완전(self-contained)하기 때문에 RealStateAgent 클래스는 settings 프레임워크에 의존하지 않는다. RealStateAgent 클래스는 매개변수화되도록 설계되어 왔다.

맞춤설정 포인트의 설명. Settings 프레임워크는 UsedStrategy setting의 설명을 지원한다. 그림 5.1에서 UI- Basic Setting 패키지는 메서드를 지원하는데 클래스 RealStateAgent 또는

다른 클래스에 대한 확장 (extension) 이 될 수도 있다. 여기서 요점은 setting을 선언하는 메서드가 Setting 클래스를 직접 참조하지 아니하고 빌더를 이용해 setting을 설명한다는 점이다. 이런 방식을 통해 설명은 참조를 소개하지 않고도 UI-Basic 패키지에서도 표시될 수 있다.

사용자 표현에 관한 설정 수집하기. Settings 패키지는 사용자가 자신의 개인설정을 변경 시 여는 Settings Browser와 같은 설정을 관리하는 데 사용될 툴들을 정의한다. Settings Browser는 설정들을 수집하고 그 설명을 이용해 개인설정의 값을 변경한다. 의존성과 프로그램의 제어 흐름은 항상 패키지 Settings로부터 개인설정을 가진 패키지를 향할 뿐, 그 반대로는 작용하지 않는다.

Settings Browser

그림 5.2에 소개된 Settings Browser 는 주로 현재 선언된 설정을 모두 살펴보고 관련 개인설정 값을 변경하도록 허용한다.

 Settings Browser를 열기 위해서는 World 메뉴 (World▷System▷Settings) 를 사용하거나 아래 표현식을 평가하면 된다:

```
SettingBrowser open
```

설정은 중간 패널의 여러 개의 트리로 표시된다. 설정 검색과 필터링은 상단 툴바에서 이용 가능하고, 하단 패널은 현재 선택된 설정의 설명(좌측 하단 패널)과 현재 패키지 집합(우측 하단 패널)을 표시한다.

개인설정 값 찾아보기와 변경하기

설정 선언은 중간 패널에서 볼 수 있는 트리에 조직된다. 어떤 설정에 관한 설명을 확인하려면 설정을 클릭하면 되는데, 설정을 선택하면 좌측 하단 패널이 선택된 설정에 관한 정보로 업데이트된다.

개인설정 값의 변경은 간단히 브라우저를 통해 이루어지며, 각 행에는 우측에 위젯이 위치하므로 당신은 이 값을 업데이트할 수 있다. 위젯의 종류는 개인설정 값의 실제 타입에 따라 달라진다. 개인설정 값은 어떤 종류든 가능하지만 설정 브라우저 (setting browser) 는 현재 Boolean, Color, FileName, DirectoryName, Font, Number, Point, String 타입에 대한 특정 입력 위젯만 표시할 수 있는 실정이다. 드롭리스트, 비밀번호 필드, 또는 슬라이더를 이용해 범위 입력 위젯도 사용 가능하다. 물론 가능한 위젯의 목록은 닫혀 있는데, 설정 브라우저가 새로운 종류의 개인설정 값을 지원하도록 만들거나 다른 입력 위젯을 사용하도록 만드는 것이 가능하기 때문이다. 이와 관련된 요점은 5.8절에서 설명한다.

실제 설정 타입이 String, FileName, DirectoryName, Number, Point 중 하나라면 사용자는 값을 변경하기 위해 편집 가능한 드롭리스트 위젯에 텍스트를 입력해야 한다. 이런 경우, 리턴 키 (또는 cmd-s를 이용) 를 눌러 입력 값을 확인해야 한다. 그러한 설정 값이 자주 변경되는

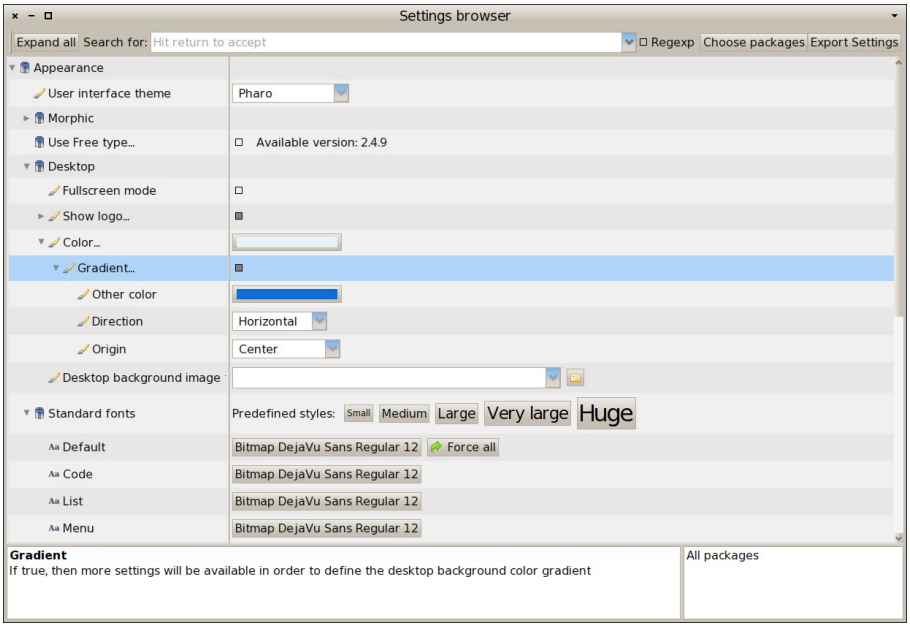


그림 5.2: Settings Browser.

경우 드롭리스트 위젯이 유용한데, 한 번의 클릭으로 이전에 입력한 값을 검색 및 이용할 수 있기 때문이다! 게다가 FileName이나 DirectoryName 인 경우 파일명이나 디렉터리명 선택자 대화상자를 열기 위한 버튼이 추가된다.

그 외의 가능한 실행은 모두 컨텍스트 메뉴에서 접근 가능하다. 선택된 설정에 따라 차이가 있을지도 모른다. 두 가지 가능한 버전을 그림 5.3에 표시하겠다.

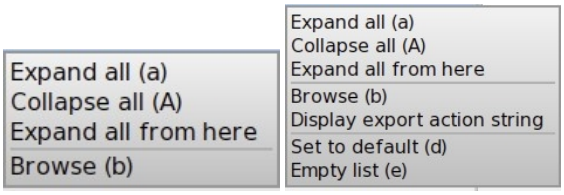


그림 5.3: 상황에 맞는 팝업메뉴.

- **Expand all (a):** 모든 설정 트리 노드를 재귀적으로 확장한다. 키보드 단축키 cmd-a 를 통해서도 접근 가능하다.
- **Collapse all (a):** 모든 설정 트리 노드를 재귀적으로 축소한다. 키보드 단축키 cmd-A 를 통해서도 접근 가능하다.
- **Expand all from here:** 현재 선택된 설정 트리 노드를 재귀적으로 확장한다.

- **Browse (b):** 설정을 선언하는 메서드 상에 시스템 브라우저를 연다. 키보드 단축키 cmd-b 를 통하거나 설정을 더블 클릭해도 접근 가능하다. 설정 구현을 변경하길 원하거나, 몇 가지 예제를 통해 프레임워크를 이해하도록 구현 방식을 살펴보고자 할 때 유용하다 (설정의 선언 방식은 5.3절에 설명되어 있다).
- **Display export action string:** 설정은 스타트업 액션으로 내보낼 수 있으며, 이 메뉴 옵션은 스타트 업 액션(start-up action)이 어떻게 코딩되는지 표시하도록 해준다 (스타트업 액션 관리는 5.7절에 설명된다).
- **Set to default (d):** 선택된 설정 값을 기본 값으로 설정한다. 가령 설정의 효과를 살펴보기 위해 이리저리 시도해보다가 결국 기본 값으로 돌아가기로 결정할 때 매우 유용하다.
- **Empty list (e):** 입력 위젯이 편집 가능한 드롭리스트인 경우 이 메뉴 항목은 기록된 목록을 비움으로써 이전에 입력된 값을 잊어버리도록 해준다.

설정 검색과 필터링

Pharo는 수많은 설정을 포함하므로 그 중에 하나를 찾기란 지루한 작업이 될 수 있다. 이 때 Settings Browser 상단 바의 검색 텍스트 필드에 텍스트를 입력함으로써 설정 목록을 필터링 할 수 있다. 그러면 당신이 입력한 텍스트가 포함된 설명이나 이름의 설정이 표시될 것이다. ”Regexp” 체크박스를 체크할 경우 텍스트는 정규 표현식이 될 수 있다.

설정 목록의 필터링 방식으로, 패키지별로 선택하는 방법이 있다. ”Choose package” 버튼을 누르면 일부 설정이 선언된 패키지의 목록이 담긴 대화상자가 열린다. 하나 또는 여러 개를 선택하면 선택된 패키지에 선언된 설정만 표시된다. 하단 우측 텍스트 패인은 선택된 패키지의 이름으로 업데이트된다.

Pharo를 어디서, 언제 사용하느냐에 따라 개인설정을 반복하여 변경해야 하는 경우가 있다. 예를 들어, 다른 사람들에게 시범을 보이고 있다면 더 큰 폰트를 원할 것이고, 직장이라면 프록시를 설정해야 할 테지만 집에서는 이 어느 것도 필요가 없다. 당신이 어디에 있고 무엇을 하느냐에 따라 개인설정 집합을 변경해야 한다는 점은 너무 지루한 일이 될 수도 있다. Settings Browser를 이용하면 현재 개인설정 값 집합을 명명된 스타일로 저장하여 후에 재로딩할 수 있다. 설정 스타일 관리는 ??(이 부분 확인 부탁드립니다.) 절에서 소개한다.

설정 선언하기

Pharo의 모든 전역적 개인설정은 Settings Browser를 이용해 살펴보거나 변경할 수 있다. 개인설정은 싱글톤의 인스턴스 변수이거나 클래스 변수인 것이 보통이다. Settings Browser에서 그 값을 변경할 수 있으려면 설정을 그에 맞게 선언해야 한다. 설정은 특정 클래스 메서드에 의해 선언되어야 하며 이는 다음과 같이 구현되어야 한다: 빌더를 인자로 취하고, <systemsettings> pragma로 태그된다.

aBuilder라는 인자는 설정 선언을 빌드하는 데에 있어 API 또는 facade의 역할을 한다. pragma는 Settings Browser 가 현재 설정 선언을 동적으로 발견하도록 허용한다.

중요한 점은 설정 선언이 패키지 특정적이어야 한다는 사실이다. 다시 말해, 각 패키지는 고유의 설정을 선언해야 할 책임이 있다. 특정 패키지의 경우, 구체적인 설정이 그 클래스들 중

하나 또는 여러 개에 의해 선언되거나 동료(companion) 패키지에 의해 구현된다. (Pharo 1.0의 경우처럼) 클래스 또는 패키지를 정의하는 전역적 설정은 없다. 직접적인 이점은 패키지가 로딩되면 그 설정도 자동으로 로딩되고, 패키지가 언로딩되면 그 설정도 자동으로 언로딩된다는 점이다. 뿐만 아니라 Setting 선언은 어떤 Setting 클래스도 참조해선 안 되고 빌더 인자를 참조해야 한다. 이를 통해 당신의 애플리케이션이 Settings로부터 의존적이지 않고, 극적으로 작은 footprint 애플리케이션을 정의하고자 할 때 Setting을 제거할 수 있도록 보장할 수 있다.

caseSensitiveFinds 개인설정의 예제를 살펴보자. 이는 부울형(boolean) 개인설정으로, 텍스트 검색에 사용된다. 그 값이 true라면 다음 검색은 대 · 소문자에 민감해진다. 이 개인설정은 TextEditor 클래스의 CaseSensitiveFinds 클래스 변수에 보관된다. 그 값은 쿼리 및 변경이 가능한데, 아래와 같이 각각 TextEditor class≫caseSensitiveFinds 와 TextEditor class≫caseSensitiveFinds: 에 의해 실행된다.

```
TextEditor class>>caseSensitiveFinds
    ^ CaseSensitiveFinds ifNil: [CaseSensitiveFinds := false]

TextEditor class>>caseSensitiveFinds: aBoolean
    CaseSensitiveFinds := aBoolean
```

이러한 개인설정(예: CaseSensitiveFinds 클래스 변수)에 대한 설정(setting)을 정의하고 그것을 Settings Browser에서 확인 및 변경 가능하도록 아래와 같은 메서드가 구현된다. 결과는 그림 5.4의 스냅샷에 표시된다.

```
CodeHolderSystemSettings class>>caseSensitiveFindsSettingsOn: aBuilder
    <systemsettings>
    (aBuilder setting: #caseSensitiveFinds)
        target: TextEditor;
        label: 'Case sensitive search' translated;
       description: 'If true, then the "find" command in text will always make its searches in
        a case-sensitive fashion' translated;
        parent: #codeEditing.
```

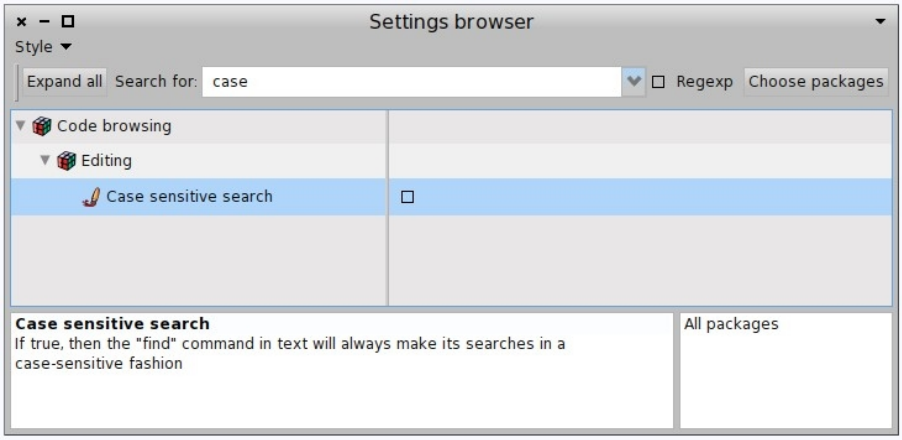


그림 5.4: caseSensitiveFinds 설정.

이제 설정 선언을 자세히 연구해보자.

헤더

```
CodeHolderSystemSettings class>>caseSensitiveFindsSettingsOn: aBuilder
{\dots}
```

해당 클래스 메서드는 CodeHolderSystemSettings 클래스에서 선언된다. 해당 클래스는 설정에 집중하며 설정 선언 외에는 어떤 것도 포함하지 않는다. 그러한 클래스를 정의하는 것은 의무가 아니며, 사실 어떤 클래스든 설정 선언을 정의할 수 있다. 우리는 레이어링(layering)을 목적으로 한 개인설정 정의 중 하나가 아니라 그와 다른 패키지로 설정 선언이 패키징되는 방식으로 정의하였다.

이 메서드는 빌더를 인자로 취한다. 그리고 해당 객체는 setting 빌딩을 위한 facade에 대해 API의 역할을 하므로, 메서드의 내용은 설정의 하위트리를 선언하고 구성하기 위해 빌더로 전송하는 메시지들로 구성되어야 한다.

pragma

설정 선언은 <systemsettings> pragma로 태그된다.

```
CodeHolderSystemSettings class>>caseSensitiveFindsSettingsOn: aBuilder
    <systemsettings>
    ...
```

사실상 settings browser가 열리면 먼저 <systemsettings> pragma로 모든 메서드를 검색함으로써 모든 설정 선언을 수집한다. 또한 Settings Browser가 열려 있는 동안 설정 선언 메서드를 컴파일할 경우, 자동으로 새 설정으로 업데이트된다.

설정 구성

설정은 인자로서 전달되는 식별자와 함께 setting: 메시지를 빌더로 전송함으로써 구현된다. 식별자가 #caseSensitiveFinds인 예제를 소개하겠다:

```
CodeHolderSystemSettings class>>caseSensitiveFindsSettingsOn: aBuilder
    <systemsettings>
    (aBuilder setting: #caseSensitiveFinds)
    ...
```

빌더에게 setting: 메시지를 전송하면 그 자체가 설정 노드의 래퍼(wrapper)인 setting node builder가 생성된다. 기본적으로 인자로서 전달된 기호는 Settings Browser가 기본설정 값을 얻기 위해 사용하는 선택자로 간주된다. 개인설정 값을 변경하기 위한 선택자는 기본적으로 getter 선택자에 콜론을 추가하여 빌드된다 (예: 본문에서는 caseSensitiveFinds: 가 되겠다). 이러한 선택자들은 기본적으로 메서드가 구현된 클래스에 해당하는 대상(target)으로 전송된다 (예: CodeHolderSystemSettings). 따라서 caseSensitiveFinds와 caseSensitiveFinds: 접근자들이 CodeHolderSystemSettings 에 구현될 경우 이 단일 행으로 된 설정 선언으로 충분하다.

사실상 기본 initializations가 당신의 요구를 충족하지 못하는 경우도 자주 발생한다. 물론 자신의 특정 상황을 고려하기 위해 설정 노드 구성을 조정하는 방법도 있다. 가령, caseSensitiveFinds 설정에 해당하는 getting/setter 접근자는 TextEditor 클래스에 구현된다. 이후 우리는 대상이 TextEditor가 되도록 명시적으로 설정해야 하는데, 이는 업데이트된 정의에 표시하듯이 인자로서 전달된 대상 클래스 TextEditor와 함께 target: 메시지를 설정 노드로 전송함으로써 가능하다.

```
CodeHolderSystemSettings class>>caseSensitiveFindsSettingsOn: aBuilder
    <systemsettings>
    (aBuilder setting: #caseSensitiveFinds)
        target: TextEditor
```

위와 같이 간략한 버전이라도 온전히 작동하며 그림 5.5에서 보이는 바와 같이 Settings Browser가 컴파일하고 고려하기에 충분하다.

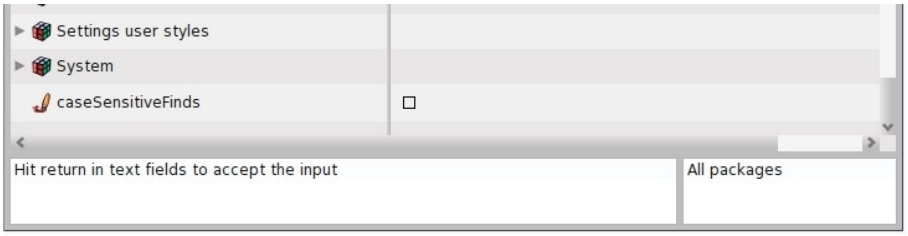


그림 5.5: caseSensitiveFinds 설정의 단순한 첫 버전.

불행히도 그 표현은 사용자 친화적이라고 할 수 없는데, 그 이유는 다음과 같다:

- settings browser에 표시된 라벨이 식별자이다 (그로 접근하기 위해 접근자를 빌드하는데 사용된 기호).
- 해당 설정에 이용할 수 있는 설명이나 해설이 없다.
- 새로운 설정이 설정 트리의 루트에 추가된다.

위와 같은 단점을 극복하기 위해서는 각각 문자열을 인자로서 취하는 label: 과 description: 메시지를 이용해 라벨과 설명으로 자신의 설정 노드를 좀 더 구성할 수 있다.

```
CodeHolderSystemSettings class>>caseSensitiveFindsSettingsOn: aBuilder
    <systemsettings>
    (aBuilder setting: #caseSensitiveFinds)
        target: TextEditor;
        label: 'Case sensitive search' translated;
        description: 'If true, then the "find" command in text will always make its searches
        in a case-sensitive fashion' translated;
        parent: #codeEditing.
```

라벨과 설명 문자열로 translated 를 전송해야 함을 잊지 말아야 하는데, 이를 전송 시 다른 언어로 해석 기능이 크게 향상될 것이다.

분류와 설정 트리 구성과 관련해 향상 방법이 여러 가지가 있는데, 다음 절에서 상세히 다루도록 하겠다.

대상 (target) 에 관한 추가 정보

설정의 대상은 개인설정 값을 얻고 변경하기 위한 수신자인데, 대부분의 경우 클래스에 해당한다. 실제로 개인설정 값은 클래스 변수에 보관되는 것이 보통이다. 따라서 클래스 측 메서드는 설정으로 접근하기 위한 접근자로서 사용된다.

하지만 수신자는 싱글톤 객체가 될 수도 있다. 다수의 개인설정이 사실상 이에 해당한다. 예를 들자면, Free Type 폰트 개인설정은 모두 **FreeTypeSettings** 싱글톤의 인스턴스 변수에 보관된다. 따라서 여기에서 수신자는 **FreeTypeSettings** 인스턴스로, 아래 표현식을 평가함으로써 얻을 수 있다:

```
FreeTypeSettings current
```

이에 따라 사용자는 이 표현식을 이용해 그에 상응하는 설정의 대상을 구성할 수 있겠다. 예를 들어, #glyphContrast 개인설정은 아래와 같이 선언이 가능하다:

```
(aBuilder setting: #glyphContrast)
    target: FreeTypeSettings current;
    label: 'Glyph contrast' translated;
    ...
```

간단하지만 안타깝게도 이와 같은 싱글톤 대상을 선언하는 것은 좋은 생각이 아니다. 이 선언은 Setting style 기능과 호환이 가능하다 (??절 참조)(이 부분도 확인 부탁드립니다). 이러한 경우 싱글톤을 얻기 위해 대상 클래스로 전송하는 메시지 선택자와 대상 클래스를 따로 표시해야 한다. 따라서 아래 예제와 같이 targetSelector: 메시지를 이용해야 한다:

```
(aBuilder setting: #glyphContrast)
    target: FreeTypeSettings;
    targetSelector: #current;
    label: 'Glyph contrast' translated;
    ...
```

기본값에 관한 추가 정보

Settings Browser가 설정 입력 위젯을 빌드하는 방법은 실제 개인설정 값의 타입에 따라 좌우된다. 개인설정에 대한 값으로 nil 을 갖는 것은 Settings Browser 에 문제가 되는데, 사용해야 할 입력 위젯을 알아내지 못하기 때문이다. 따라서 기본적으로 개인설정을 양호한 입력 위젯으로 적절하게 표시하려면 항상 nil이 아닌 값으로 설정되어야 한다. 기본값은 평상시와 같이 초기화를 통해 개인설정으로 설정이 가능하며, 개인설정의 접근자 메서드에 프로그램화된 느긋한 초기화나 #initialize 메서드를 이용하면 되겠다.

Settings Browser 와 관련해 최선의 방법은 느긋한 초기화를 사용하는 것이다 (5.3절에 제공된 #caseSensitiveFinds 개인설정 예제 참조). 사실 5.2절에서 설명한 바와 같이 Settings Browser 컨텍스트 메뉴에서 개인설정 값을 기본값으로 리셋하거나 모든 개인설정 값을 전역적으로 리셋하는 방법도 있다. 이는 개인설정 값을 nil 로 리셋되도록 설정하면 가능하다. 그 결과 개인설정은 그에 집중하는 접근자를 이용하는 즉시 자동으로 기본값에 설정된다.

접근자가 구현되는 방식을 항상 변경할 수 있는 것은 아니다. 아마도 개인설정 접근자가 당신이 변경을 허용하지 않는 다른 패키지에서 유지되기 때문일 것이다. 이 방법이 아니면, 아래

예에서 볼 수 있듯이 설정 노드로 default: 메시지를 전송함으로써 설정의 선언으로부터 기본 값을 나타내는 방법도 있다.

```
CodeHolderSystemSettings class>>caseSensitiveFindsSettingsOn: aBuilder
    <systemsettings>
    (aBuilder setting: #caseSensitiveFinds)
    default: true;
    ...
```

자신의 설정 구성하기

Settings Browser 내에서 설정은 트리로 구성되며, 관계된 설정들은 같은 부모의 자식으로 표시된다.

부모 선언하기

자신의 설정을 다른 설정의 자식으로 선언하는 가장 간단한 방법은 부모 설정의 식별자를 인자로서 전달되도록 하여 parent: 메시지를 사용하는 것이다. 아래 예제에서 부모 노드는 #codeEditing 식별자로 선언된 기존의 노드이다.

```
CodeHolderSystemSettings class>>caseSensitiveFindsSettingsOn: aBuilder
    <systemsettings>
    (aBuilder setting: #caseSensitiveFinds)
        target: TextEditor;
        label: 'Case sensitive search' translated;
     description: 'If true, then the "find" command in text will always make its searches in a case-sensitive fashion' translated;
        parent: #codeEditing.
```

#codeEditing 노드 또한 시스템 내 어딘가에서 선언되어 있다. 예를 들어 그룹으로 정의되기도 하는데, 이를 이제부터 살펴보도록 하자.

그룹 선언하기

그룹은 어떤 값도 없는 단순한 노드로서, 자식들의 그룹화에만 사용된다. #codeEditing 이 식별하는 노드는 빌더에게 인자로서 전달되는 식별자와 함께 group: 메시지를 전송하면 생성된다. 그림 5.4에서와 같이 #codeEditing 노드는 그 자체가 #codeBrowsing 노드의 자식으로 선언되기 때문에 루트에 위치하지 않음을 명심하라.

```
CodeHolderSystemSettings class>>codeEditingSettingsOn: aBuilder
    <systemsettings>
    (aBuilder group: #codeEditing)
        label: 'Editing' translated;
        parent: #codeBrowsing.
```

하위트리 선언하기

고유의 설정을 기존 노드의 자식으로 선언할 수 있다면 패키지가 기존 표준 설정을 강화하고자 할 때 매우 유용하게 작용할 수 있겠다. 하지만 애플리케이션 특정적인 설정의 경우 매우 지루한

일이 되기도 한다.

따라서 하나의 메서드에 설정의 하위트리를 직접 선언하는 것도 가능하다. 보통 루트 그룹은 애플리케이션 설정에 대해 선언되고, 자식 설정들도 동일한 메서드 내에서 선언된다. 이는 루트 그룹으로 with: 메시지를 전송하면 간단히 실행 가능하다. with: 메시지는 블록을 인자로서 취한다. 이 블록에서 모든 새로운 설정은 루트 그룹의 자식으로서 암묵적으로 선언된다 (with: 메시지의 수신자).

그림 5.6: 하나의 메서드에 하위트리 선언하기 : Configurable formatter 설정 예제.

예로 그림 5.6을 살펴보면 재팩토링 브라우저 구성 가능 포맷터를 볼 수 있다. 이러한 설정의 하위트리는 아래 제공된 RBConfigurableFormatter class≫settingsOn: 메서드에 완전히 선언된다. 두 개의 자식, #formatCommentWithStatements 와 #indentString 으로 #configurableFormatter 라는 새 루트 그룹을 선언함을 볼 수 있을 것이다.

```
RBConfigurableFormatter class>>settingsOn: aBuilder
    <systemsettings>
    (aBuilder group: #configurableFormatter)
        target: self;
        parent: #refactoring;
        label: 'Configurable Formatter' translated;
        description: 'Settings related to the formatter' translated;
        with: [
            (aBuilder setting: #formatCommentWithStatements)
                label: 'Format comment with statements' translated.
            (aBuilder setting: #indentString)
                label: 'Indent string' translated]
```

선택적 하위트리

특정 개인설정의 값에 따라 표시할 의미가 없는 설정은 숨기길 원할지도 모른다. 가령, 데스크탑의 배경색이 평범하다면 gradient 배경에 관련된 설정은 보일 필요가 없다. 하지만 사용자가 gradient 배경을 원한다면 두 번째 색상, gradient 방향, gradient 시작점(origin) 설정이 표시되어야 한다. 그림 5.7을 살펴보자 :

- 좌측에 Gradient 위젯은 체크가 해제되었는데 실제 값이 false임을 의미하며, 이런 경우 자식이 없다는 뜻이다.
- 우측에 Gradient 위젯이 체크되어 있는데 설정 값이 true로 설정되었으며 그 결과 gradient 배경을 설정하는 데에 유용한 설정이 표시된다.

선택적 설정을 처리하기란 간단한데, 선택적 설정이 부울형 부모 설정의 자식으로서 선언되면 끝이다. 이런 경우, 자식 설정은 부모의 값이 true일 때에만 표시된다. 데스크탑 gradient 예제에서 설정은 아래와 같이 PolymorphSystemSettings 에 선언된다 :

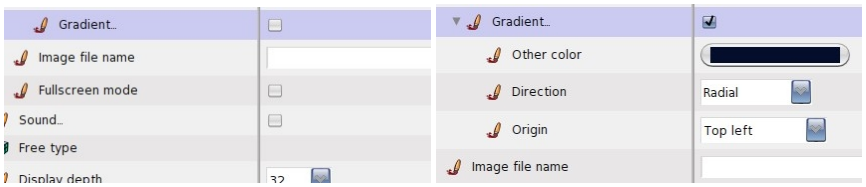


그림 5.7: 선택적 하위트리의 예제. 우측 – 어떤 gradient도 선택되지 않았다. 좌측 – gradient가 선택되어 추가 개인설정이 이용 가능하다.

```
(aBuilder setting: #useDesktopGradientFill)
    label: 'Gradient';
    description: 'If true, then more settings will be available to define the desktop
    background color gradient';
    with: [
        (aBuilder setting: #desktopGradientFillColor)
            label: 'Other color';
        description: 'This is the second color of your gradient (the first one is given by the "Color" setting' translated.
        (aBuilder pickOne: #desktopGradientDirection)
            label: 'Direction';
            domainValues: {#Horizontal. #Vertical. #Radial}.
        (aBuilder pickOne: #desktopGradientOrigin)
            label: 'Origin';
            domainValues: {
                'Top left' translated -> #topLeft. ...
```

부모 설정 값은 PolymorphSystemSettings class≫useDesktopGradientFill 를 평가함으로써 주어진다. 이것이 true를 리턴하면 자식에 해당하는 #desktopGradientFillColor, #desktopGradientDirection, #desktopGradientOrigin 가 표시된다.

설정 정렬하기

기본적으로 형제(sibling) 설정들은 라벨의 알파벳 순으로 정렬된다. 이러한 기본 행위를 변경하길 원하는 수도 있다. 설정 정렬은 두 가지 방식으로 변경이 가능한데, 기본 정렬을 단순히 금지시키거나 정렬을 분명히 명시하는 방법이다.

그림 5.8: 범위 설정에 대한 예제.

아래 #appearance 그룹의 예제에서와 같이 부모 노드로 noOrdering 메시지를 전송함으로써 어떤 정렬도 실행되지 않도록 지시할 수 있다. 이후 그 자식들은 선언된 순서대로 정렬된다.

```
appearanceSettingsOn: aBuilder
    <systemsettings>
    (aBuilder group: #appearance)
        label: 'Appearance' translated;
       description: 'All settings concerned with the look''n feel of your system' translated;
        noOrdering;
        with: [... ]
```

인자로서 전달되는 숫자와 함께 order: 메시지를 전송하여 형제들 간 설정 노드의 정렬을 나타낼 수도 있다. 숫자는 Integer 또는 Float이 가능하다. 정렬 번호에 해당하는 노드는 항상 다른 노드들보다 앞에 위치해야 하며, 각 정렬 번호에 따라 정렬된다. 순서가 항목에 주어지면 다른 형제들에게는 어떤 정렬도 적용되지 않는다.

예를 들어, #standardFonts 그룹이 선언되는 방식을 살펴보자:

```
(aBuilder group: #standardFonts)
    label: 'Standard fonts' translated;
    target: StandardFonts;
    parent: #appearance;
    with: [
        (aBuilder launcher: #updateFromSystem)
            order: 1;
            targetSelector: #current;
            script: #updateFromSystem;
            label: 'Update fonts from system' translated.
        (aBuilder setting: #defaultFont)
            label: 'Default' translated.
        (aBuilder setting: #codeFont)
            label: 'Code' translated.
        (aBuilder setting: #listFont)
    ...
```

이 예제에서 런처 #updateFromSystem이 첫 번째 노드가 되도록 선언되고, #defaultFont, #codeFont, #listFont 식별자로 된 다른 형제들은 선언된 순서로 위치한다.

좀 더 정밀한 값 도메인 제공하기

기본적으로 개인설정에 가능한 값 집합은 제한되지 않으며 개인설정의 실제 타입에 의해 주어진다. 가령, 색상 개인설정의 경우 위젯은 당신이 원하는 색상을 선택하도록 허용하며, 숫자의 경우 위젯은 사용자가 어떤 숫자든 입력하도록 허용한다. 하지만 일부 사례에서는 값의 특정 집합만이 바람직하다. 가령 표준 브라우저나 사용자 인터페이스 테마 설정의 경우 유한의 클래스 집합에서 선택해야 하며, 자유형(free type) 캐시 크기의 경우 0-50000 범위만 허용된다. 이러한 경우 위젯이 특정 값만 수락한다면 좀 더 편할 것이다. 이러한 문제를 해결하기 위해 도메인 값 집합을 범위나 값 리스트로 제한할 수 있다.

범위 설정 선언하기

예를 들어, 그림 5.8에 소개된 전체 화면 여백(margin) 개인설정을 고려해보자. 그 값은 창을 확대할 때 창 주위에 허용되는 여백의 크기를 픽셀로 표현한다.

그 값은 정수지만 -100이나 5000로 설정하는 것은 말이 되지 않는다. 최소 -5부터 최대 100까지가 양호한 범위의 값이 된다. 이 범위를 이용해 설정 위젯을 제약할 수도 있겠다. 아래 예제에서 보이듯이 간단한 설정에 비해 두 가지의 차이점이 있다:

새 설정 노드가 setting: 메시지 대신 range: 메시지를 이용해 생성된다.

유효한 범위는 설정 노드로 range: 메시지를 전송하여 주어지고, Interval이 인자로서 주어진다.

```
screenMarginSettingOn: aBuilder
    <systemsettings>
    (aBuilder range: #fullScreenMargin)
        target: SystemWindow;
        parent: #windows;
        label: 'Full screen margin' translated;
        description: 'Specify the amount of space that is let around a windows when it''s
        opened fullscreen' translated;
        range: (-5 to: 100).
```

리스트에서 선택하기

개인설정 값이 특정 값의 리스트 중 하나로 제한된 경우 그것을 선언하여 settings browser가 드롭리스트를 사용하도록 만드는 것이 가능하다. 드롭리스트는 사전에 정의된 유효 값으로 초기화된다. 가령, 창 위치 전략(window position strategy) 예를 고려해보자. 그에 해당하는 위젯이 settings browser 내에서 실행되는 모습을 그림 5.9에서 소개한다. 허용되는 값은 'Reverse Stagger', 'Cascade', 'Standard' 가 있다.

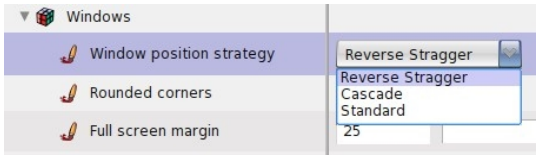


그림 5.9: 리스트 설정의 예제.

아래 예제는 창 위치 전략 설정을 간소화한 선언을 보여준다.

```
windowPositionStrategySettingsOn: aBuilder
    <systemsettings>
    (aBuilder pickOne: #usedStrategy)
        label: 'Window position strategy' translated;
        target: RealEstateAgent;
        domainValues: #(#'Reverse Stagger' #Cascade #Standard)
```

간단한 설정에 비해 유일한 두 가지 차이점은 다음과 같다:

- #setting: 메시지 대신 pickOne: 메시지로 새 설정 노드가 생성된다.
- 새로 선언된 설정 노드로 domainValues: 메시지를 전송함으로써 승인된 값의 리스트가 주어지고, Collection이 인자로서 주어진다 (기본적으로 첫 번째 인자).

이러한 창 전략 예제와 관련해, 개인설정으로 설정된 값은 #'Reverse Stagger' 이거나 #Cascade 또는 #Standard가 될 것이다.

다행히 이러한 값들은 그다지 편리하지 못하다. 가령 프로그래머는 일종의 Strategy 객체나 직접적으로 선택자 역할을 할 수 있는 Symbol로 된 다른 값을 원할지도 모른다. 사실상 이 두 번째 해결책은 RealEstateAgent 클래스 관리자들에 의해 선택되었다. RealEstateAgent usedStrategy가 리턴한 값을 살펴보면 결과가 #'Reverse Stagger', #Cascade, #Standard 중 하나가 아니라 다른 기호(symbol)임을 눈치챌 것이다. 그리고 창 위치 전략 설정이 어떻게 구현되었는지 살펴보면 선언이 이전에 주어진 기본 해결책과 다르다는 사실을 알게 될 것이다: domainValues: 인자는 단순한 Symbols의 배열이 아니라 아래 선언에서 볼 수 있듯이 Associations의 배열이다:

```
windowPositionStrategySettingsOn: aBuilder
    <systemsettings>
    (aBuilder pickOne: #usedStrategy)
        ...
        domainValues: {'Reverse Stagger' translated -> #staggerFor:initialExtent:world:. '
        Cascade' translated -> #cascadeFor:initialExtent:world:. 'Standard' translated ->

        #standardFor:initialExtent:world:};
```

Settings Browser 관점에서 보면 리스트 내용은 정확히 동일하며 사용자는 그 차이를 구별할 수가 없는데, Associations의 배열이 domainValues: 에게 인자로서 주어진다면 Associations의 키는 사용자 인터페이스에 대해 사용되기 때문이다.

개인설정 자체의 값과 관련해 RealEstateAgent usedStrategy를 살펴보면 결과가 #staggerFor:initialExtent:world:, #cascadeFor:initialExtent:world:, 그리고 #standardFor:initialExtent:world: 중에 해당하는 값임을 눈치챌 것이다. 사실 Associations의 값은 설정에 가능한 모든 실제 값을 계산하는 데에 사용된다.

가능한 값의 리스트는 어떤 종류든 가능하다. 또 다른 예로, 사용자 인터페이스 테마 설정이 PolymorphSystemSettings 클래스에서 어떻게 선언되는지를 살펴보자:

```
(aBuilder pickOne: #uiThemeClass)
    label: 'User interface theme' translated;
    target: self;
    domainValues: (UITheme allThemeClasses collect: [:c | c themeName -> c])
```

이 예제에서 domainValues: 는 연관(associations)의 배열을 취하며, Settings Browser가 열릴 때마다 계산된다. 각 연관은 테마명을 키로, 테마를 구현하는 클래스를 값으로 하여 구성된다.

스크립트 시작하기

외부 구성 툴을 시작(launch)하거나 스크립트의 도움을 받아 특정 패키지 또는 시스템을 구성하도록 허용하길 원한다고 가정해보자. 그러한 상황에서는 런처(launcher)를 선언할 수 있다. 런처는 일반 설정처럼 라벨로 표시되는데, 일반 설정과 다른 점은 라벨에 어떤 값도 입력되지 않는다는 사실이다. 대신 Launch 라고 라벨이 붙은 버튼이 Settings Browser 에 통합되고, 버튼을 클릭하면 연관된 스크립트가 시작된다.

예를 들어, 시스템이 True Type Fonts를 사용하려면 호스트 시스템에서 이용 가능한 모든 폰트를 수집하여 시스템을 업데이트해야만 하는데, 이는 아래 표현식을 평가함으로써 가능하다:

```
FreeTypeFontProvider current updateFromSystem
```

Settings Browser로부터 해당 스크립트를 실행하는 것도 가능하다. 해당하는 런처는 그림 5.10에서 소개하겠다. 그러한 런처의 통합은 꽤 간단하다. 그저 그에 맞는 설정을 선언하기만 하면 된다! 예를 들어, TT 폰트에 대한 런처가 선언되는 방식을 살펴보자.

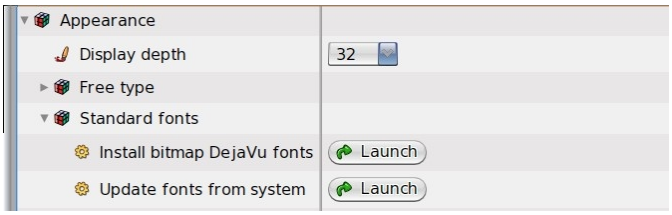


그림 5.10: 런처의 예제.

```
GraphicFontSettings class>> standardFontsSettingsOn:
    <systemsettings>
    (aBuilder group: #standardFonts)
        ...
        (aBuilder launcher: #updateFromSystem) ...
            target: FreeTypeFontProvider;
            targetSelector: #current;
            script: #updateFromSystem;
            label: 'Update fonts from system' translated.
```

간단한 설정에 비교하면 두 가지 차이점이 있다:

- 새 설정 노드는 빌더로 launcher: 메시지를 전송함으로써 생성된다.
- script: 메시지는 인자로서 전달된 스크립트의 선택자와 함께 설정 노드로 전송된다.

스타트 업 액션 (Start-up actions) 관리

구식이라는 이유로 Pharo에서 제거된 개인설정도 많지만 여전히 많은 수의 개인설정이 남아 있는 상태다. Settings Browser의 사용이 쉽다고는 하지만 가령 하위집합에 대한 개인설정만 하더라도 매번 새 이미지를 작업해야 하기 때문에 지루할 수 있다. 이에 대한 해결책은 당신이 선호하는 선택들을 모두 설정하도록 스크립트를 구현하는 방법이다. 이를 위한 최선의 방법은 그러한 목적에 맞는 구체적인 클래스를 생성하는 것이다. 이후 패키지에 포함시켜 새로운 이미지를 원할 때마다 재로딩하는 것이 가능하다. 이러한 클래스 유형을 Setting style이라 부른다.

Setting styles 를 관리하는 데에 *Settings Browser* 는 두 가지 방면에서 유용하다. 첫째, 개인설정 값을 어떻게 변경하는지 발견하도록 도와주고, 둘째, 당신을 위해 특정 스타일을 생성 및 업데이트 할 수 있다.

스크립트 설정

개인설정 변수는 접근자 메서드로 모두 접근이 가능하기 때문에 간단한 스크립트에서 개인설정 집합을 초기화하는 것도 당연히 가능하다. 단순화를 위해 Setting 스타일로 구현해보도록 하자.

예를 들어, 배경색을 변경하고 모든 폰트를 기본값보다 크게 설정하도록 스크립트를 구현할 수 있겠다. 그에 맞는 Setting 스타일 클래스를 생성해보자. 이를 MyPreferredStyle이라 부를 수 있겠다. 스크립트는 MyPreferredStyle의 메서드에 의해 정의된다. 이 선택자는 설정과 관련된 스크립트 평가에 대한 표준 도구(standard hook)이기 때문에 해당 메서드를 loadStyle이라 부르겠다.

```
MMyPreferredStyle>>loadStyle
    | f n |
    "Desktop color"
    PolymorphSystemSettings desktopColor: Color white.
    "Bigger font"
    n := StandardFonts defaultFont. "get the current default font"
    f := LogicalFontfamilyName: n familyName pointSize: 12. "font for my preferred size"
    StandardFonts setAllStandardFontsTo: f "reset all fonts"
```

PolymorphSystemSettings는 PolyMorph 와 관련된 모든 설정이 선언되는 클래스다. StandardFonts 는 Pharo 의 기본 폰트를 관리하는 데에 사용되는 클래스다.

이제 데스크탑 색상 설정이 PolymorphSystemSettings에서 선언되고 DefaultFonts 클래스가 폰트 관리를 허용함을 어떻게 알아낼 것인지를 질문해야 한다. 좀 더 일반적으로 말하자면, 이 모든 설정은 어디에서 선언되고 관리되는지가 되겠다.

이에 대한 답은 꽤 간단한데, Settings Browser를 사용하면 될 일이다! 5.2절에서 설명하였듯, cmd-b 또는 항목을 더블 클릭하면 현재 설정 노드의 선언에 브라우저가 열린다. 컨텍스트 메뉴를 사용해도 된다. 선언을 살펴보면 개인설정 값에 대한 선택자와 대상 클래스를 (개인설정 변수가 보관된) 제공할 것이다.

다음으로, MyPreferredStyle 자체가 시스템에서 로딩될 때 StyleMyPreferredStyle≫#loadStyle가 자동으로 실행되길 원한다. 이 목적을 위해 우리가 유일하게 해야 할 일은 MyPreferredStyle 클래스에 대한 initialize 메서드를 구현하는 일이다:

```
MyPreferredStyle class>>initialize
    self new loadStyle
```

Settings Browser에 스타일 통합하기

어떤 스크립트든 Settings Browser에 통합하여 후에 로딩, 브라우징 또는 그로부터 제거가 가능하다. 이 목적에 부합하려면 그에 대한 이름을 선언하고 Settings Browser가 발견하도록 확실히 해두기만 하면 된다. 자신의 스타일 클래스에서 클래스 측에 styleName 이라는 메서드를 구현하라. 앞 절의 예제에서는 아래와 같이 구현되어야겠다:

```
MyPreferredStyle class>>styleName
    ”The style name used by the SettingBrowser”
    <settingstyle>
    ^ 'My preferred style'
```

MyPreferredStyle class≫styleName는 어떤 인자도 취하지 않고 자신의 스타일 이름을 String 으로서 리턴해야 한다. *<settingstyle>* pragma를 이용해 Settings Browser 로 하여금 MyPreferredStyle이 설정 스타일 클래스라는 사실을 알도록 한다.

해당 메서드가 컴파일되고 나면 Setting Browser를 열어 *Style* 탭 메뉴를 팝업시킨다. 그림 5.11과 같이, 자신의 스타일명으로 구성된 리스트가 포함된 대화상자가 열릴 것이다.

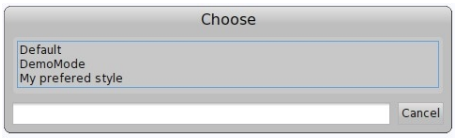


그림 5.11: 자신만의 스타일이 표시된 스타일 로드 대화창.

Settings Browser 확장하기

5.2절에 설명한 바와 같이 *Settings Browser* 는 기본적으로 간단한 개인설정 타입을 관리할 수 있다. 이러한 기본값만으로도 보통은 충분하다. 하지만 더 복잡한 개인설정 값의 처리 능력이 매우 유용하게 작용하는 상황들이 몇 가지 있다.

예를 들어 텍스트 선택영역(text selection) 개인설정을 중심적으로 살펴보자. 일차 선택영역, 3가지 유형의 선택적 텍스트 선택영역, 이차 선택영역, 찾기 및 바꾸기 선택영역, 선택영역 바(selection bar)가 있다. 모든 선택영역에 대해 배경색을 설정 가능하다. 일차와 이차 선택영역, 그리고 찾기 및 바꾸기 선택영역에 대한 텍스트 색상도 선택이 가능하다.

선택영역 설정을 개별적으로 선언하기

지금까지는 기본 가능성에 따라 설정을 텍스트 선택영역의 각 특성에 따라 선언이 가능하여 그에 해당하는 개인설정을 Settings Browser로부터 개별적으로 변경할 수 있었다. 특정 유형의 선택영역에 선언된 설정은 모두 설정 그룹의 자식으로 그룹화할 수 있다. 곧바로 적용시켜보면, 선택적 텍스트 선택영역에 대해 단순한 그룹 대신 부울형(boolean) 설정을 이용할 수 있겠다.

예를 들어, 이차 선택영역을 살펴보자. 해당 유형의 텍스트 선택영역은 선택적이며, 사용자는 그에 대한 배경색과 텍스트 색상을 설정할 수 있다. 그에 해당하는 개인설정은 ThemeSettings의 인스턴스 변수로서 선언된다. 그들의 값은 연관된 ThemeSettings 인스턴스를 얻음으로써 현재 테마로부터 읽고 변경하기가 가능하다. 따라서 두 가지 색상 설정은 아래와 같이 #useSecondarySelection 부울형 설정의 자식으로 선언될 수 있겠다.

```
(aBuilder setting: #useSecondarySelection)
    target: UITheme;
    targetSelector: #currentSettings;
    label: 'Use the secondary selection' translated;
    with: [
        (aBuilder setting: #secondarySelectionColor)
            label: 'Secondary selection color' translated.
        (aBuilder setting: #secondarySelectionTextColor)
            label: 'Secondary selection text color' translated].
```

그림 5.12는 Settings Browser 에서 이러한 설정 선언들을 보여준다. 모양과 느낌은 깔끔하지만 두 가지가 관찰된다:

1. 각 선택영역 유형마다 세 개의 행을 취하는데, 각 선택영역을 보는 데에 너무 많은 수직 공간을 차지하여 약간 불편한 감이 있다.
2. 기본이 되는 모델이 명시적으로 디자인되지 않았고, 한 종류의 선택영역에 대한 설정이 Settings Browser 에 그룹화되었으나 그에 대한 개인설정 값은 ThemeSettings의 구분된 인스턴스 변수로서 선언되었다. 다음 절에서는 첫 번째 해결책을 더 나은 디자인으로 개선하는 방법을 살펴보겠다.

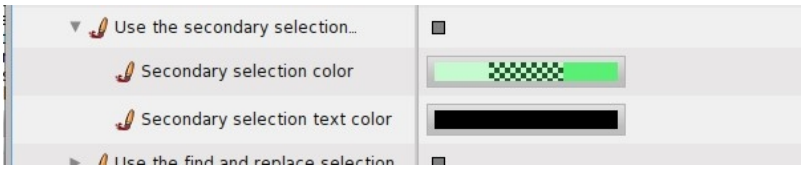


그림 5.12: 기본 설정값으로 선언된 2차 선택영역 설정.

개선된 선택영역 개인설정 디자인

텍스트 선택영역 개인설정의 개념을 디자인하는 것이 더 나은 해결책이 되겠다. 그러면 각 선택영역 개인설정마다 관리하는 데에 세 개가 아니라 하나의 값만 갖게 된다. 텍스트 선택영역 개인설정은 기본적으로 두 가지 색상으로 구성되는데, 하나는 배경색, 나머지는 텍스트 색상이다. 일차 선택영역을 제외하고 각 선택영역은 선택적이다. 그렇다면 아래와 같이 텍스트 선택영역 개인설정을 디자인할 수 있을 것이다:

```
Object subclass: #TextSelectionPreference
    instanceVariableNames: 'backgroundColor textColor mandatory used'
    classVariableNames: 'FindReplaceSelection PrimarySelection SecondarySelection
        SelectionBar'
    poolDictionaries: ''
    category: 'Settings-Tools'
```

TextSelectionPreference는 네 개의 인스턴스 변수로 구성된다. 그 중 두 개는 색상과 관련된다. mandatory 인스턴스 변수가 false로 설정되었다면 사용된 부울형 인스턴스 변수는

변경이 가능하다. 대신 mandatory가 true로 설정되었다면 사용된 인스턴스 변수는 true로 설정되고 변경이 불가하다.

TextSelectionPreference 에도 네 개의 클래스 변수가 있는데, 가능한 텍스트 선택영역 개인설정 유형마다 하나씩 해당한다. getters와 setters 또한 Settings Browser 로부터 이러한 개인설정을 관리할 수 있도록 구현될 것이다. 예로 PrimarySelection를 살펴보자:

```
TTextSelectionPreference class>>primarySelection
    ^ PrimarySelection
        ifNil: [PrimarySelection := self new
            textColor: Color black;
            backgroundColor: (Color blue alpha: 0.5);
            mandatory: true;
            yourself]
```

mandatory 속성이 true로 초기화되었음을 눈치챌 것이다.

선택영역 바 개인설정의 또 다른 예는 다음과 같다:

```
TextSelectionPreference class>>selectionBar
    ^ SelectionBar
        ifNil: [SelectionBar := self new
            backgroundColor: Color lightBlue veryMuchLighter;
            mandatory: false;
            yourself]
```

여기서 개인설정은 선택적으로, 그리고 어떤 텍스트 색상도 없이 선언됨을 눈치챌 것이다.

이러한 개인설정이 Settings Browser로부터 변경 가능하려면 두 가지 메서드를 선언해야 한다. 첫 번째는 설정 선언을, 두 번째는 뷰(view)를 구현하기 위함이다.

설정 선언은 아래와 같이 구현된다:

```
TextSelectionPreference class>>selectionPreferenceOn: aBuilder
    <systemsettings>
    (aBuilder group: #selectionColors)
        label: 'Text selection colors' translated;
        parent: #appearance;
        target: self;
        with: [(aBuilder setting: #primarySelection) order: 1;
            label: 'Primary'.
        (aBuilder setting: #secondarySelection)
            label: 'Secondary'.
        (aBuilder setting: #findReplaceSelection)
            label: 'Find/replace'.
        (aBuilder setting: #selectionBar)
            label: 'Selection bar']
```

보다시피 이 선언에서 새로운 내용은 전혀 없다. 변한 내용이라면 개인설정의 값이 사용자가 정의한 클래스라는 점이다. 사실 사용자 정의 또는 애플리케이션 특정적 개인설정 클래스의 경우 해야 할 일은 뷰에 대해 보완(supplementary) 메서드를 하나 구현하는 것 뿐이다. 이 메서드는 settingInputWidgetForNode: 로 명명되어야 하며, 클래스 메서드로서 구현되어야 한다.

SettingInputWidgetForNode: 메서드는 Settings Browser 에 대한 입력 위젯을 빌드하는 책임을 갖고 있다. 해당 메서드는 SettingDeclaration을 인자로 취한다. SettingDeclaration은 기본적으로 모델이며, 그 인스턴스들은 Settings Browser가 관리한다.

각 SettingDeclaration 인스턴스는 개인설정 값 홀더의 역할을 한다. 사실 Settings Browser에서 볼 수 있는 각 설정은 내부적으로 SettingDeclaration 인스턴스에 의해 표현된다.

우리 예제에서 텍스트 선택영역 개인설정마다 그 색상을 변경 가능하고, 선택영역이 선택적인 경우 활성화/해제를 원한다. 색상과 관련해서는, 선택영역 개인설정 값에 따라 배경색만 항상 표시된다. 사실 개인설정 값의 텍스트 색이 nil인 경우, 이는 텍스트 색상을 갖고 있는 것이 의미가 없음을 의미하므로, 그에 해당하는 색상 선택자가 빌드되지 않는다.

settingInputWidgetForNode: 메서드는 아래와 같이 구현 가능하다:

```
TextSelectionPreference class>>settingInputWidgetForNode: aSettingDeclaration
    | preferenceValue backColorUI usedUI uiElements |
    preferenceValue := aSettingDeclaration preferenceValue.
    usedUI := self usedCheckboxForPreference: preferenceValue.
    backColorUI := self backgroundColorChooserForPreference: preferenceValue.
    uiElements := {usedUI. backColorUI},
        (preferenceValue textColor
            ifNotNil: [ { self textColorChooserForPreference: preferenceValue } ]
            ifNil: [{}]).
    ^ (self theme newRowIn: self world for: uiElements)
        cellInset: 20;
        yourself
```

이 메서드는 몇 가지 기본 요소를 한 행으로 추가하여 그 행을 리턴한다. 첫 번째를 보면 #preferenceValue 를 SettingDeclaration으로 전송함으로써 실제 개인설정 값, 즉 TextSelectionPreference의 인스턴스를 얻게 됨을 눈치챌 수 있을 것이다. 이후 사용자 인터페이스 요소를 실제 TextSelectionPreference 인스턴스를 기반으로 빌드할 수 있다.

첫 번째 요소는 checkbox 이거나 #usedCheckboxForPreference: 호출에 의해 리턴된 빈 공간이다. 해당 메서드는 아래와 같이 구현된다:

```
TextSelectionPreference class>>usedCheckboxForPreference: aSelectionPreference
    ^ aSelectionPreference optional
        ifTrue: [self theme
            newCheckboxIn: self world
            for: aSelectionPreference
            getSelected: #used
            setSelected: #used:
            getEnabled: #optional
            label: ''
            help: 'Enable or disable the selection']
        ifFalse: [Morph new height: 1;
            width: 30;
            color: Color transparent]
```

다음 요소들은 두 개의 색상 선택자들이다. 예를 들어, 배경색 선택자는 아래와 같이 빌드된다:

```
TextSelectionPreference class>>backgroundColorChooserForPreference:
    aSelectionPreference
    ^ self theme
    newColorChooserIn: self world
    for: aSelectionPreference
    getColor: #backgroundColor
    setColor: #backgroundColor:
    getEnabled: #used
    help: 'Background color' translated
```

이제 Settings Browser 에서 사용자 인터페이스는 이전 버전에서 각 선택영역마다 세 개의 행이 아니라 그림 5.13에서 보이는 바와 같이 하나의 행으로만 표시된다.

요약

이번 장에서는 개인설정을 모듈식으로 관리하기 위한 새로운 프레임워크, Settings를 소개하였다. Settings에서 중점은 모듈식 제어 흐름을 지원한다는 데에 있어서, 패키지가 맞춤설정 포인트를 정의하는 데 책임이 있고, 이를 국부적으로 사용할 수 있으며, Settings를 이용해 그러한 맞춤설정 포인트를 설명하는 것이 가능하다. 마지막으로 Settings Browser는 그러한 설정 설명들을 수집하여 사용자에게 제시한다. 이후 Settings Browser 로부터 맞춤설정된 패키지로 제어의 흐름이 이동한다.

그림 5.13: 특정 개인설정 클래스를 이용해 구현된 텍스트 선택영역 설정.

제 6 장

Pharo에서의 정규 표현식

***Oscar Nierstrasz** 참여(oscar.nierstrasz@acm.org)*

정규 표현식은 Perl, Python, Ruby와 같은 많은 스크립팅 언어에서 널리 사용된다. 정규 표현식은 특정 패턴에 일치하는 문자열을 식별하고, 입력이 예상된 포맷에 따르는지 검사하며, 문자열을 새로운 포맷으로 재작성하는 데에 유용하다. Pharo 또한 Vassili Bykov가 기여한 Regex 패키지 때문에 정규 표현식을 지원한다. Regex는 기본적으로 Pharo에 설치된다.

정규 표현식[1]은 문자열 집합에 일치하는 템플릿이다. 예를 들어, 정규 표현식 'h.∗o'는 문자열 'ho, 'hiho', 'hello'에 매치하겠지만 'hi' 또는 'yo'에는 매치하지 않을 것이다. Pharo에서 아래와 같이 볼 수 있다:

```
'ho' matchesRegex: 'h.\(\ast{}\)o' → true
'hiho' matchesRegex: 'h.\(\ast{}\)o' → true
'hello' matchesRegex: 'h.\(\ast{}\)o' → true
'hi' matchesRegex: 'h.\(\ast{}\)o' → false
'yo' matchesRegex: 'h.\(\ast{}\)o' → false
```

본 장은 몇 가지 클래스를 발전시켜 웹 사이트의 매우 단순한 사이트 맵을 생성하는 작은 지침용 예제로 시작하겠다. 정규 표현식을 이용하여 (I) HTML 파일을 식별하고, (ii) 파일의 전체 경로명에서 경로명을 빼내고, (iii) 사이트 맵에 대한 각 웹 페이지를 추출하고, (iv) 웹 사이트의 루트 디렉터리로부터 그것이 포함하는 HTML 파일에 대한 상대 경로를 생성할 것이다. 지침용 예제를 완료한 후에는 대체적으로 Regex 패키지에 Vassili Bykov가 제공한 문서를 바탕으로 패키지[2]에 대한 완전한 설명을 제공하겠다.

[1] http://en.wikipedia.org/wiki/Regular_expression
[2] 원본 문서는 RxParser의 클래스 측에서 찾아볼 수 있다.

지침용 예제 – 사이트 맵 생성하기

우리가 할 일은 하드 드라이브에 국부적(locally)으로 보관된 웹 사이트에 대한 사이트 맵을 생성시킬 간단한 애플리케이션을 작성하는 일이다. 사이트 맵은 링크의 텍스트로서 문서의 제목을 이용해 웹 사이트 내 각 HTML 파일에 대한 링크를 포함할 것이다. 또한 링크는 웹 사이트의 디렉터리 구조를 반영하도록 할 것이다.

웹 디렉터리 접근하기

 자신의 기계에 웹 사이트가 없다면 테스트베드(test bed) 역할을 하도록 몇 개의 HTML 파일을 로컬 디렉터리로 복사하라.

두 개의 클래스, WebDir와 WebPage를 개발하여 디렉터리와 웹 페이지를 나타내도록 할 것이다. 우리의 웹 사이트를 포함하는 루트 디렉터리를 가리킬 WebDir의 인스턴스를 생성하는 것이 의도이다. 이곳으로 makeToc 메시지를 전송하면 그 내부의 파일과 디렉터리를 살펴보고 사이트 맵을 형성할 것이다. 이후 웹 사이트 내 모든 페이지의 링크를 포함하는 toc.html이라는 새 파일이 생성될 것이다.

이 때 유의해야 할 점이 하나 있다: 각 **WebDir** 과 **WebPage** 는 웹 사이트의 루트에 대한 경로를 기억하여 루트에 상대적인 링크를 적절하게 생성할 수 있도록 해야 한다.

 webDir와 homePath 인스턴스 변수가 있는 WebDir 클래스를 정의하고, 적절한 초기화(initialization) 메서드를 정의한다. 또한 아래와 같이 클래스 측 메서드를 정의하여 사용자에게 자신의 컴퓨터 상에서 웹 사이트의 위치를 입력하라고 표시될(prompt) 것이다.

```
WWebDir>>setDir: dir home: path
    webDir := dir.
    homePath := path

WebDir class>>onDir: dir
    ^ self new setDir: dir home: dir pathName

WebDir class>>selectHome
    ^ self onDir: FileList modalFolderSelector
```

마지막 메서드는 열어야 할 디렉터리를 선택하기 위해 브라우저를 연다. 이제 WebDir selectHome의 결과를 검사한다면 자신의 웹 페이지를 포함하는 디렉터리를 입력하도록 표시될 것이고, 당신은 webDir과 homePath가 당신의 웹 사이트와 해당 디렉터리의 전체 경로명을 포함하는 디렉터리로 적절하게 초기화되는지 확인할 수 있을 것이다.

프로그램에 따라 WebDir를 인스턴스화할 수 있다면 좋을 것이므로 다른 생성 메서드를 추가해보자.

 아래 메서드를 추가하여 WebDir onPath: 'path to your web site' 의 결과를 검사함으로써 시도해보라.

```
WebDir class>>onPath: homePath
    ^ self onPath: homePath home: homePath

WebDir class>>onPath: path home: homePath
    ^ self new setDir: (path asFileReference) home: homePath
```

HTML 파일의 패턴 매칭

지금까진 매우 좋다. 이제 regex를 이용해 이 웹 사이트가 어떤 HTML 파일을 포함하는지 알아내도록 하겠다.

AbstractFileReference 클래스를 살펴보면, fileNames 메서드가 디렉터리 내 모든 파일을 열거할 것임을 알 수 있다. 우리는 파일 확장자가 .html 인 파일만 선택하길 원한다. 우리에게 필요한 정규 표현식은 '.*\.html'이다. 첫 번째 점은 어떤 문자든 매치시킬 것이다.

```
'x' matchesRegex: '.' → true
' ' matchesRegex: '.' → true
Character cr asString matchesRegex: '.' → true
```

* (이것을 개발한 Stephen Kleene의 이름을 본따 "클리니 스타(Kleene star)"라고 알려짐)은 정규 표현식 연산자로서, (0을 포함해) 앞에 몇 개의 정규 표현식이든 나타날 수 있다는 의미다.

```
\textit{matchesRegex: 'x\(\ast{}\)' → true
'x' matchesRegex: 'x\(\ast{}\)' → true
'xx' matchesRegex: 'x\(\ast{}\)' → true
'y' matchesRegex: 'x\(\ast{}\)' → false
```

점(dot)은 regex에서 특수 문자기 때문에 말 그대로 점을 일치시키기 위해서는 escape 시켜야 한다.

```
'.' matchesRegex: '.' → true
'x' matchesRegex: '.' → true
'.' matchesRegex: '\textbackslash .' → true
'x' matchesRegex: '\textbackslash .' → false
```

이제 HTML 파일이 예상한 대로 작동하는지 알아보기 위해 정규 표현식을 확인해보자.

```
'index.html' matchesRegex: '.\(\ast{}\)\textbackslash .html' → true
'foo.html' matchesRegex: '.\(\ast{}\)\textbackslash .html' → true
'style.css' matchesRegex: '.\(\ast{}\)\textbackslash .html' → false
'index.htm' matchesRegex: '.\(\ast{}\)\textbackslash .html' → false
```

괜찮아 보인다. 그렇다면 애플리케이션에서 시도해보자.

 아래 메서드를 **WebDir** 에 추가하고 자신의 테스트용 웹 사이트에서 시도해보라.

```
WebDir>>htmlFiles
    ^webDir fileNames select: [ :each | each matchesRegex: '.*\.html' ]
```

htmlFiles 를 WebDir 인스턴스로 보내고 print it하면, 아래와 같은 결과를 볼 수 있을 것이다.

```
(WebDir onPath: '{\dots}') htmlFiles → \#('index.html' {\dots})
```

정규 표현식의 캐시 저장

이제 matchesRegex: 를 살펴보면 이것을 전송할 때마다 RxParser의 새로운(fresh) 인스턴스를 생성하는 것은 String의 확장 메서드라는 것을 발견할 것이다. Ad hoc 쿼리에는 괜찮지만 같은 정규 표현식을 웹 사이트 내 모든 파일로 적용한다면 RxParser의 인스턴스 하나만 생성하여 재사용하는 편이 현명하겠다. 이 방법을 시도해보자.

 새로운 인스턴스 변수 htmlRegex를 WebDir로 추가하고, 우리의 정규 표현식 문자열에 asRegex를 전송하여 초기화하라. 아래와 같이 매번 같은 정규 표현식을 사용하도록 WebDir≫htmlFiles를 수정하라.

```
WebDir>>initialize
    htmlRegex := '.*\.html' asRegex

WebDir>>htmlFiles
    ^webDir fileNames select: [ :each | htmlRegex matches: each ]
```

이제 동일한 regex를 여러 번 재사용한다는 점을 제외하면 HTML 파일의 열거는 이전처럼 작동할 것이다.

웹 페이지 접근하기

각 웹 페이지의 세부 내용을 접근하는 일은 구분된 클래스의 책임이므로, 이를 정의하여 WebDir 클래스가 인스턴스를 생성하도록 하자.

 HTML 파일을 식별하기 위한 인스턴스 변수 path, 웹 사이트의 루트 디렉터리를 식별하기 위한 인스턴스 변수 homePath가 있는 클래스 WebPage를 정의하라. (이는 웹 사이트의 루트로부터 웹 사이트가 포함하는 파일로 올바르게 링크를 생성 시 필요할 것이다.) 클래스 측에 생성 메서드와 인스턴스 측에 초기화 메서드를 정의하라.

```
WebPage>>initializePath: filePath homePath: dirPath
    path := filePath.
    homePath := dirPath

WebPage class>>on: filePath forHome: homePath
    ^ self new initializePath: filePath homePath: homePath
```

WebDir 인스턴스는 그것이 포함하는 모든 웹 페이지의 목록을 리턴할 수 있어야 한다.

 아래 메서드를 WebDir에 추가하고, 올바로 작동하는지 확인하기 위해 리턴값을 검사하라.

```
WebDir>>webPages
    ^ self htmlFiles collect:
        [ :each | WebPage
            on: webDir fullName, '/', each
            forHome: homePath ]
```

아래와 같은 결과가 보일 것이다:

```
(WebDir onPath: '{\dots}') webPages → an Array(a WebPage a WebPage {\dots})
```

문자열 치환(substitution)

뜻을 정확히 알 수 없으므로 정규 표현식을 이용해 각 웹 페이지에 대한 실제 파일명을 얻도록 하자. 이를 위해서는 마지막 디렉터리까지의 경로명에서 모든 문자를 제거하고자 한다. Unix 파일 시스템에서 디렉터리는 슬래시(/)로 끝나므로 파일 경로에서 마지막 슬래시까지 모두 제거할 필요가 있다.

String 확장 메서드 copyWithRegex:matchesReplacedWith: 가 우리가 원하는 일을 수행한다:

```
'hello' copyWithRegex: '[elo]+' matchesReplacedWith: 'i' → 'hi'
```

이 예제에서 정규 표현식 [elo]은 e, l, o 중 어떤 문자든 매치시킨다. 연산자 + 는 클리니 스타와 같지만 그 앞에 오는 정규 표현식의 하나 또는 그 이상의 인스턴스와 정확히 매치한다. 여기서는 전체 하위문자열 'ello'에 매치하여 i 문자로 된 새 문자열에서 다시 보일 것이다.

 아래 메서드를 추가하고, 예상대로 작동하는지 검사하라.

```
WebPage>>fileName
    ^ path copyWithRegex: '.*/' matchesReplacedWith: ''
```

이제 자신의 테스트용 웹 사이트에 아래와 같은 내용이 보일 것이다:

```
(WebDir onPath: '...') webPages collect: [:each | each fileName ]
    → #('index.html' ...)
```

정규 표현식 매치 대상의 추출

다음 임무는 각 HTML 페이지의 제목을 추출하는 일이다.

먼저 각 페이지의 내용에 접근할 방법이 필요한데, 다음과 같은 방법이 간단하겠다.

아래 메서드를 추가하여 시도해보라.

```
WebPage>>contents
    ^ (FileStream oldFileOrNoneNamed: path) contents
```

사실상 자신의 웹 페이지가 ASCII가 아닌 문자를 포함한다면 문제가 있겠으나, 그런 경우 아래의 코드로 시작하는 편이 나을 것이다.

```
WebPage>>contents
    ^ (FileStream oldFileOrNoneNamed: path)
        converter: Latin1TextConverter new;
        contents
```

그러면 아래와 같은 내용을 볼 수 있을 것이다.

```
(WebDir onPath: '{\dots}') webPages first contents → '<head>
<title>Home Page</title>
{\dots}
'
```

이제 제목을 추출해보자. 이번 경우, 우리는 HTML 태그 <titles>와 </title> 사이에서 발생하는 텍스트를 보고 있다.

우리에게 필요한 것은 정규 표현식의 매치 부분을 추출하는 일이다. 정규 표현식의 하위표현식은 괄호로 구분된다. 정규 표현식 ([^aeiou]+)([aeiou]+)를 고려해보자. 이는 두 개의 하위표현식으로 구성되는데, 첫 번째는 하나 또는 그 이상의 비모음(non-vowels)의 시퀀스에 일치하고, 두 번째는 하나 또는 그 이상의 모음에 일치할 것이다 (괄호로 된 문자 집합의 시작에 위치한 연산자 ^는 집합을 부정한다.[3]).

이제 'pharo' 문자열의 접두사를 매치시키고 하위매치를 추출하도록 하겠다:

```
re := '([^aeiou]+)([aeiou]+)' asRegex.
re matchesPrefix: 'pharo' → true
re subexpression: 1 → 'pha'
re subexpression: 2 → 'ph'
re subexpression: 3 → 'a'
```

문자열에 대한 정규 표현식을 성공적으로 매치시키고 나면 전체 매치를 추출하기 위해 subexpression: 1 메시지를 언제든지 전송할 수 있다. subexpression: n 메시지도 전송할 수 있는데 여기서 n은 정규 표현식 내 하위표현식의 개수이다. 위의 정규 표현식에는 2와 3으로 번호 매겨진 두 개의 하위표현식이 있다.

같은 수법을 이용해 HTML 파일로부터 제목을 추출해보겠다.

아래 메서드를 정의하라.

[3] NB: Pharo에서 탈자 기호는 리턴 키워드, 즉 ^에 해당하기도 한다. 혼동을 막기 위해, 문자 집합을 부정 시 정규 표현식 내에서 탈자 기호를 사용할 때 ^를 쓰겠지만 사실상 두 가지가 동일함을 잊어선 안 된다.

```
WebPage>>title
    | re |
    re := '[\w\W]*<title>(.*)</title>' asRegexIgnoringCase.
    ^ (re matchesPrefix: self contents)
        ifTrue: [ re subexpression: 2 ]
        ifFalse: [ '(', self fileName, ' -- untitled)' ]
```

HTML은 태그가 대문자인지 소문자인지 신경 쓰지 않으므로 우리는 asRegexIgnoringCase를 인스턴스화함으로써 정규 표현식이 대 · 소문자에 민감하지 않도록 만들어야 한다.

이제 제목 추출기를 테스트할 수 있고, 아래와 같은 내용을 볼 것이다.

```
(WebDir onPath: '{\dots}') webPages first title → 'Home page'
```

추가 문자열 치환

사이트 맵을 생성하기 위해서는 각 웹 페이지로의 링크를 생성할 필요가 있겠다. 문서 제목을 링크의 이름으로 사용할 수 있다. 웹 사이트의 루트에서 웹 페이지에 대한 올바른 경로를 생성하면 된다. 다행히 손쉬운 작업인데, 웹 페이지에 대한 전체 경로에서 웹 사이트의 루트 디렉터리의 전체 경로를 제하기란 단순하기 때문이다.

한 가지만 주의하면 되겠다. HomePath 변수는 /로 끝나지 않기 때문에 하나를 추가해야만 상대 경로가 /로 시작하지 않을 것이다. 아래 두 결과의 차이를 주목하라.

```
'/home/testweb/index.html' copyWithRegex: '/home/testweb' matchesReplacedWith: }
→ '/index.html'

'/home/testweb/index.html' copyWithRegex: '/home/testweb/' matchesReplacedWith: \textit{
→ 'index.html'
```

첫 번째 결과는 절대 경로, 어쩌면 우리가 원하는 결과를 제공할 것이다.

아래 메서드를 정의하라.

```
WebPage>>relativePath
    ^ path
        copyWithRegex: homePath , '/'
        matchesReplacedWith: ''

WebPage>>link
    ^ '<a href="', self relativePath, '">', self title, '</a>'
```

이제 아래와 같은 결과를 확인할 것이다.

```
(WebDir onPath: '{\dots}') webPages first link → '<a href="index.html">Home Page</a>'
```

사이트 맵 생성하기

사실상 사이트 맵을 생성하는 데에 필요한 정규 표현식은 끝이 난 셈이다. 애플리케이션을 완성하기 위해서는 몇 가지 메서드만 필요하다.

사이트 맵 생성을 보고 싶다면 아래 메서드를 추가하기만 하면 된다.

우리 웹 사이트에 하위디렉터리가 있다면 하위디렉터리로 접근하는 방법이 필요하다.

```
WebDir>>webDirs
    ^ webDir directoryNames
        collect: [ :each | WebDir onPath: webDir pathName , '/' , each home: homePath ]
```

웹 디렉터리의 각 웹 페이지에 대한 링크를 포함하는 HTML bullet 리스트를 생성할 필요가 있다. 하위디렉터리는 자체 bullet 리스트에서 들여쓰기가 되어 있어야 한다.

```
WebDir>>printTocOn: aStream
    self htmlFiles
        ifNotEmpty: [
            aStream nextPutAll: '<ul>'; cr.
            self webPages
                do: [:each | aStream nextPutAll: '<li>';
                    nextPutAll: each link;
                    nextPutAll: '</li>'; cr].
            self webDirs
                do: [:each | each printTocOn: aStream].
            aStream nextPutAll: '</ul>'; cr]
```

루트 웹 디렉터리에 "toc.html"이라는 파일을 생성하고 그곳에 사이트 맵을 덤프(dump)한다.

```
WebDir>>tocFileName
    ^ 'toc.html'

WebDir>>makeToc
    | tocStream |
    tocStream := (webDir / self tocFileName) writeStream.
    self printTocOn: tocStream.
    tocStream close.
```

이제 임시 웹 디렉터리에 대한 내용의 테이블을 생성할 수 있다!

```
WebDir selectHome makeToc
```

정규 표현식 구문

이제 Regex 패키지가 지원하는 정규 표현식의 구문을 자세히 살펴볼 것이다.

가장 단순한 정규 표현식은 단일 문자이다. 이는 해당 문자에 정확히 매치한다. 문자의 시퀀스는 문자의 시퀀스와 정확히 동일한 문자열을 매치한다:

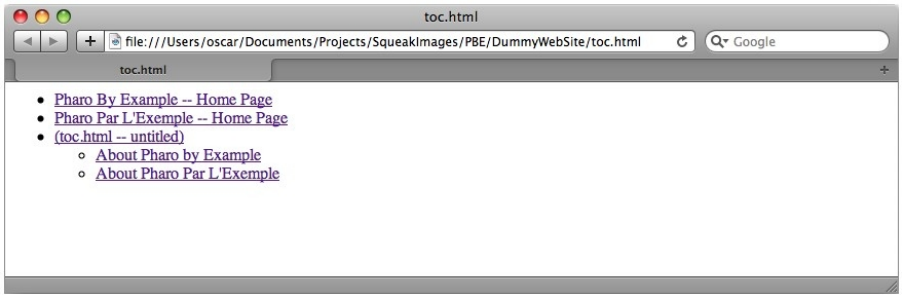


그림 6.1: 작은 사이트 맵.

```
'a' matchesRegex: 'a' → true
'foobar' matchesRegex: 'foobar' → true
'blorple' matchesRegex: 'foobar' → false
```

연산자는 좀 더 복잡한 정규 표현식을 생성하도록 정규 표현식에 적용된다. 연산자로서 순서화(sequencing; 표현식을 연이어 열거하는)는 "보이지 않는(invisible)" 경우가 보통이다.

클리니 스타(∗)와 + 연산자는 이미 살펴보았다. 정규 표현식 다음에 붙는 별표(∗)는 원본 표현식의 매치 중 어떤 숫자(0을 포함)에든 매치한다. 예를 들자면 다음과 같다.

```
'ab' matchesRegex: 'a\(\ast{}\)b' → true
'aaaaab' matchesRegex: 'a\(\ast{}\)b' → true
'b' matchesRegex: 'a\(\ast{}\)b' → true
'aac' matchesRegex: 'a\(\ast{}\)b' → false "b does not match"
```

클리니 스타는 시퀀싱보다 우선순위가 높다. 별표는 그보다 앞설 수 있는 가장 짧은 하위표현식에 적용된다. 예를 들어, ab∗ 는 "ab의 0번 또는 그 이상의 발생"이 아니라 a 다음에 b의 0번 또는 그 이상의 발생을 의미한다.

```
'abbb' matchesRegex: 'ab\(\ast{}\)' → true
'abab' matchesRegex: 'ab\(\ast{}\)' → false
```

"ab의 0번 또는 그 이상의 발생" 과 일치하는 정규 표현식을 얻기 위해서는 ab를 괄호로 감싸야 한다.

```
'abab' matchesRegex: '(ab)\(\ast{}\)' → true
'abcab' matchesRegex: '(ab)\(\ast{}\)' → false "c spoils the fun"
```

∗와 비슷하면서 유용한 연산자로 두 가지, +와 ?가 있다. +는 그것이 수정하는 정규 표현식의 하나 또는 그 이상의 인스턴스를 매치하고, ?는 0개 또는 하나의 인스턴스를 매치할 것이다.

```
'ac' matchesRegex: 'ab\(\ast{}\)c' → true
'ac' matchesRegex: 'ab+c' → false "need at least one b"
'abbc' matchesRegex: 'ab+c' → true
'abbc' matchesRegex: 'ab?c' → false "too many b's"
```

살펴보았듯 *, +, ?, (, 와) 문자들은 정규 표현식 내에서 특별한 의미를 가진다. 그 중 어떤 것이라도 문자 그대로 매치하기 위해서는 그 앞에 백슬래시 \를 붙임으로써 escape시켜야 한다. 따라서 백슬래시도 특수 문자에 해당하며, 리터럴 매치에 대해 escape할 필요가 있다. 우리가 살펴볼 모든 특수 문자에도 마찬가지로 적용된다.

```
'ab\(\ast{}\)' matchesRegex: 'ab\(\ast{}\)' → false "star in the right string is special"
'ab\(\ast{}\)' matchesRegex: 'ab\textbackslash \(\ast{}\)' → true
'a\textbackslash c' matchesRegex: 'a\textbackslash \textbackslash c' → true
```

마지막 연산자 |는 두 개의 하위표현식 사이에 선택을 나타낸다. 이는 두 하위표현식 중 하나가 문자열에 매치할 경우 문자열에 매치한다. 해당 연산자의 우선순위는 가장 낮으며, 순서화(sequencing)보다도 낮다. 가령, ab*|ba* 는 "a 다음에 b의 어떤 개수든 따라올 수 있거나, b 다음에 a의 어떤 개수든 따라올 수 있음"을 의미한다.

```
'abb' matchesRegex: 'ab\(\ast{}\)|ba\(\ast{}\)' → true
'baa' matchesRegex: 'ab\(\ast{}\)|ba\(\ast{}\)' → true
'baab' matchesRegex: 'ab\(\ast{}\)|ba\(\ast{}\)' → false
```

조금 더 복잡한 예로, 표현식 c(a|d)+r 를 들 수 있는데, 이는 Lips-style car, cdr, caar, cadr, … 함수 중 어떤 이름이든 매치한다.

```
'car' matchesRegex: 'c(a|d)+r' → true
'cdr' matchesRegex: 'c(a|d)+r' → true
'cadr' matchesRegex: 'c(a|d)+r' → true
```

빈 문자열을 매치하는 표현식을 작성하는 것도 가능한데, 가령 a | 표현식은 빈 문자열을 매치한다. 하지만 그러한 표현식에 , +, 또는 ?를 적용시키는 것은 오류이며, (a |) 는 무효하다.

지금까지 정규 표현식의 가장 작은 구성요소로서 문자만 사용해왔다. 그 외에 다른 흥미로운 구성요소들도 있다. 문자 집합은 사각 괄호로 된 문자열이다. 이는 괄호 사이에 나타날 경우 어떤 단일 문자든 매치한다. 예를 들자면, [01]는 0 또는 1을 매치한다:

```
'0' matchesRegex: '[01]' → true
'3' matchesRegex: '[01]' → false
'11' matchesRegex: '[01]' → false "a set matches only one character"
```

플러스 연산자를 이용해 다음과 같은 이진수 인식기(recognizer)를 빌드할 수도 있겠다:

```
'10010100' matchesRegex: '[01]+' → true
'10001210' matchesRegex: '[01]+' → false
```

여는 괄호 다음에 오는 첫 번째 문자가 ^라면 집합은 역전(inverted)되어, 괄호 안에 없는 어떤 단일 문자든 매치한다.

```
'0' matchesRegex: '[\textasciicircum{}01]' → false
'3' matchesRegex: '[\textasciicircum{}01]' → true
```

편의상 집합은 범위를 포함할 수 있는데, 범위란 하이픈(-)으로 구분된 문자 쌍이다. 이는 가운데 위치한 모든 문자를 열거하는 것과 같아서 '[0-9]'는 '[0123456789]'와 같다. 집합 내

특수 문자는 ^, -, 집합을 닫는]가 있다. 아래는 집합 내 특수문자를 매치하는 방법을 예로 든 것이다.

```
'\textasciicircum{}' matchesRegex: '[01\textasciicircum{}]' → true "put the caret anywhere ex
    cept the start"
'-' matchesRegex: '[01-]' → true "put the hyphen at the end"
']' matchesRegex: '[]01]' → true "put the closing bracket at the start"
```

따라서 빈 집합과 전체 집합은 명시할 수 없다.

표 6.1: 정규 표현식의 간단한 구문

구문	구문이 표현하는 대상
a	문자 a의 리터럴 매치
.	어떤 char든 매치
(…)	그룹 하위표현식
\	뒤에 오는 특수 문자를 escape
*	클리니 스타 – 이전 정규 표현식을 0번 또는 그 이상 매치
?	이전 정규 표현식을 0번 또는 한 번 매치
¦	좌측이나 우측 정규 표현식 선택을 매치
[abcd]	문자 abcd의 선택을 매치
[^abcd]	문자의 부정된 선택을 매치
[0-9]	0부터 9까지 문자 범위를 매치
\w	영숫자(alphanumeric)를 매치
\W	영숫자가 아닌 대상을 매치
\d	digit를 매치
\D	digit가 아닌 대상을 매치
\s	공백을 매치
\S	공백이 아닌 대상을 매치

Character 클래스

정규 표현식은 아래와 같은 역다옴표 escape를 포함시켜 잘 사용되는 문자의 클래스를 참조할 수 있다: \w는 digit를 매치하고, \s는 공백을 매치한다. 이를 대문자로 변형한 \W, \D, \S는 complementary(채움) 문자를 매치한다 (영수자가 아닌, digit가 아닌, 공백이 아닌 대상을 매치). 지금까지 살펴본 구문의 요약본은 표 6.1에서 볼 수 있겠다.

서론에서 언급하였듯 정규 표현식은 특히 사용자 입력을 검증하는 데에 유용하며, 그러한 정규 표현식을 정의하는 데에는 특히 character 클래스가 유용한 것으로 밝혀졌다. 예를 들어, 음수가 아닌 숫자는 정규 표현식 d+ 를 이용해 매치할 수 있다.

```
'42' matchesRegex: '\textbackslash d+' → true
'-1' matchesRegex: '\textbackslash d+' → false
```

오히려 0이 아닌 숫자는 0으로 시작해선 안 됨을 명시하는 편이 낫겠다.

```
'0' matchesRegex: '0|([1-9]\textbackslash d\(\ast{}\))' → true
'1' matchesRegex: '0|([1-9]\textbackslash d\(\ast{}\))' → true
'42' matchesRegex: '0|([1-9]\textbackslash d\(\ast{}\))' → true
'099' matchesRegex: '0|([1-9]\textbackslash d\(\ast{}\))' → false "leading 0"
```

음수와 양수 또한 검사할 수 있겠다.

```
'0' matchesRegex: '(0|((\textbackslash +|-)?[1-9]\textbackslash d\(\ast{}\)))' → true
'-1' matchesRegex: '(0|((\textbackslash +|-)?[1-9]\textbackslash d\(\ast{}\)))' → true
'42' matchesRegex: '(0|((\textbackslash +|-)?[1-9]\textbackslash d\(\ast{}\)))' → true
'+99' matchesRegex: '(0|((\textbackslash +|-)?[1-9]\textbackslash d\(\ast{}\)))' → true
'-0' matchesRegex: '(0|((\textbackslash +|-)?[1-9]\textbackslash d\(\ast{}\)))' → fa
    lse     "negative zero"
'01' matchesRegex: '(0|((\textbackslash +|-)?[1-9]\textbackslash d\(\ast{}\)))' → fa
    lse "leading zero"
```

부동 소수점 수는 점 다음에 적어도 하나의 숫자를 필요로 한다.

```
'0' matchesRegex: '(0|((\textbackslash +|-)?[1-9]\textbackslash d\(\ast{}\)))(\textbacksl
    ash .\textbackslash d+)?' → true
'0.9' matchesRegex: '(0|((\textbackslash +|-)?[1-9]\textbackslash d\(\ast{}\)))(\textback
    slash .\textbackslash d+)?' → true
'3.14' matchesRegex: '(0|((\textbackslash +|-)?[1-9]\textbackslash d\(\ast{}\)))(\textbac
    kslash .\textbackslash d+)?' → true
'-42' matchesRegex: '(0|((\textbackslash +|-)?[1-9]\textbackslash d\(\ast{}\)))(\textback
    slash .\textbackslash d+)?' → true
'2.' matchesRegex: '(0|((\textbackslash +|-)?[1-9]\textbackslash d\(\ast{}\)))(\textbacks
    lash .\textbackslash d+)?' → false "need digits after ."
```

999 혹은 999.999 혹은 -999.999e+21 과 같이 무엇이든 해당하는 일반 숫자 형식에 대한 인식기를 간단한 예로 들자.

표 6.2: 정규 표현식 character 클래스

구문	구문이 표현하는 대상
[:alnum:]	어떤 영숫자든 가능
[:alpha:]	어떤 영문자든 가능
[:cntrl:]	어떤 제어 문자든 가능 (ascii 코드 <32)
[:digit:]	어떤 10진 숫자든 가능
[:graph:]	어떤 그래픽 (graphical) 문자든 가능 (ascii 코드≥32)
[:lower:]	어떤 소문자든 가능
[:print:]	어떤 인쇄 가능 문자든 가능 (여기서는 [:graph:]와 동일)
[:punct:]	어떤 구두 문자든 가능
[:space:]	어떤 공백 문자든 가능
[:upper:]	어떤 대문자든 가능
[:xdigit:]	어떤 16진 문자든 가능

이러한 요소들은 character 클래스의 구성요소여서 유효한 정규 표현식을 형성하기 위해서는 사각 괄호 안에 포함시켜야 함을 주목하라. 예를 들어, 비어 있지 않은 digit 문자열은 [[:digit:]]+ 로 표현할 수 있겠다. 위의 프리미티브 표현식과 연산자는 정규 표현식의 많은 구현에 공통된다.

```
'42' matchesRegex: '\href{http://trans.onionmixer.net/mediawiki/index.php?title=Digit:}{:d
    igit:}+' → true
```

특수 문자 클래스

다음 프리미티브 표현식은 스몰토크 구현에 유일하다. 콜론 사이의 문자 시퀀스는 단항 선택자로서 취급되는데, 이는 문자들이 이해해야 한다. 문자는 그 선택자를 이용해 메시지로 true를 응답할 경우 그러한 표현식에 매치한다. 이는 character 클래스를 좀 더 쉽게 읽을 수 있고 효율적인 방식으로 명시하도록 허용한다. 예를 들어, [0-9]는 :isDigit: 와 같지만 후자가 더 효율적이다. Character 집합과 동일하게 character 클래스도 부정이 가능하여, ^isDigit: 는 isDigit에 false로 답하는 문자를 매치시키므로 [^0-9]와 같다.

지금까지 비어 있지 않은 digit의 문자열을 매치하는 정규 표현식을 작성하는 방법으로 [0-9]+, d+, []+, [[:digit:]]+, :isDigit:+. 를 살펴보았다.

```
'42' matchesRegex: '[0-9]+' → true
'42' matchesRegex: '\textbackslash d+' → true
'42' matchesRegex: '[\textbackslash d]+' → true
'42' matchesRegex: '\href{http://trans.onionmixer.net/mediawiki/index.php?title=Digit:}{:d
    igit:}+' → true
'42' matchesRegex: ':isDigit:+' → true
```

경계 매치하기

특수 프리미티브 표현식의 마지막 그룹은 표 6.3에 표시되어 있는데, 이는 문자열의 경계를 매치하는 데에 사용된다.

표 6.3: 문자열 경계를 매치하기 위한 프리미티브

구문	구문이 표현하는 대상
^	행의 시작에 빈 문자열을 매치
$	행의 끝에 빈 문자열을 매치
\b	단어 경계에서 빈 문자열을 매치
\B	단어 경계가 아닌 곳에서 빈 문자열을 매치
\<	단어 시작에 빈 문자열을 매치
\>	단어 끝에 빈 문자열을 매치

```
'hello world' matchesRegex: '.\(\ast{}\)\textbackslash bw.\(\ast{}\)' → true "word boundar
    y before w"
'hello world' matchesRegex: '.\(\ast{}\)\textbackslash bo.\(\ast{}\)' → false "no boundary
     before o"
```

정규 표현식 API

지금까지는 주로 정규 표현식의 구문에 중점을 두었다. 이제 문자열과 정규 표현식이 이해하는 여러 메시지들을 자세히 살펴보겠다.

접두사의 매칭과 대 · 소문자 무시

여태껏 보인 예제들 대부분은 String 확장자 메서드 matchesRegex: 를 사용해왔다.

문자열 또한 prefixMatchesRegex:, matchesRegexIgnoringCase:, prefixMatchesRegex-IgnoringCase: 를 이해한다.

PrefixMatchesRegex: 메시지는 matchesRegex 와 같지만 전체 수신자가 인자로서 전달된 정규 표현식을 매치할 것으로 기대하지 않고 그 접두사만 매치한다는 점에서 다르다.

```
'abacus' matchesRegex: '(a|b)+' → false
'abacus' prefixMatchesRegex: '(a|b)+' → true
'ABBA' matchesRegexIgnoringCase: '(a|b)+' → true
'Abacus' matchesRegexIgnoringCase: '(a|b)+' → false
'Abacus' prefixMatchesRegexIgnoringCase: '(a|b)+' → true
```

열거 인터페이스

몇몇 애플리케이션은 문자열 내 특정 정규 표현식의 모든 매치로 접근해야 할 필요가 있다. 매치는 친숙한 Collection과 같은 열거형 프로토콜을 본따 모델화된 프로토콜을 이용해 접근이 가능하다.

regex:matchesDo: 는 수신자 문자열 내 정규 표현식의 매치마다 1인자 aBlock을 평가한다.

```
list := OrderedCollection new.
'Jack meet Jill' regex: '\textbackslash w+' matchesDo: [:word | list add: word].
list → an OrderedCollection('Jack' 'meet' 'Jill')
```

regex:matchesCollect: 는 수신자 문자열 내 정규 표현식의 매치마다 1인자 aBlock을 평가한다. 이후 결과를 수집하여 SequenceableCollection으로서 응답한다.

```
''Jack meet Jill' regex: '\w+' matchesCollect: [:word | word size] →
    an OrderedCollection(4 4 4)
```

allRegexMatches: 는 정규 표현식에 대한 모든 매치의 (수신자 문자열의 하위문자열) 컬렉션을 리턴한다.

```
'Jack and Jill went up the hill' allRegexMatches: '\textbackslash w+' →
an OrderedCollection('Jack' 'and' 'Jill' 'went' 'up' 'the' 'hill')
```

대체와 번역

copyWithRegex:matchesReplacedWith:를 이용해 정규 표현식의 모든 매치를 특정 문자열로 대체하는 것이 가능하다.

```
'Krazy hates Ignatz' copyWithRegex: '\<[[:lower:]]+\>' matchesReplacedWith: 'loves'
    → 'Krazy loves Ignatz'
```

좀 더 일반적인 대체는 매치의 번역이다. 이 메시지는 블록을 평가하고 이를 수신자 문자열 내 정규 표현식의 매치마다 전달하며, 각 매치 대신 블록 결과를 이은 수신자의 복사본을 응답한다.

```
'Krazy loves Ignatz' copyWithRegex: '\b[a-z]+\b' matchesTranslatedUsing: [:each | each asUpper
    case]
    → 'Krazy LOVES Ignatz'
```

열거 및 대체 프로토콜의 모든 메시지들은 대 · 소문자에 민감한 매치를 수행한다. 대 · 소문자에 둔감한 버전은 String 프로토콜의 일부로 제공되지 않는다. 대신 아래 질문에 제시된 저수준 매칭 인터페이스를 이용해 접근이 가능하다.

저수준 인터페이스

matchesRegex: 메시지를 문자열로 전송하면 다음과 같은 일이 발생한다.

1. RxParser의 새로운 인스턴스가 생성되고, 정규 표현식 문자열이 그곳으로 전달되어 표현식의 구문 트리가 생성된다.
2. 구문 트리는 RxMatcher의 인스턴스에 initialization 매개변수로서 전달된다. 인스턴스는 트리가 설명한 정규 표현식에 대한 인식기로서 작용하게 될 데이터 구조를 준비한다.
3. 원본 문자열이 matcher로 전달되고, matcher는 매치를 확인한다.

Matcher

String에 정의된 메시지들 중 하나를 이용해 다수의 문자열을 동일한 정규 표현식에 대해 반복적으로 매치할 경우, 매치마다 정규 표현식 문자열이 파싱되고 새로운 matcher가 생성된다. 이러한 오버헤드는 정규 표현식에 대한 matcher를 빌드한 후 matcher를 반복하여 재사용함으로써 피할 수 있다. 예를 들어, 클래스 또는 인스턴스 초기화 단계에서 matcher를 생성하고, 이를 추후 사용을 위해 변수에 보관하라. 다음 방법들 중 하나를 이용해 matcher를 생성할 수 있겠다.

- 문자열로 asRegex 또는 asRegexIgnoringCase를 전송할 수 있다.
- 그 클래스 메서드들, forString: 또는 forString:ignoreCase: 중 하나를 이용해 RxMatcher를 직접 인스턴스화할 수 있다 (위의 편리한 메서드가 하는 일을 할 것이다).

문자열에서 발견되는 모든 매치를 수집하도록 matchesIn: 을 전송한다.

```
octal := '8r[0-9A-F]+' asRegex.
octal matchesIn: '8r52 = 16r2A' → an OrderedCollection('8r52')
hex := '16r[0-9A-F]+' asRegexIgnoringCase.
hex matchesIn: '8r52 = 16r2A' → an OrderedCollection('16r2A')
hex := RxMatcher forString: '16r[0-9A-Fa-f]+' ignoreCase: true.
hex matchesIn: '8r52 = 16r2A' → an OrderedCollection('16r2A')
```

매칭

matcher는 이러한 메시지들을 이해한다 (모두들 성공적인 매치나 검색을 나타내기 위해 true를 리턴하고, 그렇지 않은 경우 false를 답한다).

matches: aString – 전체 인자 문자열 (aString) 이 매치할 경우 true.

```
'\textbackslash w+' asRegex matches: 'Krazy' → true
```

matchesPrefix: aString – 인자 문자열의 일부 접두사가 매치할 경우 true (전체 문자열이 매치할 필요는 없다).

```
'\textbackslash w+' asRegex matchesPrefix: 'Ignatz hates Krazy' → true
```

search: aString – 하위문자열이 처음으로 매치하는 문자열을 검색한다. (첫 두 개의 문자열은 문자열 처음부터 매치를 시도함을 주목하라.) matcher를 이용해 위의 예제에서 a+ 를 검색할 경우 해당 메서드는 'baaa' 문자열이 주어지면서 성공을 답하겠지만 앞의 두 개는 실패할 것이다.

```
'\textbackslash b[a-z]+\textbackslash b' asRegex search: 'Ignatz hates Krazy' → true "find
    s 'hates'"
```

matcher는 마지막 매치 시도의 결과도 보관하고 보고(report)하기도 한다: lastResult는 Boolean을 답했다: 가장 최근에 시도한 매치의 결과. 어떤 매치도 시도되지 않은 경우 답은 명시되지 않는다.

```
number := '\textbackslash d+' asRegex.
number search: 'Ignatz throws 5 bricks'.
number lastResult → true
```

matchesStream:, matchesStreamPrefix:, searchStream: 는 위의 세 메시지와 같지만 스트림을 인자로서 취한다.

```
ignatz := ReadStream on: 'Ignatz throws bricks at Krazy'.
names := '\textbackslash <[A-Z][a-z]+\textbackslash >' asRegex.
names matchesStreamPrefix: ignatz → true
```

하위표현식 매치

매치 시도가 성공하고 나면 원본 문자열의 어떤 부분이 어떤 정규 표현식 부분에 매치하였는지 질의할 수 있다. 하위표현식은 정규 표현식에서 괄호로 표시되거나, 전체 표현식이 될 수도 있다. 정규 표현식이 컴파일되면 그 하위표현식은 1부터 시작하는 깊이 우선의 좌우(depth-first, left-to-right)로 할당된 색인이 된다.

예를 들어, 정규 표현식 ((\\d+)\\s*(\\w+)) 에는 그 자체를 포함해 4개의 하위표현식이 있다.

```
1: ((\textbackslash d+)\textbackslash s\(\ast{}\)(\textbackslash w+)) "the complete expre
    ssion"
2: (\textbackslash d+)\textbackslash s\(\ast{}\)(\textbackslash w+) "top parenthesized su
    ∈bexpression"
3: \textbackslash d+ "first leaf subexpression"
4: \textbackslash w+ "second leaf subexpression"
```

가장 높은 유효 색인은 1에 매칭 괄호 개수를 더한 값과 같다. 따라서 1은 괄호로 감싼 하위표현식이 없더라도 항상 유효한 색인이다).

매치가 성공하고 나면 matcher는 원본 문자열의 어느 부분이 어느 하위표현식에 매치하였는지를 보고할 수 있다. matcher는 다음과 같은 메시지들을 이해한다:

subexpressionCount는 전체 하위표현식 개수를 응답하고, 최고 값은 해당 matcher와 하위표현식 색인으로서 사용 가능하다. 해당 값은 초기화 직후에 이용 가능하며 절대 변경되지 않는다.

subexpression: 는 유효 색인을 그 인자로서 취하고, 성공적인 매치 시도 이후에만 전송이 가능하다. 메서드는 해당하는 하위표현식을 매치시켜야 하는 원본 문자열의 하위문자열을 응답한다.

subBeginning: 과 subEnd: 는 주어진 하위표현식 매치가 각각 시작되고 끝나는 인자 문자열 또는 스트림 내에서 위치를 응답한다.

```
items := '((\textbackslash d+)\textbackslash s\(\ast{}\)(\textbackslash w+))' asRegex.
items search: 'Ignatz throws 1 brick at Krazy'.
items subexpressionCount → 4
items subexpression: 1 → '1 brick' "complete expression"
items subexpression: 2 → '1 brick' "top subexpression"
items subexpression: 3 → '1' "first leaf subexpression"
items subexpression: 4 → 'brick' "second leaf subexpression"
items subBeginning: 3 → an OrderedCollection(14)
items subEnd: 3 → an OrderedCollection(15)
items subBeginning: 4 → an OrderedCollection(16)
items subEnd: 4 → an OrderedCollection(21)
```

좀 더 세부적인 예로, 일자를 3 요소 배열, 즉 연도, 월, 일의 문자열로 변환하기 위해 MMM DD, YYYY 날짜 포맷을 이용하는 아래 예제를 들어보자.

```
date := '(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)\s+(\d\d?)\s*,\s*19(\d\d)' asRegex.
result := (date matches: 'Aug 6, 1996')
    ifTrue: [{ (date subexpression: 4) .
        (date subexpression: 2) .
        (date subexpression: 3) } ]
    ifFalse: ['no match'].
result → #('96' 'Aug' '6')
```

열거 및 대체

이번 절 앞 부분에서 살펴본 String 열거 및 대체 프로토콜은 사실상 matcher에 의해 구현된다. RxMatcher는 문자열 내 매치를 반복하기 위해 matchesIn:, matchesIn:do:, matchesIn:collect:, copy:replacingMatchesWith:, 그리고 copy:translatingMatchesUsing: 과 같은 메서드들을 구현한다.

```
seuss := 'The cat in the hat is back'.
aWords := '\<([^aeiou]|[a])+\>' asRegex. "match words with 'a' in them"
aWords matchesIn: seuss
    → an OrderedCollection('cat' 'hat' 'back')
aWords matchesIn: seuss collect: [:each | each asUppercase ]
    → an OrderedCollection('CAT' 'HAT' 'BACK')
aWords copy: seuss replacingMatchesWith: 'grinch'
    → 'The grinch in the grinch is grinch'
aWords copy: seuss translatingMatchesUsing: [ :each | each asUppercase ]
    → 'The CAT in the HAT is BACK'
```

스트림 내에서 매치를 반복하는 matchesOnStream:, matchesOnStream:do:, matchesOnStream:collect:, copyStream:to:replacingMatchesWith:, copyStream:to:translatingMatchesUsing: 와 같은 메서드들도 존재한다.

```
in := ReadStream on: '12 drummers, 11 pipers, 10 lords, 9 ladies, etc.'.
out := WriteStream on: ''.
numMatch := '\<\d+\>' asRegex.
numMatch
    copyStream: in
    to: out
    translatingMatchesUsing: [:each | each asNumber asFloat asString ].
out close; contents → '12.0 drummers, 11.0 pipers, 10.0 lords, 9.0 ladies, etc.'
```

오류 처리

정규 표현식을 빌드하는 동안에 RxParser가 발생시키는 예외가 몇 가지 있다. 예외는 RegexError를 공통 부모로 갖는다. 이러한 예외를 포착하고 처리하기 위해 일반적인 스몰토크 예외 처리 메커니즘을 사용할 수 있다.

- 정규 표현식을 파싱하는 동안 구문 오류가 감지될 경우 RegexSyntaxError가 발생한다.
- matcher를 빌드하는 동안 오류가 감지될 경우 RegexCompilationError가 발생한다.
- 매치하는 동안 오류가 발생할 경우 (예를 들어, ':<selector>:' 구문을 이용해 올바르지 않은 선택자가 명시되거나, matcher의 내부 오류로 인해) RegexMatchingError 가 발생한다.

```
['+' asRegex] on: RegexError do: [:ex | ^ ex printString ]
    → 'RegexSyntaxError: nullable closure'
```

Vassili Bykov의 구현 노트

먼저 확인해야 할 것. 90% 는 String≫matchesRegex: 메서드만으로 패키지에 접근이 가능할 것이다. RxParser는 정규 표현식과 함께 문자의 스트림이나 문자열을 수락하고, 표현식에 해당하는 구문 트리를 생산한다. 트리는 Rxs∗ 클래스로 구성된다.

RxMatcher는 파서가 빌드한 정규 표현식의 구문 트리를 수락하고, 이를 Rxm∗ 클래스의 인스턴스로 만들어진 구조인 matcher로 컴파일한다. RxMatcher 인스턴스는 문자의 위치지정 가능한 스트림 또는 문자열이 본래 정규 표현식에 매치하는지 테스트하거나 표현식에 매치하는 하위문자열을 스트림이나 문자열에서 검색할 수 있다. 매치가 발견되면 matcher는 전체 표현식에 매치하거나 그 중에서 괄호로 표기된 하위표현식에 매치한 특정 문자열을 보고할 수 있다. 다른 모든 클래스들도 동일한 기능을 지원하며, RxParser, RxMatcher, 또는 둘 다에 의해 사용된다.

통고. matcher는 C 에서 Henry Spencer의 원본 정규 표현식 구현과 정신은 비슷하지만 디자인은 다르다. 효율성이 아니라 단순성에 중점을 두었다. 필자는 어떤 것도 최적화하거나 작성하지 않았다. matcher가 H. Spencer의 테스트 도구("test suite" 프로토콜 참조)에 몇 가지 테스트를 추가해 전달하므로 그다지 많은 버그는 없을 것으로 사료된다.

감사의 말. matcher의 첫 발표 이후 여러 스몰토크 동료들의 투입에 감사하게도 native Smalltalk 정규 표현식 match를 살려두기 위해 노력할만한 가치가 있음을 확신하게 되었다. 이것이 가능하도록 조언과 격려를 준 Felix Hack, Eliot Miranda, Robb Shecter, David N. Smith, Francis Wolinski, 그리고 아직 만나거나 이야기를 들은 바는 없지만 온전한 시간 낭비가 아니었음을 동의하는 이들에게 감사의 말을 전한다.

요약

정규 표현식은 문자열을 쉽게 조작하는 데 필수적인 툴이다. 본 장에서는 Pharo용 Regex 패키지를 제시하였다. 이번 장의 요점은 다음과 같다.

- 간단한 매칭을 위해서는 문자열로 matchesRegex: 만 전송하라.
- 성능이 중요하면 정규 표현식을 표현하는 문자열로 asRegex를 전송하고, 다수의 매치에 대해 결과 matcher를 재사용하라.
- 매치하는 정규 표현식의 하위표현식은 임의 깊이(arbitrary depth)로 쉽게 검색이 가능하다.
- 매치하는 정규 표현식은 매치된 문자열의 새 복사본에 하위표현식을 번역 또는 대체할 수 있다.
- 특정 정규 표현식의 모든 매치로 접근하도록 열거 인터페이스가 제공된다.
- 정규 표현식은 문자열 뿐만 아니라 스트림과도 작동한다.

제 II 부

Source Management

제 7 장

자신의 코드를 Monticello로 버전관리하기

***Oscar Nierstrasz** 와 공동 작성 (oscar.nierstrasz@acm.org)*

버전관리 시스템은 코드의 여러 버전을 보관하고 로그하도록 도와준다. 게다가 공통 소스 코드 저장소로 동시 접근을 관리하도록 도와주기도 한다. 또 문서 집합에 일어나는 모든 변경 사항을 추적하고, 여러 개발자들의 협력을 가능하게 한다. 소프트웨어 크기가 몇 개의 클래스 이상으로 증가하는 즉시 버전관리 시스템이 필요할 것이다.

여러 다른 버전관리 시스템이 이용 가능하다. CVS[1], Subversion[2], Git[3]가 아마도 가장 유명할 것이다. 원칙상 이들은 Pharo 소프트웨어 프로젝트의 개발을 관리하는 데에 사용할 수 있지만 그러한 실습은 버전관리 시스템을 Pharo 환경에서 분리시킬 것이다. 뿐만 아니라 CVS와 같은 툴은 일반 텍스트 파일만 버저닝하고 개별적인 패키지, 클래스 또는 메서드는 버저닝하지 않는다. 따라서 적절한 세분성 (granularity) 수준에서 변경 내용을 추적하는 능력이 우리에게 부족할 것이다. 만일 당신이 일반 텍스트 대신 클래스나 메서드를 보관한다는 사실을 버전관리 툴이 알고 있다면 개발 과정을 더 잘 지원할 수 있을 것이다.

프로젝트를 보관하는 저장소에는 여러 가지가 있다. SmalltalkHub[4]와 Squeaksource 3[5]은 두 가지 무료로 이용 가능한 대표적인 저장소다. 이들은 텍스트의 행 대신 클래스와 메서드가 변경의 단위인 Pharo를 위한 버전관리 시스템들이다. 이번 장에서는 SmalltalkHub 를 이용하겠지만 Squeaksource 3도 이와 동일하게 이용 가능하다. SmalltalkHub 는 SourceForge와 같고, Monticello는 CVS와 같다.

본 장을 통해 당신은 Monticello와 SmalltalkHub 를 이용해 자신의 소프트웨어를 관리하

[1] http://www.nongnu.org/cvs
[2] http://subversion.tigris.org
[3] http://git-scm.com/
[4] http://smalltalkhub.com/
[5] http://ss3.gemstone.com/

는 방법을 학습할 것이다. 앞의 여러 장에 걸쳐 Monticello는 간략하게 살펴보았으나[6], 이번 장에서는 Monticello를 세부적으로 살펴보고, 큰 애플리케이션을 버저닝하는 데에 유용한 추가 기능들을 몇 가지 설명하겠다.

기본 사용

패키지를 생성하고 변경 내용을 적용하는 기본 내용을 검토하는 것으로 시작해 업데이트와 변경 내용을 병합하는 방법을 살펴볼 것이다.

예제 실행하기 – 완전수(perfect numbers)

이번 장에서는 Monticello의 기능을 설명하기 위해 완전수[7]를 실행하는 작은 예제를 사용하겠다. 몇 가지 간단한 테스트를 정의함으로써 프로젝트를 시작하겠다.

 **Perfect** 패키지에 **PerfectTest** 라 불리는 **TestCase** 의 서브클래스를 정의하고, **running** 프로토콜 내 아래 테스트 메서드를 정의하라.

```
PerfectTest>>testPerfect
    self assert: 6 isPerfect.
    self assert: 7 isPerfect not.
    self assert: 28 isPerfect.
```

물론 정수에 대한 isPerfect 메서드를 구현하지 않았기 때문에 위의 테스트들은 실패할 것이다. 해당 코드를 수정하고 확장하는 동안 Monticello의 제어 하에 두고자 한다.

Monticello 시작하기

Monticello는 표준 Pharo 배포판에 포함된다. Monticello Browser는 World 메뉴에서 선택 가능하다. 그림 7.1을 참조하면, Monticello Browser가 두 개의 리스트 패인과 하나의 버튼 패인으로 구성됨을 볼 수 있다. 좌측 패인은 설치된 패키지를 열거하고 우측 패인은 알려진 저장소를 표시한다. 버튼 패인과 두 개의 리스트 패인의 메뉴를 통해 여러 연산을 실행할 수 있다.

패키지 생성하기

Monticello는 패키지의 버전들을 관리한다. 패키지는 근본적으로 클래스와 메서드의 명명된 집합이다. 사실 패키지는 하나의 객체, PackageInfo의 인스턴스로서, 자신에게 속한 메서드와 클래스를 식별하는 방법을 알고 있다.

우리의 PerfectText 클래스를 버저닝하고자 한다. 올바른 방법은 Perfect라고 불리는 패키지를 정의하는 것으로, 해당 패키지는 PerfectTest와 그 외의 관련된 클래스 및 메서드를 모두

[6] "A first application(첫 번째 애플리케이션)"과 "The Pharo programming environment(Pharo 프로그래밍 환경)"
[7] 완전수는 Euclid(유클리드 기하학에 나오는 그분 맞다)에 의해 발견되었다. 완전수는 자신의 진약수들의 합이 자신이 되는 양의 정수를 나타낸다. 6=1+2+3이 첫 번째 완전수가 된다.

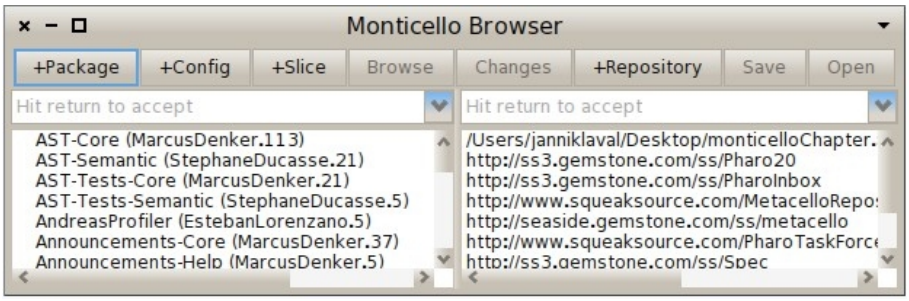


그림 7.1: Monticello Browser.

포함하는데, 이와 관련해서는 추후에 소개하겠다. 현재로선 그러한 패키지가 전혀 존재하지 않는다. 단지 Perfect(이름이 우연의 일치는 아니다)라고 불리는 범주를 갖고 있다. Monticello는 우리를 위해 범주를 패키지로 매핑할 것이므로 이는 완벽하다.

Monticello 브라우저에서 +Package 를 누르고 Perfect 를 입력하라.

짜잔! 이제 Perfect Monticello 패키지가 생성되었다.

Monticello 패키지는 클래스와 메서드 범주를 대상으로 한 중요한 명명 규칙을 많이 따른다. Perfect 라고 명명된 우리의 새 패키지는 아래를 포함한다.

- **Perfect** 범주 내 또는 이름이 **Perfect-** 로 시작되는 범주 내 모든 클래스. 현재로선 우리의 **PerfectTest** 클래스만 포함된다.
- **perfect** 또는 ∗**Perfect** 로 명명된 프로토콜이나, 이름이 ∗**perfect-** 또는 ∗**Perfect-** 로 시작되는 프로토콜에서 정의된 어떤 클래스든 (어떤 범주든) 그에 속하는 모든 메서드. 그러한 메서드는 *확장(extensions)*이라고 한다. 아직 우리에겐 이러한 메서드가 없지만 조만간 정의하게 될 것이다.
- **Perfect** 의 범주, 또는 이름이 **Perfect-** 로 시작되는 범주 내 어떤 클래스든 그에 속하는 모든 메서드로, ∗ 로 이름이 시작되는 프로토콜 내 클래스에 속하는 메서드는 제외된다 (*예:* 다른 패키지에 속하는). 이는 프로토콜 **running** 에 속하기 때문에 우리의 **testPerfect** 메서드를 포함한다.

변경 적용하기

그림 7.2 에서 Save 버튼은 비활성화(회색 처리)되었음을 주목하라.

우리의 Perfect 패키지를 저장하기 전에 먼저 어디에 이것을 저장하길 원하는지 명시할 필요가 있다. 저장소는 패키지 컨테이너로, 자신의 머신에 국부적이거나 원격적(네트워크에 걸쳐 접근)이다. 다양한 프로토콜을 이용해 Pharo 이미지와 저장소 간 연결을 구축할 수 있다. 후에 7.5절에서 살펴보겠지만 Monticello는 다양한 저장소 선택을 지원하지만 그 중에서 가장 공통적으로 사용되는 것은 HTTP인데, 그 이유는 SmalltalkHub에서 사용되기 때문이다.

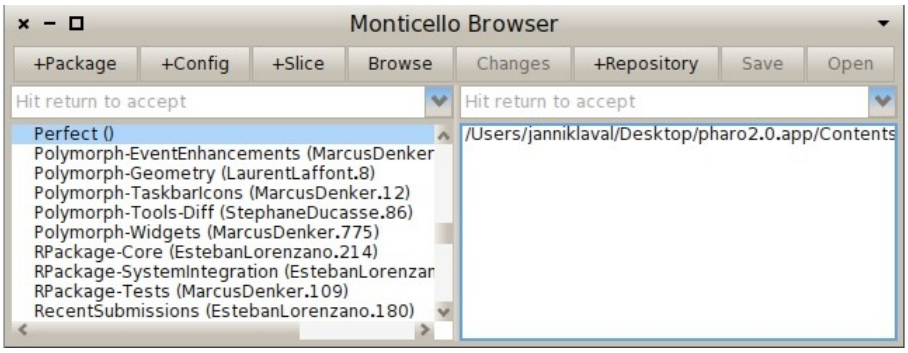


그림 7.2: Perfect 패키지 생성하기.

적어도 기본적으로 package-cache라고 불리는 저장소가 마련되어 있는데, Monticello 브라우저(그림 7.1 참고)의 우측에 열거된 저장소 리스트에서 가장 첫 엔트리에 표시될 것이다. package-cache는 Pharo 이미지가 위치한 로컬 디렉터리에 자동으로 생성된다. 이는 당신이 원격 저장소에서 다운로드한 모든 패키지의 복사본을 포함할 것이다. 패키지의 복사본을 원격 서버로 저장하면 이 또한 기본적으로 package-cache에 저장된다.

각 패키지는 그것을 저장할 수 있는 저장소를 안다. 선택된 패키지로 새 저장소를 추가하려면 +Repository 버튼을 누른다. 그러면 HTTP를 포함해 다양한 저장소 유형이 제공될 것이다. 이번 장의 나머지 부분에서는 Monticello의 기능을 탐구하는 데에 필요한 package-cache 저장소를 작업할 것이다.

package cache 로 명명된 디렉터리 저장소를 선택하고 Save 를 누른 후, 적절한 로그 메시지를 입력하고 변경 내용을 저장하기 위해 Accept 를 눌러라.

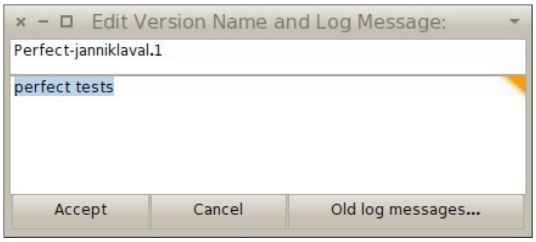


그림 7.3: 패키지의 버전을 저장 시에는 새로운 버전 이름과 커밋(commit) 메시지를 설정할 수 있다.

Perfect 패키지는 이제 package-cache에 저장되는데, 이는 Pharo 이미지와 동일한 디렉터리에 포함된 하나의 디렉터리에 지나지 않는다. 하지만 다른 유형의 저장소(예: HTTP, FTP, 다른 로컬 디렉터리)를 사용할 경우 당신의 패키지 복사본 또한 package-cache에 보관될 것임을 주목하라.

 자신이 가장 좋아하는 파일 브라우저 (예 : Windows Explorer, Finder 또는 XTerm) 를 이용해 파일 Perfect-XX.1.mcz가 패키지 캐시에서 생성되었음을 확인하라. XX는 당신의 이름이나 이니셜에 일치한다.

[8]

version은 저장소로 작성된 패키지의 변경 불가한 스냅샷이다. 각 버전은 저장소에서 그것을 식별하는 유일한 버전 번호를 갖는다. 하지만 이 숫자는 전역적으로 유일하진 않음을 주목해야 하는데, 다른 저장소에서 이와 같은 파일 식별자는 다른 스냅샷을 의미하는 수도 있다. 예를 들어, 다른 저장소에서 Perfect-onierstrasz.1.mcz 는 프로젝트의 deployed 버전의 final이 될 수도 있는 것이다! 버전을 저장소로 저장할 때는 그 다음으로 이용 가능한 번호가 자동으로 버전에 할당되지만 원한다면 번호를 수정할 수도 있다. 버전 branch들은 번호 매김 방식을 간섭하지 않음을 (CVS나 Subversion처럼) 명심하라. 나중에 살펴보겠지만 버전은 기본적으로 저장소를 살펴볼 때 버전 번호에 따라 정렬된다.

클래스 확장

우리의 테스트를 초록색으로 만들어줄 메서드를 구현해보자.

 아래 두 메서드를 Integer 클래스에 정의하고, 각 메서드를 ∗perfect라 불리는 프로토콜에 집어 넣자. 새로운 경계 테스트도 추가하라. 테스트가 초록색 (green) 인지 확인하라.

```
Integer>>isPerfect
    ^ self > 1 and: [self divisors sum = self]

Integer>>divisors
    ^ (1 to: self - 1 ) select: [ :each | (self rem: each) = 0 ]

PerfectTest>>testPerfectBoundary
    self assert: 0 isPerfect not.
    self assert: 1 isPerfect not.
```

Integer 에서 메서드는 ***Perfect*** 범주에 속하진 않지만, 이름이 ∗ 로 시작되고 패키지명을 매치하기 때문에 Perfect 패키지에 속한다. 그러한 메서드들은 기존 클래스를 *확장하기* 때문에 클래스 확장으로 알려진다. 이러한 메서드들은 Perfect 패키지를 로딩하는 사람*만* 이용 가능하다.

"깨끗한" 패키지와 "더러운" 패키지

어떤 개발 툴이든 그것이 포함된 패키지에서 코드를 수정하면 패키지를 더럽게 만든다. 즉, 이미지 내 패키지 버전은 저장 또는 로딩된 버전과 다르다는 의미다.

[8] 과거에는 개발자들이 이니셜만 이용해 변경 내용을 로그하는 것이 규칙이었다. 현재는 같은 이니셜을 공유하는 많은 개발자들이 "apblack"이나 "AndrewBlack"과 같이 풀 네임을 기반으로 한 식별자를 이용하는 것을 규칙으로 한다.

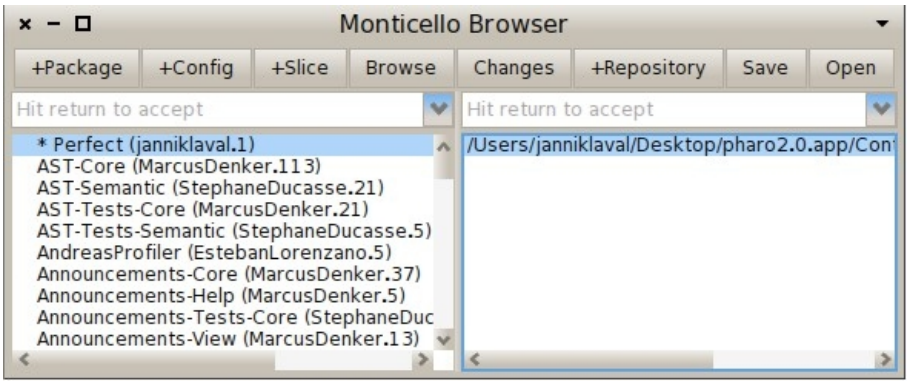


그림 7.4: Perfect 패키지를 수정하면 "더러워질" 것이다.

Monticello 브라우저에서 더러운 패키지는 이름 앞에 붙은 별표(∗)로 확인된다. 이는 변경 내용을 적용하지 않은, 따라서 변경 내용이 손실 되지 않기 위해서 저장소로 저장되어야 하는 패키지를 나타낸다. 더러운 패키지를 저장하면 깨끗해진다.

Browser 와 Changes 버튼이 하는 일을 확인하고 싶다면 눌러보기 바란다. 변경 내용을 ***Perfect*** 패키지로 저장하려면 Save 를 이용하라. 패키지가 다시 "깨끗해"졌는지 확인하라.

Repository 인스펙터

저장소 내용은 저장소 인스펙터(repository inspector)를 이용해 살펴볼 수 있는데, 인스펙터는 Monticello 의 Open 버튼을 이용해 시작된다 (그림 7.5).

package-cache 저장소를 선택하고 열면 그림 7.5과 같은 모습이 표시될 것이다.

저장소 내 모든 패키지는 인스펙터의 좌측에 열거된다:

- 밑줄이 그어진 패키지명은 해당 패키지가 이미지 내에 설치되었음을 의미한다.
- **굵은 글씨체로 밑줄이 그어진 패키지명**은 패키지가 설치되어 있으나 저장소 내에 더 최신 버전이 존재함을 의미한다.
- 일반 글씨체로 된 패키지명 은 패키지가 이미지 내에 설치되어 있지 않음을 의미한다.

패키지를 선택하면 선택된 패키지의 버전이 우측 패인에 열거된다:

- 밑줄이 그어진 버전 이름은 이미지 내에 해당 버전이 설치되었음을 의미한다.

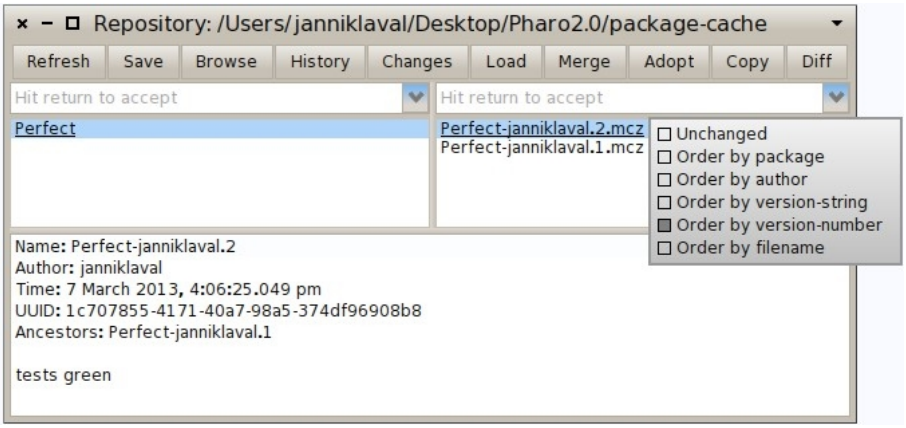


그림 7.5: 저장소 인스펙터.

- **굵은 글씨체의 버전 이름**은 해당 버전이 설치된 버전의 조상이 아님을 의미한다. 즉, 이것이 더 최신 버전이거나, 설치된 버전의 다른 branch에 속한다는 말이다.
- 일반 글씨체로 된 버전 이름 은 설치된 최신 버전보다 조금 더 오래된 버전임을 나타낸다.

인스펙터 우측을 액션 클릭하면 여러 가지 정렬 옵션이 열거된 메뉴가 열린다. 메뉴의 unchanged 엔트리는 어떤 특정 정렬이든 취소하고, 저장소에 의해 주어진 정렬을 사용한다.

패키지 로딩, 언로딩, 업데이트하기

현재로선 package-cache 저장소에 두 개의 Perfect 패키지 버전이 안전하게 보관되어 있다. 이제 해당 패키지를 언로딩하고 기존 버전을 로딩하여 업데이트하는 방법을 살펴보겠다.

Monticello 브라우저에서 Perfect 패키지와 그 저장소를 선택하라. 패키지명을 액션 클릭하고 unload package를 선택하라.

이제 Perfect 패키지가 이미지에서 사라졌음을 확인할 수 있을 것이다!

Monticello 브라우저에서, 패키지 패인에서 어떤 내용도 선택하지 않고 저장소 패인에 ***package-cache*** 를 선택한 후 저장소 인스펙터를 Open 하라. 아래로 스크롤하여 ***Perfect*** 패키지를 선택하면 일반 인터페이스에 그것이 설치되지 않았음이 표시될 것이다. 이제 패키지의 첫번째 버전을 선택하고 Load 하라.

원본(빨간색) 테스트만 로딩되었음을 확인할 수 있을 것이다.

저장소 인스펙터에서 ***Perfect*** 패키지의 두 번째 버전을 선택하고 Load 하라. 이제 패키지가 최신 버전으로 업데이트되었다.

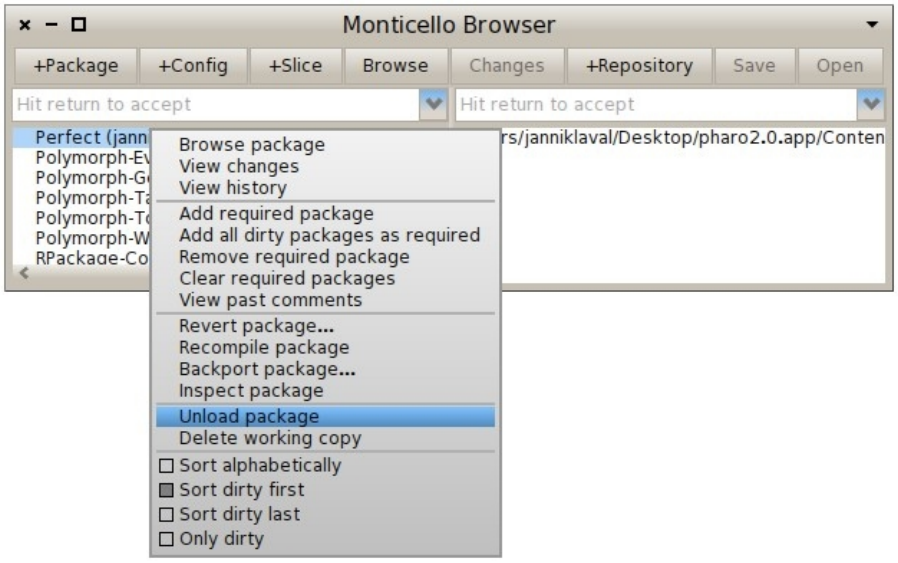


그림 7.6: 패키지 언로딩하기.

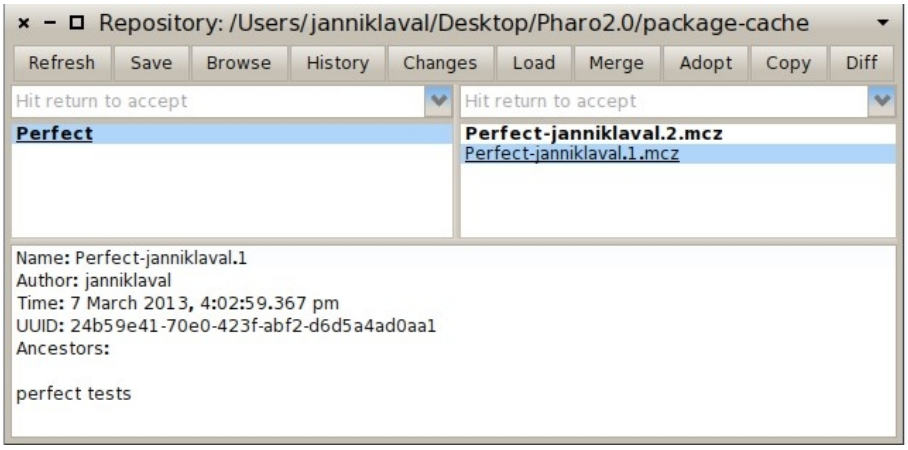


그림 7.7: 이전 (earlier) 버전 로딩하기.

이제 다시 테스트는 초록색이 되었을 것이다.

branching

branch는 다른 계열과 독립적으로 존재하지만 충분히 멀리 되돌아보면 공통된 조상 버전을 공유하는 개발 버전의 계열이다.

당신은 패키지를 저장할 때 새 버전 branch를 생성할 수 있다. 브랜칭은 새로운 병렬 (par-

allel) 개발을 목표로 할 때 유용하다. 예를 들어, 기업 내에서 소프트웨어를 관리하는 것이 당신의 일이라고 가정해보자. 어느 날 다른 부서에서 동일한 소프트웨어가 일을 약간만 달리 수행하도록 몇 가지만 수정하여 줄 것을 요청한다고 치자. 이러한 상황을 처리하기 위해서는 첫 번째 branch는 손대지 않고 두면서 수정(tweaks)을 병합하는 프로그램의 두 번째 branch를 생성해야 한다.

저장소 인스펙터에서 ***Perfect*** 패키지의 첫번째 버전을 선택한 후 Load 하라. 두번째 버전이 굵은 글씨체로 다시 표시가 되는데, 더 이상 로딩되지 않음을 의미한다 (첫번째 버전의 조상이 아니기 때문이다). 이제 두 개의 ***Integer*** 메서드를 구현하여 *∗**perfect*** 프로토콜에 위치시킨 후, 기존의 ***PerfectTest*** 테스트 메서드를 아래와 같이 수정하라.

```
Integer>>isPerfect
    self < 2 ifTrue: [ ^ false ].
    ^ self divisors sum = self

Integer>>divisors
    ^ (1 to: self - 1 ) select: [ :each | (self \\ each) = 0]

PerfectTest>>testPerfect
    self assert: 2 isPerfect not.
    self assert: 6 isPerfect.
    self assert: 7 isPerfect not.
    self assert: 28 isPerfect.
```

완전수에 대한 우리 구현이 약간 다르겠지만 다시 테스트는 초록색이 될 것이다.

Perfect 패키지의 두번째 버전의 로딩을 시도해보라.

변경 내용이 저장되지 않았다는 경고를 받을 것이다.

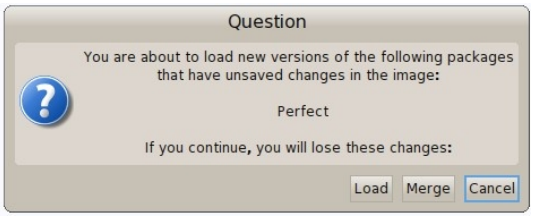


그림 7.8: 저장되지 않은 변경 내용을 알리는 경고.

Cancel 을 선택해 새 메서드의 오버라이드를 피하라. 이제 변경 내용을 Save 하라. 로그 메시지를 입력하고 새 버전을 Accept 하라.

축하한다! **Perfect** 패키지의 새 branch 가 성공적으로 생성되었다.

저장소 인스펙터가 아직 열려 있다면 새 버전으로 Refresh 하라 (그림 7.9).

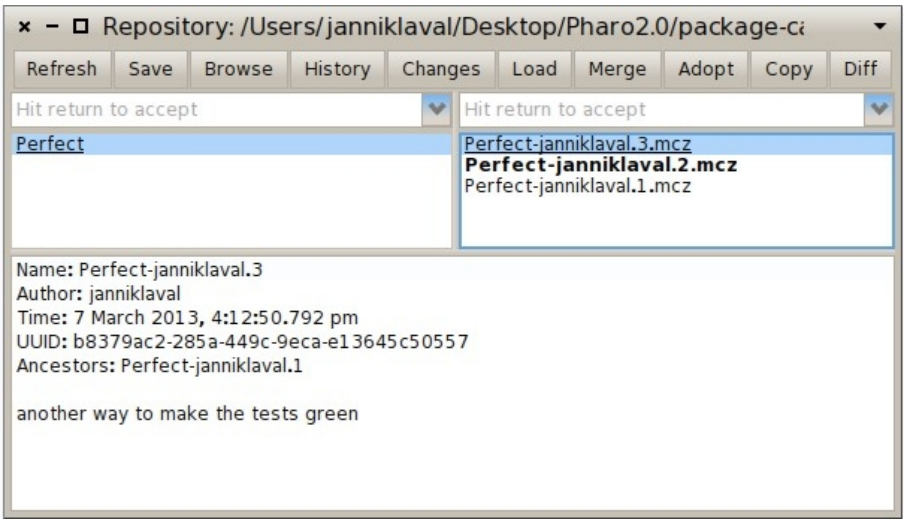


그림 7.9: 두번째 버전과 세번째 버전은 각각 첫번째 버전의 구분된 branch다.

병합하기 (merging)

Monticello 브라우저에서 Merge 버튼을 이용해 패키지의 한 버전을 다른 버전과 병합하는 것도 가능하다. 보통은 (i) 오래된 버전으로 작업하고 있음을 발견할 때나, (ii) 이전에 독립적이었던 branch들이 재병합되어야 하는 경우에 이 작업이 필요할 것이다. 두 상황 모두 여러 개발자들이 동일한 패키지를 작업할 때 흔히 겪는 시나리오에 해당한다.

그림 7.10의 왼쪽 그림에서 보는 바와 같이 Perfect 패키지의 현재 상황을 고려해보자. 우리는 첫번째 버전을 기반으로 새로운 세번째 버전을 공개했다. 두번째 버전 또한 첫번째 버전을 기반으로 하므로 두번째 버전과 세번째 버전은 독립된 branch가 된다.

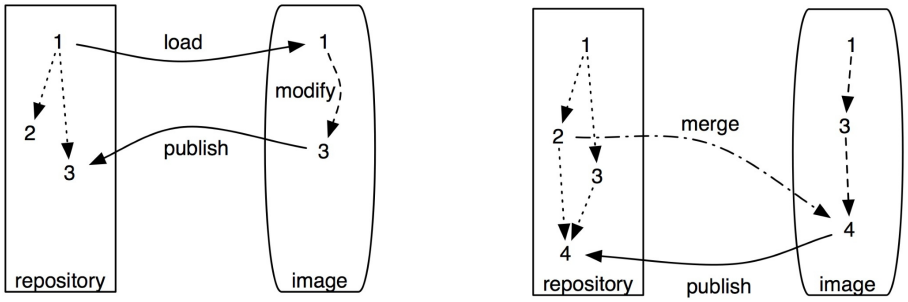


그림 7.10: 브랜칭 (왼쪽) 과 병합 (오른쪽).

이 시점에서 우리는 두번째 버전에 대한 변경 내용을 세번째 버전의 변경 내용과 병합하면 좋겠다고 생각한다. 세번째 버전이 현재 로딩되어 있기 때문에 두번째 버전의 변경 내용으로 병합하여 그림 7.10에서 오른쪽 그림과 같이 병합된 네번째 버전을 공개하고자 한다.

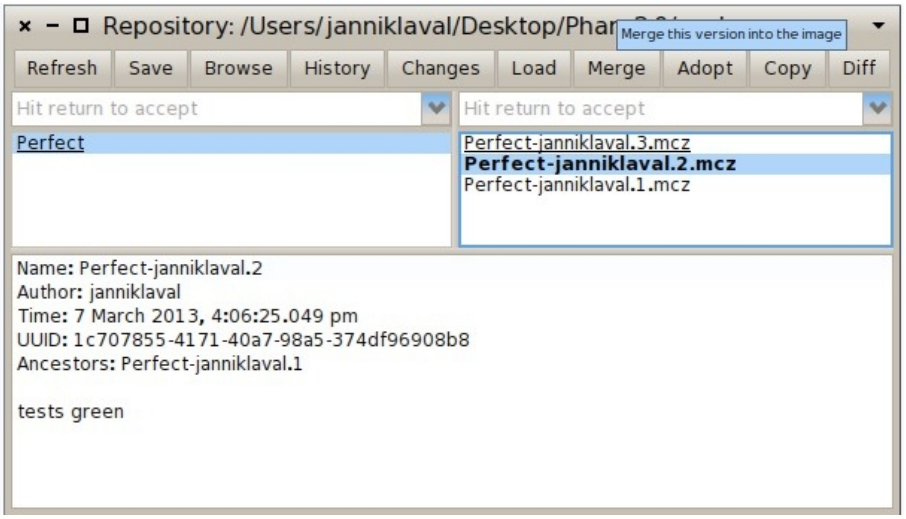


그림 7.11: 병합되어야 할 구분된 branch(굵은 글씨) 선택하기.

 저장소 브라우저에서 그림 7.11과 같이 두번째 버전을 선택하고 Merge 버튼을 클릭하라.

병합 툴은 정밀한 버전 병합을 허용한다. 병합될 패키지에 포함된 요소들은 상단 텍스트 패인에 열거된다. 하단 텍스트 패인은 선택된 요소의 정의를 표시한다.

그림 7.12을 보면 Perfect 패키지에 대한 두번째 버전과 세번때 버전간의 차이점이 세 가지가 있다. PerfectTest≫testPerfectBoundary 메서드가 새로 나타나고, 표시된 Integer의 메서드 두 개가 변경되었다. 하단 패인에는 Integer≫isPerfect의 소스 코드에 대한 기존 버전과 새 버전이 표시된다. 새 코드는 빨간색으로, 제거된 코드는 파란색 취소선으로 표시되고, 변경되지 않은 코드는 검정색으로 나타난다.

메서드나 클래스는 그 정의가 수정된 경우 서로 충돌한다. 그림 7.12는 Integer 클래스에 두 개의 충돌하는 메서드, isPerfect와 divisors를 보여준다. 충돌하는 패키지 요소는 밑줄, 취소선, 또는 **굵은 글씨체**로 표시된다. 글씨체 규칙은 아래와 같다.

일반 글씨체 = 충돌 없음. 일반 글씨체는 정의가 충돌하지 않음을 의미한다. 예를 들어 PerfectTe st≫testPerfectBoundary 메서드는 기존 메서드와 충돌하지 않으므로 설치할 수 있다.

빨간색 = 메서드가 충돌한다. 제안한 변경 내용을 고수하거나 거부하도록 결정을 내릴 필요가 있다. 제안한 메서드 Integer≫isPerfect 는 이미지에 있는 기존의 정의와 충돌한다. 충돌은 메

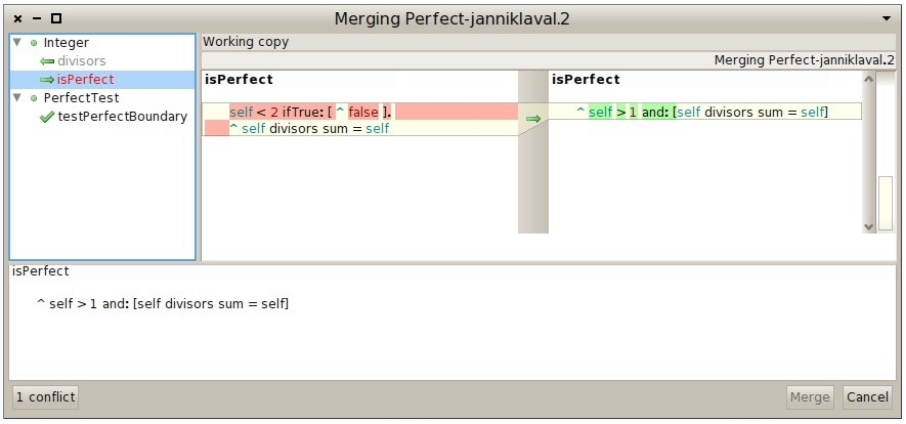


그림 7.12: 현재 세번째 버전과 병합 중인 Perfect 패키지의 두번째 버전.

서드를 오른쪽 마우스로 클릭하여 Keep current version(현재 버전 유지) 또는 Use incoming version(새로운 버전 사용)을 통해 해결이 가능하다.

오른쪽 화살표 = 저장소가 현재 버전을 대체한다. 오른쪽 화살표가 표시된 요소가 사용되고 이미지 내 현재 버전을 대체할 것이다. 그림 7.12에서 두번째 버전의 Integer≫isPerfect 가 사용되었음을 볼 수 있다.

왼쪽 화살표 = 저장소 버전이 거부되었다. 왼쪽 화살표가 표시된 요소는 거부되고, 로컬 정의는 대체되지 않을 것이다. 그림 7.12는 두번째 버전의 Integer≫divisors 가 거부되었기 때문에 세번째 버전의 정의가 그대로 유지될 것을 보여준다.

 **Integer≫isPerfect** 의 새로운 버전을 사용해 **Integer≫divisors** 의 현재 버전을 유지하고, Merge 버튼을 클릭하라. 테스트가 모두 초록색인지 확인하라. Perfect 패키지의 새로 병합된 버전을 네번째 버전으로 적용하라.

지금 저장소 인스펙터를 새로고침하면, 굵은 글씨체로 표시된 버전이 더 이상 없음을 눈치챌 것인데, 이는 모든 버전이 현재 로딩된 네번째 버전의 조상이라는 의미다 (그림 7.13).

Monticello 저장소 살펴보기

Monticello에는 그 외에 유용한 기능들이 많다. 그림 7.1에서 볼 수 있듯이 Monticello 브라우저 창에는 8개의 버튼이 위치한다. 그 중 지금까지 4개-+Package, Save, +Repository, Open 버튼을 사용해보았다. 이제 저장소의 상태와 내역을 살펴보기 위해 Browse 와 Changes 버튼을 살펴보겠다.

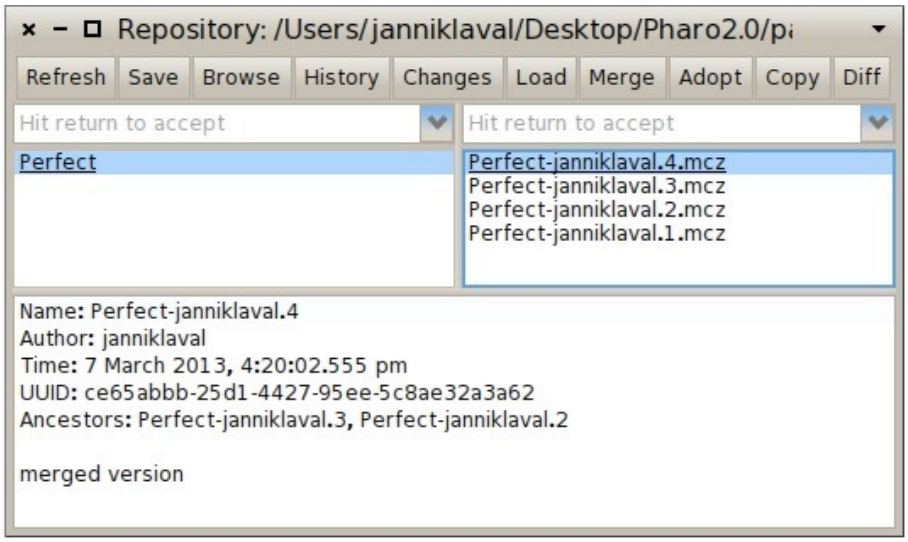


그림 7.13: 모든 오래된 버전들은 현재 병합되어있는 네번째 버전의 조상들이다.

Browse(탐색하기)

Browse 버튼을 이용하면 패키지의 내용을 표시하기 위해 "스냅샷 브라우저(snapshot browser)"가 열린다. 브라우저보다 스냅샷 브라우저를 사용할 때 이점은 클래스 확장을 표시하는 능력에 있다.

Perfect 패키지를 선택하고 Browse 버튼을 클릭하라.

예를 들자면, 그림 7.14는 Perfect 패키지에 정의된 클래스 확장을 표시한다. (환경이 그에 따라 준비되는 경우) 클래스명이나 메서드명을 액션 클릭할 경우 일반 브라우저를 열 수는 있지만 여기서 코드를 수정할 수는 없음을 주목하라.

패키지의 코드를 공개하기 전에 항상 브라우징하는 것은 훌륭한 실습이 되는데, 자신이 생각하는 내용을 실제로 포함하는지 확신할 수 있기 때문이다.

Changes(변경사항)

Changes 버튼은 이미지 내 코드와 저장소 내 가장 최신 패키지의 차이를 계산한다.

Monticello 브라우저에서 ***PerfectTest*** 에 아래와 같은 변경 내용을 적용하여 Changes 버튼을 클릭하라.

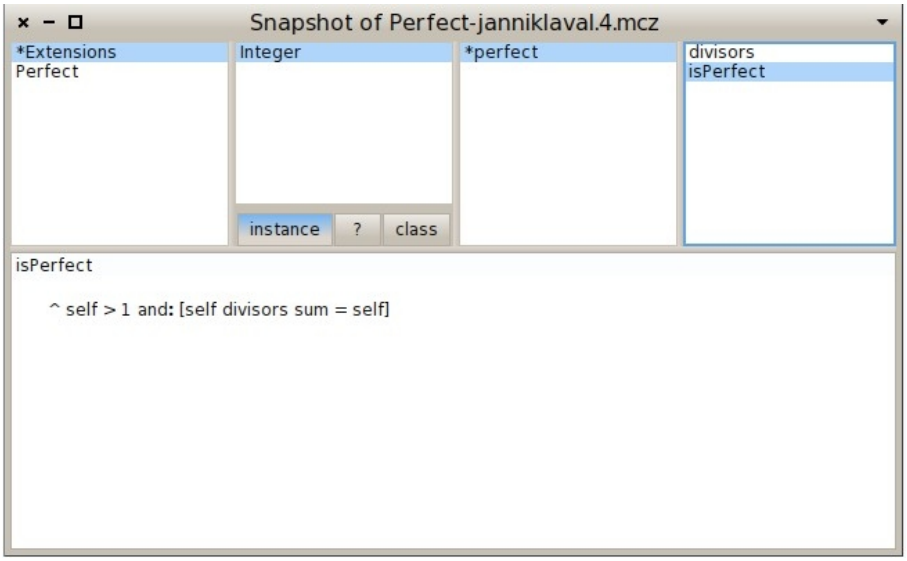


그림 7.14: 스냅샷 브라우저는 Perfect 패키지가 2개의 메서드가 있는 Integer 클래스를 확장함을 보여준다.

```
PerfectTest>>testPerfect
    self assert: 2 isPerfect not.
    self assert: 6 isPerfect.
    self assert: 7 isPerfect not.
    self assert: 496 isPerfect.

PerfectTest>>testPerfectTo1000
    self assert: ((1 to: 1000) select: [:each | each isPerfect]) = #(6 28 496)
```

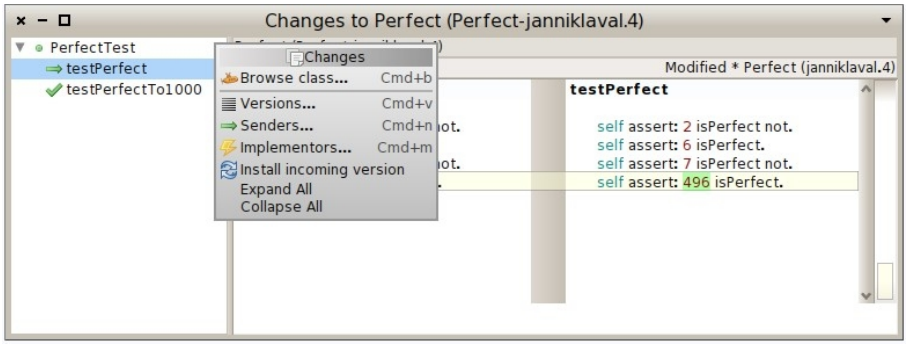


그림 7.15: 패치 브라우저는 이미지 내 코드와 가장 최근에 적용된 버전의 차이를 표시한다.

그림 7.15는 Perfect 패키지가 부분적으로 수정되어 하나의 메서드가 변경되고 하나의 새 메서드가 생성되었음을 보여준다. 늘 그렇듯 변경 내용을 액션 클릭하면 컨텍스트의 연산에

대한 선택권을 제공한다.

고급 주제

이제부터 살펴볼 내용은 몇 가지 고급 주제들로서, 히스토리, 의존성 관리, 구성하기, 클래스 초기화가 포함된다.

히스토리

패키지를 액션 클릭하여 History 항목을 선택하면 선택된 패키지의 각 버전과 함께 적용된 주석을 표시하는 버전 히스토리 뷰어가 열린다 (그림 7.16 참고). **Perfect** 의 경우 패키지 버전은 좌측에 열거되고, 선택된 버전에 관한 정보는 우측에 표시된다.

Perfect 패키지를 선택하고 오른쪽 마우스를 클릭한 후 History 항목을 선택하라.

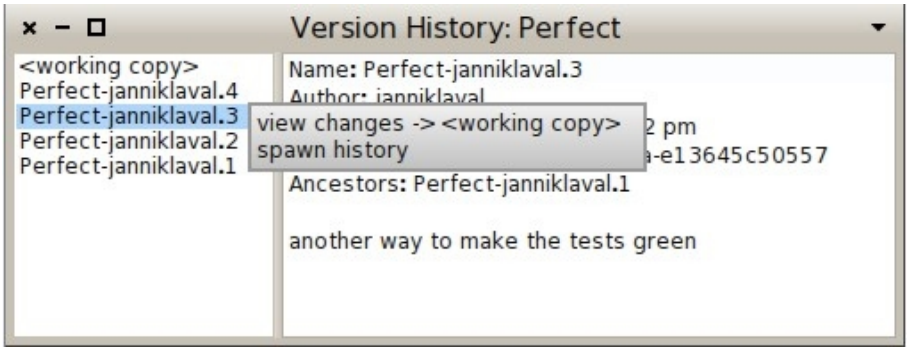


그림 7.16: 버전 히스토리 뷰어는 패키지의 다양한 버전에 관한 정보를 제공한다.

특정 버전을 액션 클릭하면 이미지에 로딩된 패키지의 현재 작업 중인 복사본과 관련된 변경 내용을 살펴보거나, 선택된 버전에 관련해 새 히스토리 브라우저를 야기할 수 있다.

의존성

대부분의 애플리케이션은 독립적으로 살아갈 수 없으며, 적절하게 작동하기 위해선 다른 패키지들이 필요하다. 가령 meta-described 내용 관리 시스템인 Pier[9]를 살펴보도록 하자. Pier는 다양한 측면(툴, 문서, 블로그, catch 전략, 보안 등)을 가진 커다란 소프트웨어 조각이다. 각 측면은 구분된 패키지에 의해 구현된다. 대부분의 Pier 패키지들은 독립적으로 사용될 수가 없는데, 다른 패키지에서 정의된 메서드와 클래스를 참조하기 때문이다. Monticello는 주어진 패키지에서 요구되는 패키지를 선언하는 의존성 메커니즘을 제공하여 그것이 올바르게 로딩되도록 보장한다.

[9] http://source.lukas-renggli.ch/pier

근본적으로 의존성 메커니즘은 패키지 자체가 로딩되기 전에 그것이 필요로 하는 모든 패키지가 로딩되도록 보장한다. 요구되는 패키지들 또한 다른 패키지들을 필요로 하기 때문에 이 과정은 의존성 트리에 재귀적으로 적용되어 트리의 leaf들은 그들이 의존하는 branch보다 먼저 로딩되도록 한다. 요구되는 패키지의 새 버전이 확인(check in)될 때마다 그에 의존하는 패키지의 새 버전들은 자동으로 새 버전에 의존할 것이다.

> *의존성은 저장소에 걸쳐서 표현될 수 없다.* 요구되고 요구하는 모든 패키지들은 동일한 저장소에 상주해야 한다.

그림 7.17은 Pier에서 이것이 어떻게 작용하는지를 보여준다. Pier-All 패키지는 일종의 우산 역할을 하는 빈 페키지다. 이는 Pier-Blog, Pier-Caching과 다른 모든 Pier 패키지들을 필요로 한다.

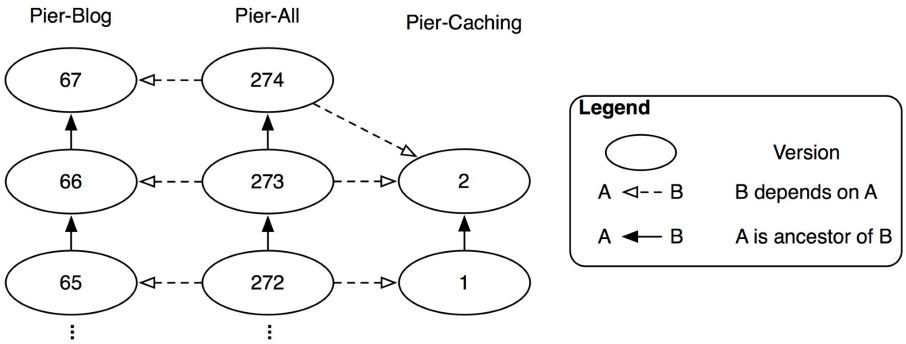


그림 7.17: Pier에서 의존성.

이러한 의존성 때문에 Pier-All을 설치 시 다른 모든 Pier 패키지들이 설치되는 결과가 야기된다. 뿐만 아니라 개발 시 저장해야 하는 유일한 패키지는 Pier-All로, 모든 의존적인 더러운 패키지들은 자동으로 저장된다.

실제로 어떻게 작동하는지 살펴보도록 하자. 우리의 Perfect 패키지는 현재 implementation과 함께 테스트를 번들한다. 대신 이것들을 구분된 패키지로 구분하여 implementation을 테스트 없이 로딩할 수 있기를 원한다고 가정하자. 하지만 기본적으로는 모든 것을 로딩하길 원한다.

아래 단계를 따라하라.

- *패키지 캐시에서* ***Perfect*** *패키지의 네번째 버전을 로딩하라.*
- *브라우저에* ***NewPerfect-Tests*** *라는 새 패키지를 생성하고 해당 패키지로* ***PerfectTest*** *를 드래그하라.*
- ***Integer*** *클래스의* ***∗perfect*** *프로토콜을* ***∗newperfect-extensions****(재명명은 Action 클릭을 통해)로 재명명하라.*

- *Monticello 브라우저에서 **NewPerfect-All** 과 **NewPerfect-Extensions** 패키지를 추가하라.*
- ***NewPerfect-Extensions** 와 **NewPerfect-Tests** 를 요구되는 패키지로서 **NewPerfect-All** 에 추가하라(**NewPerfect-All**에서 액션 클릭).*
- *package-cache 저장소에 **NewPerfect-All** 패키지를 저장하라. Monticello는 요구되는 패키지를 저장하기 위해 주석의 입력을 요구할 것이다.*
- *세 개의 패키지가 모두 package cache 에 저장되었는지 확인하라.*
- *Monticello는 **Perfect** 가 여전히 로딩되었다고 생각한다. 이를 언로딩한 후 저장소 인스펙터로부터 **NewPerfect-All** 를 로딩하라. 그러면 **NewPerfect-Extensions** 와 **NewPerfect-Tests** 가 요구하는 패키지들과 함께 로딩될 것이다.*
- *모든 테스트가 실행되는지 확인하라.*

Monticello 브라우저에서 **NewPerfect-All** 을 선택하면 의존하는 패키지들이 **굵은 글씨체**로 표시된다 (그림 7.18 참고).

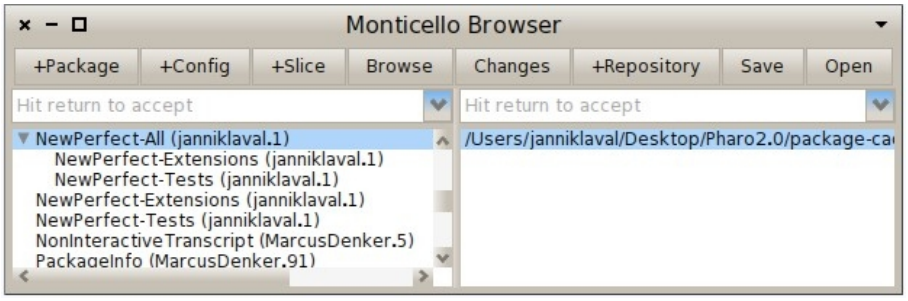


그림 7.18: NewPerfect-All 은 NewPerfect-Extensions와 NewPerfect-Tests 를 필요로 한다.

> 추가로 Perfect 패키지를 개발할 경우 NewPerfect-All이 요구하는 패키지들은 제외하고 NewPerfect-All만 로딩 또는 저장해야 한다.

그 이유는 다음과 같다.

- 저장소에서 (package-cache 또는 다른 곳에서) NewPerfect-All 을 로딩 시 같은 저장소에서 NewPerfect-Extensions와 NewPerfect-Tests 가 로딩될 것이다.
- PerfectTests 클래스를 수정할 경우 NewPerfect-Tests와 NewPerfect-All 패키지 둘 다 더러워질 것이다 (NewPerfect-Extensions는 해당하지 않음).
- 변경 내용을 적용하려면 NewPerfect-All 을 저장해야 한다. 이는 NewPerfect-All 의 새 버전을 시작하여 후에 NewPerfect-Tests의 새 버전을 필요로 한다. (이 또한 기존에 수정되지 않은 NewPerfect-Extensions 버전에 의존할 것이다.) NewPerfect-All의 최신 버전을 로딩 시 요구되는 패키지의 최신 버전도 로딩할 것이다.

- 대신 NewPerfect-Tests를 저장하면 NewPerfect-All은 저장되지 않을 것이다. 이는 의존성을 효과적으로 파괴하기 때문에 바람직하지 못하다. 이후 NewPerfect-All의 최신 버전을 로딩할 경우 필요로 하는 패키지들의 최신 버전을 얻을 수 없다. 삼가하라!

> 최상위 수준 패키지의 이름에 자신의 하위패키지에 일치하는 접두사를 붙여 명명하지 말라(예 : Perfect). Perfect를 Perfect-Extensions 또는 PerfectTest가 필요로 하는 패키지로 정의하지 않도록 하라. 두 개의 패키지만 사용을 원하고 하나는 빈 채로 최상의 수준에 남겨두고자 할 때에도 Monticello가 세 패키지에 대해 모든 클래스를 저장할 것이다.

패키지들 간 좀 더 유연한 의존성을 빌드하기 위해서는 Metacello 설정(configuration)의 사용을 권한다 (제 9장 참조). +Config 버튼을 이용하면 일종의 설정 구조를 생성한다. 이 때는 의존성만 추가하면 될 일이다.

클래스 초기화

Monticello가 패키지를 이미지로 로딩하면 클래스 측에 initialize 메서드를 정의하는 클래스에게 initialize 메시지가 전송될 것이다. 메시지는 클래스 측에서 해당 메서드를 정의하는 클래스에게만 전송된다. 해당 메서드를 정의하지 않는 클래스는 초기화되지 않을 것이며, 그 슈퍼클래스들 중 하나가 initialize를 정의하더라도 마찬가지다. 주의 : initialize 메서드는 패키지를 재로딩한다고 해서 호출되지 않는다!

클래스 초기화를 이용해 원하는 횟수만큼 검사 또는 특수 액션을 실행할 수 있다. 클래스에 새 인스턴스 변수를 추가 시 특히 유용하겠다.

클래스 확장은 클래스로 새 메서드를 추가하는 것으로 엄격히 제한된다. 하지만 때때로 확장 메서드는 새 인스턴스 변수의 존재를 필요로 할 수도 있다.

예를 들어 SUnit의 TestCase 클래스를 확장하여 테스트가 마지막으로 빨간색이었던 히스토리를 기록하는 메서드가 포함되길 원한다고 가정해보자. 어딘가에 정보를 보관해야 하지만 불행히도 인스턴스 변수를 확장의 일부로 정의할 수는 없다.

여기서 해결책은 클래스들 중 하나의 클래스 측에 initialize 메서드를 정의하는 방법이 된다.

```
TTestCaseExtension class>>initialize
    (TestCase instVarNames includes: 'lastRedRun')
        ifFalse: [TestCase addInstVarName: 'lastRedRun']
```

우리 패키지가 로딩되면 위의 코드가 평가되고 인스턴스 변수가 존재하지 않을 경우 추가될 것이다. 자신의 패키지에 없는 클래스를 변경하더라도 다른 패키지가 더러워질 것이란 사실을 명심하라. 앞의 예제에서 SUnit 패키지는 TestCase를 포함한다. TestCaseExtension를 설치한 후 SUnit 패키지는 더러워질 것이다.

두 버전으로부터 변경 집합 얻기

Monticello 버전은 하나 또는 그 이상의 패키지에 대한 스냅샷이다. 각 버전은 기본 패키지를 구성하는 클래스 및 메서드 정의의 완전한 집합을 포함한다. 때로는 두 개의 버전으로부터 ”

패치(patch)"를 얻는 편이 유용하다. 패치는 어떤 버전 A로부터 다른 버전 B로 이동하기 위해 시스템에 필요한 모든 부가적 효과의 집합을 의미한다.

변경 집합(change set)은 시스템 패치를 정의하기 위해 Pharo에 내장된 메커니즘이다. 변경 집합은 시스템에 미치는 전역적인 부가적 효과로 구성되어 있다. 새로운 변경 집합은 Change Sorter 에서 생성 및 편집할 수 있다. 해당 툴은 World▷Tools 엔트리에서 이용 가능하다.

두 개의 Monticello 버전 간 차이는 패키지의 두 번째 버전을 로딩하기 전에 새 변경 집합을 생성하면 쉽게 포착이 가능하다. 예로, Perfect 패키지의 첫번째 버전과 두번째 버전의 차이를 포착해보겠다.

1. Monticello 브라우저에서 Perfect 의 첫번째 버전을 로딩하라.
2. Change sorter(변경 정렬기)를 열고 새 변경 집합을 생성하라. DiffPerfect 로 명명하자.
3. 두번째 버전을 로딩하라.
4. Change sorter를 보면 첫번째 버전과 두번째 버전의 차이점이 표시될 것이다. 변경 집합을 액션 클릭하고 file out을 선택하면 변경 집합이 파일시스템에 저장될 것이다. **DiffPerfect.X.cs** 파일은 이제 당신의 Pharo 이미지 옆에 위치한다.

저장소 종류

Monticello가 지원하는 저장소 종류에는 여러 가지가 있으며, 각각이 다른 특성과 용도를 가진다. 저장소는 읽기만 가능하거나, 쓰기만 가능하거나, 읽기-쓰기가 가능하다. 접근 권한은 전역적으로 정의되거나 특정 사용자에게 (예: SmalltalkHub에서와 같이) 지정할 수 있다.

HTTP. HTTP 저장소는 SmalltalkHumb가 지원하는 저장소 종류라 아마도 가장 많이 사용되는 저장소 종류일 것이다.

HTTP 저장소의 좋은 점은 웹 사이트로부터 특정 버전으로 직접 연계하기가 쉽다는 점이다. HTTP 서버에 약간의 구성만 작업하면 HTTP 저장소는 일반 웹 브라우저, WebDAV 클라이언트 등을 통해 브라우징이 가능하다.

HTTP 저장소는 SmalltalkHub 외에 HTTP 서버와 함께 사용할 수 있다. 예를 들어, 간단한 구성만으로[10] Apache를 제한된 접근 권한을 가진 Monticello 저장소로 전환한다.

[10] http://www.visoracle.com/squeak/faq/monticello-1.html

```
"My apache2 install worked as a Monticello repository right out of the box on my
RedHat 7.2 server. For posterity's sake, here's all I had to add to my apache2 config:"
Alias /monticello/ /var/monticello/
<Directory /var/monticello>
    DAV on
    Options indexes
    Order allow,deny
    Allow from all
    AllowOverride None
    # Limit write permission to list of valid users.
    <LimitExcept GET PROPFIND OPTIONS REPORT>
        AuthName "Authorization Realm"
        AuthUserFile /etc/monticello-auth
        AuthType Basic
        Require valid-user
    </LimitExcept>
</Directory>
"This gives a world-readable, authorized-user-writable Monticello repository in
/var/monticello. I created /etc/monticello-auth with htpasswd and off I went.
I love Monticello and look forward to future improvements."
```

FTP. 이는 HTTP 저장소와 비슷한데, 대신 FTP 서버를 사용한다는 점이 다르다. FTP 서버 또한 제한된 접근 권한을 제공하여 다른 FTP 클라이언트들을 이용해 Monticello 저장소와 같은 대상을 살펴볼 수 있다.

GOODS. 해당 저장소는 버전을 GOODS 객체 데이터베이스에 보관한다. GOODS는 완전 분산형 객체지향 데이터베이스 관리 시스템으로, 능동적인 클라이언트 모델을 사용한[11]. 이는 읽기-쓰기가 가능한 저장소기 때문에 버전을 저장하거나 검색하기에 훌륭한 "작업" 저장소가 된다. GOODS가 제공하는 거래(transaction) 지원, 저널, 복사(replication) 기능 때문에 다수의 클라이언트가 사용하는 대규모 저장소에 적합하다.

Directory(디렉터리). 디렉터리 저장소는 로컬 파일 시스템 내 디렉터리로 버전을 보관한다. 준비하는 데에 약간의 수고만 요하므로 개인(private)적인 프로젝트에 유용하고, 네트워크 연결을 요구하지 않으므로 연결해제된 개발에 유일한 옵션이 된다. 이번 장의 실습에 사용된 package-cache가 이러한 저장소 유형에 해당한다. 디렉터리 저장소 내 버전은 추후 public 또는 공유 저장소로 복사할 수 있다. SmalltalkHub는 주어진 프로젝트로 패키지 버전을 (.mcz 파일) 가져오도록 허용함으로써 이러한 기능을 지원한다. SmalltalkHub 로 로그인하고, 프로젝트를 검색한 후 Import Versions 링크를 클릭하면 된다.

Directory with Subdirectories(하위디렉터리가 있는 디렉터리). "하위디렉터리가 있는 디렉터리"는 "디렉터리"와 매우 유사하지만 이용 가능한 패키지 리스트를 검색 시 하위디렉터리를 살펴본다는 점에서 다르다. 단층(flat) 디렉터리가 모든 패키지 버전을 포함하는 대신 저장소가 하위디렉터리로 계층적으로 구조화된다.

SMTP. SMTP 저장소는 버전을 메일로 전송할 때 유용하다. SMTP 저장소를 생성할 때는 대상

[11] http://www.garret.ru/goods.html

이메일 주소를 명시한다. 이는 가령 패키지의 관리자처럼 다른 개발자의 주소가 되기도 하고 pharo-project와 같은 우편 목록이 되기도 한다. 해당 저장소에 저장된 버전은 모두 이 주소로 메일을 통해 전송된다. SMTP 저장소는 쓰기만 가능하다.

Programmatically adding repositories(계획에 따라 추가하는 저장소). 특별한 목적을 위해 새 저장소를 계획에 따라 추가해야 하는 경우가 있다. 이는 구성 또는 분산된 Monticello 패키지의 거대한 집합을 관리하거나, Monticello 브라우저에서 이용 가능한 엔트리를 단순히 맞춤 설정할 때에 발생한다. 예를 들어, 아래 코드 조각은 새 디렉터리 저장소를 계획에 따라 추가한다.

```
{'/path/to/repositories/project-1/'.
'/path/to/repositories/project-2/'.
'/path/to/repositories/project-3/'. } do:
[ :path |
    repo := MCDirectoryRepository new directory:
        (path asFileReference).
    MCRepositoryGroup default addRepository: repo ].
```

SmalltalkHub 사용하기

SmalltalkHub는 온라인 저장소로서, 자신의 Monticello 패키지를 보관하는 데에 사용할 수 있다. 인스턴스는 `http://smalltalkhub.com/` 실행 및 접근 가능하다.

 웹 브라우저를 이용해 메인 페이지, `http://smalltalkhub.com/` 를 방문하라. 프로젝트를 선택하면 아래와 같은 *저장소 표현식* 확인될 것이다.

```
MCHttpRepository
location: '\url{http://smalltalkhub.com/mc/PharoExtras/Phexample/main'}
user: \textit{
password: }
```

해당 저장소에서 +Repository *를 클릭한 후* HTTP *를 선택하여 저장소를 Monticello에 추가한다. 프로젝트에 해당하는 URL로 템플릿을 작성하라. 위의 저장소 표현식을 웹 페이지에서 복사해 템플릿에 붙여넣을 수도 있다. 해당 패키지의 새 버전을 적용하는 것이 아니기 때문에 사용자 이름과 비밀번호는 입력할 필요가 없겠다. 저장소를* Open *하고, Phexample 의 최신 버전을 선택한 후* Load *를 클릭하라.*

SmalltalkHub 계정이 없다면 SmalltalkHub 홈 페이지에서 Join 링크를 누르는 일이 첫 단계가 될 것이다. 회원이 되면 +New Project 를 이용 시 새 프로젝트를 생성할 수 있다.

SmalltalkHub는 프로젝트 저장소를 설정하기 위한 옵션을 제공한다 (그림 7.20 참고). 태그를 할당하고, 프로젝트에 참여하지 않는 사람들의 접근 권한을 제한하는 (private, public) 라이센스를 선택할 수 있고, 사용자를 프로젝트의 구성원으로 저장할 수도 있다. 프로젝트를 공유하는 팀을 생성할 수도 있다.

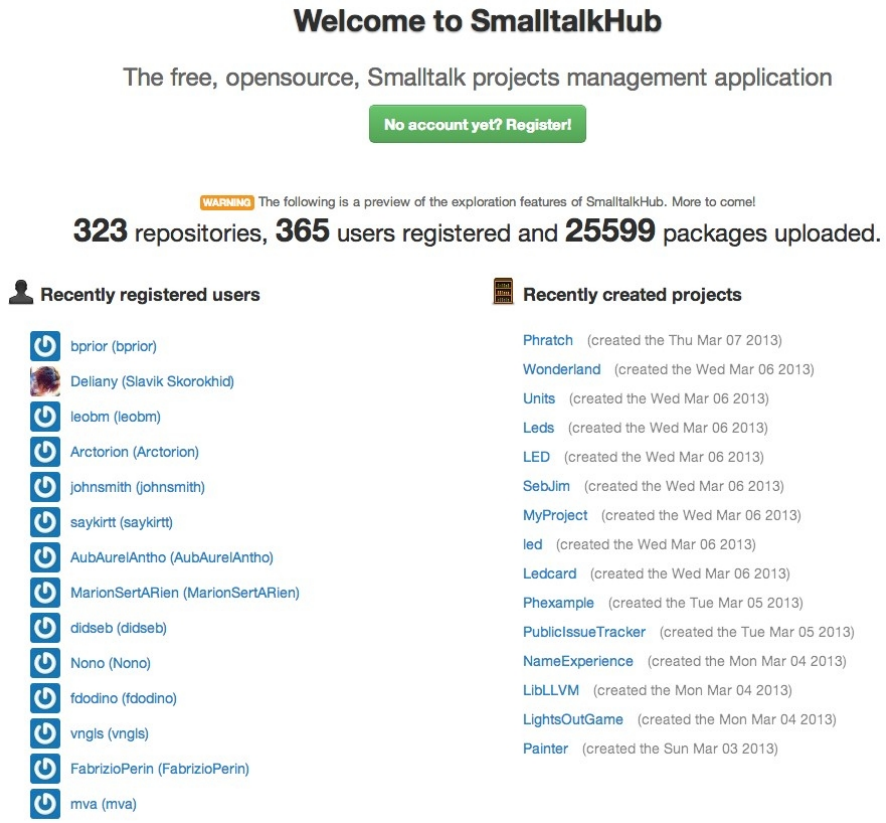


그림 7.19: SmalltalkHub, 온라인 Monticello 코드 저장소.

.mcz 파일 포맷

버전은 저장소에 이진 파일(binary file)로서 보관된다. 이러한 파일은 .mcz라는 확장자를 가진 파일이라 보통 "mcz 파일"이라고 부른다. mcs 파일은 소스 코드와 다른 메타 데이터를 포함하는 압축(zipped) 파일이기 때문에 "Monticello zip"이란 의미를 지닌다.

> Mcz 파일은 변경 집합처럼 오픈 이미지 파일로 드래그 앤 드롭이 가능하다. Pharo 는 그것이 포함하는 패키지의 로딩을 원하는지 사용자에게 질문할 것이다. 하지만 Monticello는 패키지가 어떤 저장소로부터 왔는지는 알지 못할 것이므로 개발 시에는 이러한 기법을 사용하지 않기를 바란다.

소스 코드를 직접 살펴보기 위해 파일의 압축 해제를 시도해 볼 수 있겠지만 최종 사용자는 스스로 이러한 파일을 압축 해제할 필요가 없는 것이 보통이다. 압축 해제 시 아래와 같은 mcs

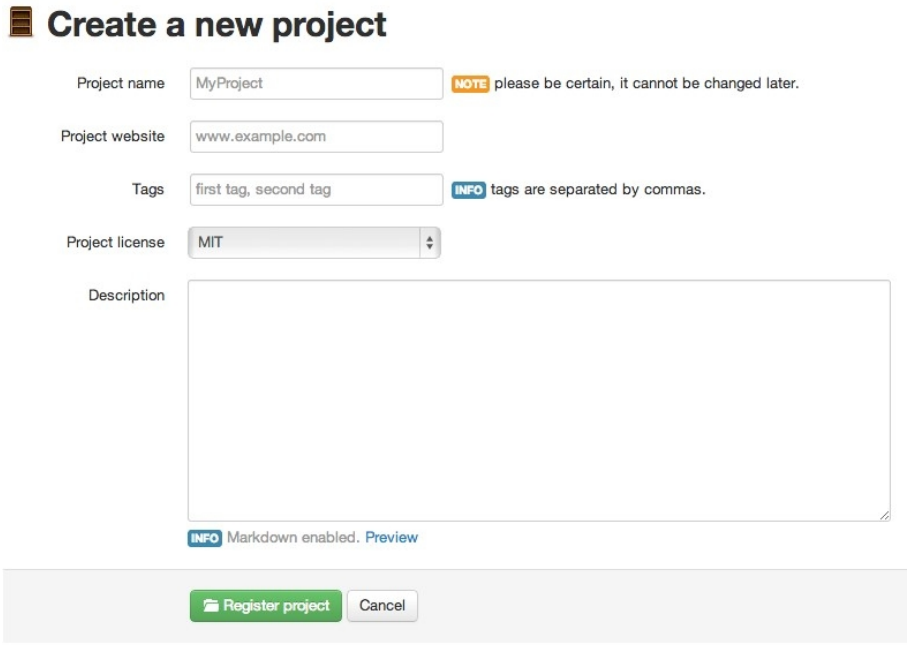


그림 7.20: SmalltalkHub에서 저장소를 설정 가능하다.

파일의 멤버(member)를 발견할 것이다.
파일 내용. Mcz 파일은 사실 특정 규칙을 따르는 ZIP 아카이브에 해당한다. 개념적으로 한 버전당 네 가지를 포함한다.

- *Package.* 버전은 특정 패키지와 관련이 있다. 각 mcz 파일은 패키지명에 관한 정보를 포함하는 "package"라는 파일을 갖고 있다.
- *VersionInfo.* 스냅샷에 관한 메타 데이터다. 작성자 이니셜, 스냅샷을 얻은 날짜와 시간, 스냅샷의 조상을 포함한다. 각 mcz 파일은 이러한 정보를 포함하는 "version"이란 멤버를 포함한다.
  버전은 소스 코드의 전체 히스토리를 포함하지는 않는다. 이는 특정 시점에 소스 코드의 스냅샷으로서, 스냅샷을 식별하는 UUID와, 그것이 파생된 이전의 모든 스냅샷의 UUID 기록을 포함한다.
- *Snapshot.* Snapshot은 특정 시간에 패키지의 상태에 대한 기록이다. 각 mcz 파일은 "snapshot/"이란 이름의 디렉터리를 포함한다. 해당 디렉터리 내 모든 멤버들은 프로그램 요소의 정의를 포함하는데, 이들을 결합하면 Snapshot이 형성된다. Monticello 최신 버전들은 이 디렉터리에 하나의 멤버, "source.st"만 생성한다.
- *Dependencies.* 버전은 다른 패키지의 특정 버전에 의존할 수 있다. Mcz 파일에는 각 의존성마다 하나의 멤버가 존재하는 "dependencies/" 디렉터리가 포함되어 있다. 이러한

멤버들은 Monticello 패키지가 의존하는 패키지의 이름을 따서 명명될 것이다. 예를 들어, Pier-All mcz 파일은 그 dependencies 디렉터리에 Pier-Blog와 Pier-Caching으로 명명된 파일들을 포함할 것이다.

소스 코드 인코딩. "snapshot/source.st"로 명명된 멤버는 패키지에 속하는 코드의 표준 file-out을 포함한다.

메타데이터 인코딩. 그 외 zip 아카이브의 멤버들은 S-표현식을 이용해 인코딩된다. 개념상 표현식은 nestable dictionary를 표현한다. 목록에서 각 요소 쌍은 키와 값을 나타낸다. 예를 들어, 아래는 AA로 명명된 패키지에서 "version" 파일을 발췌한 것이다:

```
(name 'AA-ab.3' message 'empty log message' date '10 January 2008' time '10:31:06 am' author 'ab' ancestors ((name 'AA-ab.2' message
```

이는 AA-ab.3 버전에 빈 로그 메시지가 있고, ab에 의해 2008년 1월 10일에 생성되었으며, AA-ab.2, …라는 조상이 있다는 의미다.

요약

본 장에서는 Monticello의 기능을 상세히 설명하였으며, 아래의 요점들을 다루었다.

- Monticello는 Smalltalk 범주와 메서드 프로토콜에 매핑된다. Foo라는 패키지를 Monticello에 추가하면 이는 Foo라는 범주 또는 Foo-로 시작되는 범주 내 모든 클래스를 포함할 것이다. 그러한 범주에 위치한 모든 메서드도 포함할 것이지만 ∗로 시작하는 프로토콜 내 메서드들은 해당하지 않는다. 마지막으로, 시스템 어디든 ∗foo 라는 프로토콜 또는 ∗foo-로 시작되는 프로토콜 내 모든 클래스 확장 메서드들도 포함할 것이다.
- 패키지 내 어떤 메서드나 클래스든 이를 수정할 경우 Monticello에 "dirty"로 표시될 것이며, 저장소로 저장이 가능하다.
- 저장소에는 많은 종류가 있으며, 가장 자주 사용되는 저장소는 SmalltalkHub가 사용하는 것과 같은 HTTP 저장소다.
- 저장된 패키지는 package-cache라는 디렉터리에 국부적으로 보관되는 캐시다.
- Monticello 저장소 인스펙터를 이용해 저장소의 브라우징이 가능하다. 어떤 패키지 버전을 로딩 또는 언로딩할 것인지 선택할 수 있다.
- 새 버전을 최신 버전보다 앞의 것에 해당하는 버전에 기반을 두어 패키지의 새 branch를 생성할 수 있다. 저장소 인스펙터는 패키지들의 계통(ancestry)을 추적하고, 특정 branch에 속하는 버전이 어떤 것인지 알려준다.
- branch는 병합이 가능하다. Monticello는 병합 버전들 간 충돌을 해결하는 데에 양호한 제어 수준을 제공한다. 병합된 버전은 계승된 두 버전을 조상으로 갖는다.
- Monticello는 패키지들 간 의존성을 추적할 수 있다. 필요로 하는 패키지에 대한 의존성을 가진 패키지가 저장되면 해당 패키지의 새 버전이 생성되어 요구되는 모든 패키지들의 최신 버전에 의존한다.

- 자신의 패키지에 포함된 클래스들이 클래스 측 initialize 메서드를 갖는 경우, 패키지를 로딩 시 initialize가 그러한 클래스들로 전송된다. 이러한 메커니즘을 이용해 다양한 검사 또는 스타트 업 액션을 실행할 수 있다. 특히 자신이 정의하는 확장 메서드에 해당하는 클래스로 새 인스턴스 변수를 추가할 때 유용하게 사용된다.
- Monticello는 파일 확장자가 .mcz인 특수 압축 파일로 패키지 버전을 보관한다. Mcz 파일은 당신의 패키지 버전의 전체 소스 코드에 대한 스냅샷 뿐만 아니라 패키지 의존성과 같은 다른 중요한 메타데이터가 들어 있는 파일들도 포함한다.
- 이미지에 mcz 파일을 드래그 앤 드롭하여 빠르게 로딩할 수 있다.

제 8 장

Gofer: 패키지 로딩 스크립팅하기

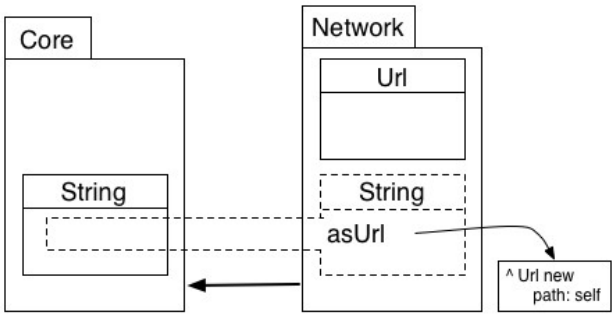
Pharo는 구문을 기반으로 한 병합, 트리 diff-merge, git-like 분산 버저닝 시스템과 같은 소스 코드를 관리하는 강력한 툴을 제시한다. 특히 Monticello 장에서 보인 바와 같이 Pharo는 Monticello라는 패키지 시스템을 사용한다. 이번 장에서는 Monticello의 주요 측면들을 상기시킨 후에 Gofer를 이용해 패키지를 스크립팅하는 방법을 보일 것이다. Gofer는 Monticello를 위한 단순한 API다. Gofer는 L. Renggli에 의해 개발되었고, 후에 E. Lorenzano와 C. Bruni에 의해 확장되었다. 이는 Metacello 장에서 소개한 패키지 맵을 관리하는 데에 사용되는 언어인 Metacello 에 의해 사용된다.

서문: 패키지 관리 시스템

패키지. 패키지는 클래스 및 메서드 정의의 리스트다. Pharo에서는 패키지가 네임스페이스와 연관되지 않는다. 패키지는 다른 패키지에 정의된 클래스를 확장할 수 있는데, 가령 Network 패키지에 String이 정의되어 있지 않더라도 Network 패키지는 String 클래스에 메서드를 추가할 수 있다는 말이다. 클래스 확장은 레이어의 정의를 지원하고, 패키지의 자연적 정의를 허용한다.

패키지를 정의하기 위해서는 Monticello 브라우저를 이용해 패키지 하나를 선언하고 클래스 확장을 정의하면 되는데, '*' 으로 시작해 패키지명이 따라오는 (본문에서는 '*network') 범주로 된 메서드를 정의하는 것으로 충분하다.

```
Object subclass: #ButtonsBar
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```



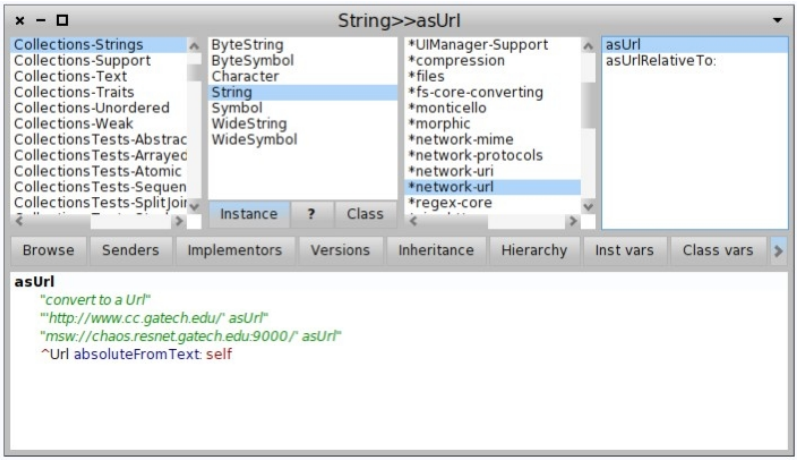


그림 8.1: 브라우저는 String 클래스가 network-url 패키지로부터 asUrl 과 asUrlRelativeTo: 메서드를 얻음을 보여준다.

```
category: 'Zork'
```

Monticello Browser에서 패키지를 선택하고 Changes 를 클릭하면 공개 전에 패키지의 변경 리스트를 얻을 수 있다.

패키지 버저닝 시스템. 버전 관리 시스템은 버전 저장을 도와주고 시스템 진화의 히스토리를 보관한다. 뿐만 아니라 소스 코드 저장소에 대한 동시 접근까지 관리한다. 저장된 모든 변경 내용의 흔적을 유지하고, 다른 기술자들과 협력하도록 해준다. 프로젝트가 성장할수록 버전 관리 시스템의 사용은 더 중요해진다.

Monticello는 Pharo의 버전 관리와 패키지 시스템을 정의한다. Pharo에서 클래스와 메서드는 액션이 실행될 때 (슈퍼클래스 변경, 인스턴스 변수 변경, 메서드의 추가, 변경, 삭제 등) Monticello에 의해 버저닝되는 기본 엔티티다. 소스는 HTTP 서버로서, Monticello가 관리하는 프로젝트를 (특히 패키지) 저장할 수 있도록 해준다. 이는 기부자들의 관리와 상태, 선명도

(visibility) 정보, RSS 피드와 함께 wiki를 제공한다. 모든 사람에게 공개되는 소스는 `http://www.smalltalkhub.com/` 이용하면 된다.

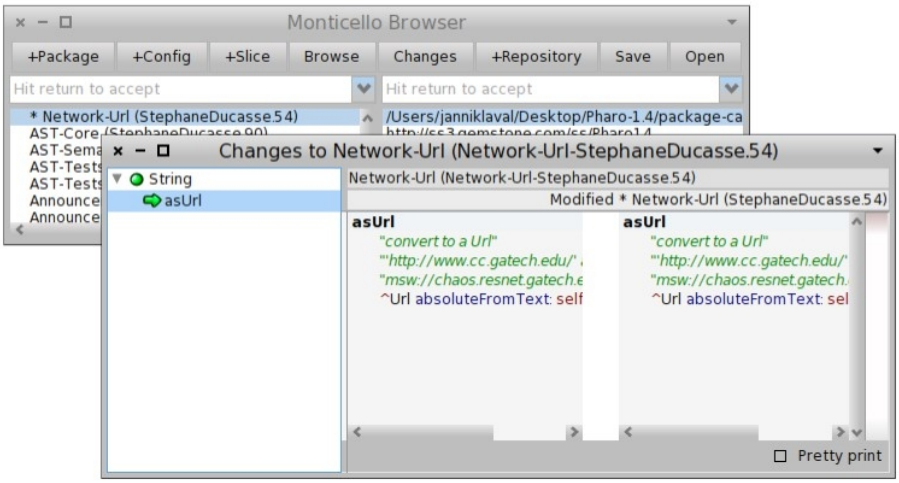


그림 8.2: change 브라우저를 통해 String≫asUrl 메서드가 변경되었음을 볼 수 있다.

Monticello의 분산 아키텍처. Monticello는 git처럼 분산 버전 제어 관리 시스템이지만 Smalltalk를 집중으로 한다. Monticello는 클래스, 메서드와 같은 소스 코드 엔티티를 조작한다. 그리고 로컬 및 분산형 코드 서버를 관리하는 것이 가능하다. Gofer는 그러한 서버들을 스크립팅하여 서버의 공개(publish), 다운로드, 동기화를 허용한다.

Monticello는 패키지에 대한 로컬 캐시를 사용한다. 패키지가 필요할 때마다 이러한 로컬 캐시에서 먼저 검색된다. 이와 유사한 방식으로, 패키지를 저장할 때는 로컬 캐시에도 저장된다. 물리적 관점에서 보면 Monticello 패키지는 패키지의 완전한 소스 코드와 메타 데이터를 포함하는 압축 파일이다. 좀 더 명확히 설명하기 위해 앞으로는 Pharo 이미지에 로딩된 패키지와 캐시에 저장되었으나 로딩되지 않은 패키지를 구별하겠다. 현재 로딩된 패키지를 패키지의 작업 사본(working copy)이라고 부른다. 그리고, 이미지(가상 머신이 실행하는 바이트코드와 객체), 로딩된 패키지(메모리에 로딩된 서버로부터 로딩된 패키지), 더러운(dirty) 패키지(저장되지 않은 수정내용이 있는 로딩된 패키지)로 정의하겠다. 더러운 패키지는 로딩된 패키지를 의미한다.

예를 들어, 그림 8.3에서 패키지 a.1 은 smalltalk.com 서버로부터 로딩되었으나 수정되지 않았다. 패키지 b.1 는 yoursource.com 서버로부터 로딩되었으나 이미지에 국부적(locally)으로 수정되었다. 더러운 패키지에 해당하는 b.1 이 yoursource.com 서버 상에 저장되면 캐시와 원격 서버에 저장된 b.2 로 버저닝된다.

Gofer란 무엇인가?

Gofer는 Monticello를 위한 스크립팅 툴이다. 이는 Lukas Renggli에 의해 개발되었고 Metacello에 의해 사용된다 (Monticello의 최상위에 빌드된 맵과 프로젝트 관리 시스템). Gofer는

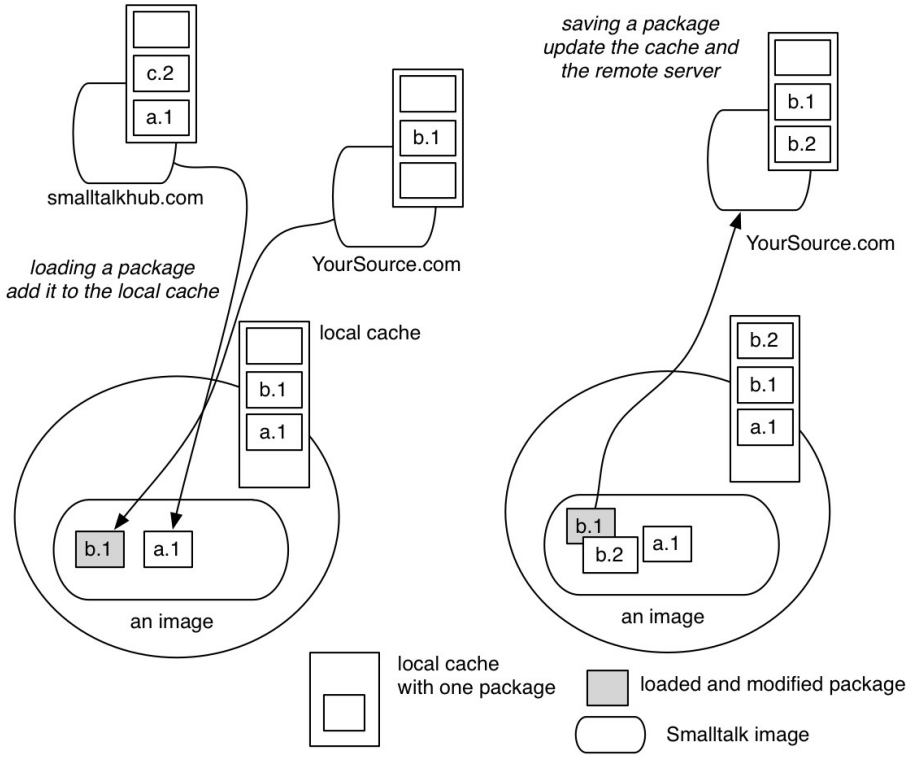


그림 8.3: (왼쪽) 로딩 및 캐시 저장된 깨끗한 패키지와 더러운 패키지의 일반적인 셋업 - (오른쪽) 공개된 패키지.

패키지의 로딩, 저장, 병합, 업데이트, 인출(fetch)을 위해 스크립트를 쉽게 생성하도록 지원한다. 뿐만 아니라 Gofer는 시스템이 확실히 깨끗한 상태로 유지되도록 한다. Gofer는 하나의 연산으로 여러 저장소에 위치한 패키지를 로딩하고, 최신 버전이나 최근 개발된 버전을 로딩하도록 해준다. Gofer는 Pharo의 기초에 해당하는데, Pharo 1.0 Metacello는 복잡한 프로젝트를 로딩하기 위해 Gofer를 기본 기반 구조로 사용하기 때문이다.

아래 표현식을 실행하여 Gofer에게 스스로를 업데이트하도록 요청할 수 있다.

```
Gofer gofer update
```

Gofer 사용하기

Gofer의 사용은 간단한데, 위치, 로딩할 패키지, 실행해야 할 동작을 명시하기만 하면 된다. 위치는 종종 파일 시스템을 표현하는데, HTTP, FTP 또는 단순히 하드 디스크가 되기도 한다. 위치는 Monticello 저장소로 접근하는 데 사용되는 것과 동일하다. 아래 표현식에서 사용되는 것처럼 `'http://smalltalkhub.com/mc/MyAccount/MyPackage/main'` 을 예로 들 수 있겠다.

```
MCHttpRepository
    location: 'http://smalltalkhub.com/mc/MyAccount/MyPackage/main'
    user: ''
    password: ''
```

전형적인 Gofer 스크립트를 소개하겠다. 이는 JennikLaval 계정의 http://www.smalltalkhub.com 에서 이용 가능한 PBE2GoferExample 저장소로부터 PBE2GoferExample 패키지를 로딩하고 싶다는 의미다.

```
Gofer new
    url: 'http://smalltalkhub.com/mc/PharoBooks/GoferExample/main';
    package: 'PBE2GoferExample';
    load.
```

저장소(HTTP 또는 FTP)가 식별(identification)을 필요로 한다면 url:username:password: 메시지를 이용할 수 있다. 이것은 하나의 메시지이므로 사이에 cascade를 넣지 않도록 주의하라. directory: 메시지는 로컬 파일로의 접근을 지원한다.

```
Gofer new
    url: 'http://smalltalkhub.com/mc/PharoBooks/GoferExample/main'
    username: 'pharoUser'
    password: 'pharoPwd';
    package: 'PBE2GoferExample';
    load.
```

```
"we work on the project PBE2GoferExample and provide credentials"
Gofer new
    url: 'http://smalltalkhub.com/mc/PharoBooks/GoferExample/main/PBE2GoferExample'
    username: 'pharoUser'
    password: 'pharoPwd';
    package: 'PBE2GoferExample';        "define the package to be loaded"
    disablePackageCache;                "disable package lookup in local cache"
    disableRepositoryErrors;            "stop the error raising"
    load.                               "load the package"
```

동일한 public 서버가 사용되는 것이 보통이기 때문에 Gofer의 API는 스크립트를 단축하기 위해 많은 단축키를 제공한다. 스크립트를 작성해 다른 사람들이 우리 코드를 로딩할 수 있도록 제공하길 원할 때가 있는데, 이런 경우 비밀번호를 명시해야 하는 것은 바람직하지 못하다. smalltalkHub의 예로, GoferExample 프로젝트에 대해 'http://smalltalkhub.com/mc/PharoBooks/GoferExample/main' 와 같은 긴 urls를 들어보자. smalltalkHubUser:project: 메시지를 이용해 최소한의 정보만 명시하겠다. 이번 장에서는 http://ss3.gemtalksystems.com/ss 에 대한 단축키로 squeaksource3: 를 이용하겠다.

```
"Specifying a user but no password"
Gofer new
    smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
    package: 'PBE2GoferExample';
    load
```

게다가 명시된 URL에 패키지를 Gofer가 성공적으로 로딩하지 못하면 보통 자신의 이미지의 루트에 위치한 로컬 캐시를 살펴본다. disablePackageCache를 이용해 Gofer가 캐시를

사용하지 못하도록 강요하거나, enablePackageCache 메시지를 이용해 캐시를 사용하도록 강요할 수 있다.

비슷한 방식으로 Gofer는 저장소들 중 하나로 접근할 수 없을 때에 오류를 리턴한다. 우리는 이에 disableRepositoryErrors 메시지를 이용해 그러한 오류를 무시하도록 지시했다. 반대로 활성화하는 메시지는 enableRepositoryErrors를 이용할 수 있겠다.

패키지 식별

URL과 옵션이 명시되고 나면 로딩하고자 하는 패키지를 정의해야 한다. version: 메시지를 사용하면 확실히 로딩해야 할 버전을 정의하는 반면, package: 메시지는 모든 저장소에서 이용 가능한 최신 버전을 로딩하는 데 사용해야 한다.

아래 예제는 패키지의 버전 2를 로딩한다.

```
Gofer new
    smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
    version: 'PBE2GoferExample-janniklaval.1';
    load
```

블록을 전달하도록 package: aString constraint: aBlock 메시지를 이용해 패키지를 식별하기 위한 몇 가지 제약을 명시하는 것도 가능하다.

예를 들어, 아래 코드는 janniklaval 이라는 개발자가 저장한 패키지의 최신 버전을 로딩할 것이다.

```
Gofer new
    smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
    package: 'PBE2GoferExample'
    constraint: [ :version | version author = 'janniklaval' ];
    load
```

Gofer 액션

여러 개의 패키지 로딩하기

우리는 여러 개의 서버로부터 여러 개의 패키지를 로딩할 수 있다. 구체적인 예를 들기 위해 OSProcess의 Metacello 설정을 이용해 먼저 로딩해야겠다.

```
Gofer new
    "we will load a version of the configuration of OSProcess "
    url: 'http://www.squeaksource.com/MetacelloRepository';
    package: 'ConfigurationOfOSProcess';
    load.

"Now to load OSProcess you need to ask for its configuration."
((Smalltalk at: #ConfigurationOfOSProcess) project version: #stable) load.
```

아래 코드 조각은 여러 서버로부터 다수의 패키지를 로딩한다. 로딩 순서는 언제나 그렇듯 스크립트가 Collections-Arithmetic을 먼저 로딩한 후 Sound, 마지막으로 Scratch 비주얼 프로그래밍 언어의 포트인 Phratch를 로딩한다.

Phratch 애플리케이션을 완전히 로딩하기 때문에 시간이 어느 정도 소요될 수 있다는 점을 명심하라.

```
Gofer new
    url: 'http://smalltalkhub.com/mc/PharoExtras/CollectionArithmetic/main';
    package: 'Collections-Arithmetic';
    url: 'http://smalltalkhub.com/mc/PharoExtras/Sound/main';
    package: 'Sound';
    package: 'Settings-Sound';
    package: 'SoundScores';
    package: 'SoundMorphicUserInterface';
    url: 'http://smalltalkhub.com/mc/JLaval/Phratch/main';
    package: 'Phratch';
    load
```

이 예제를 살펴보면 Collections-Arithmetic이 smalltalkhub 서버의 CollectionArithmetic 저장소에서 검색되고, Phratch가 smalltalkhub 서버의 Phratch 프로젝트에서 검색된다는 인상을 받을 수도 있다. 하지만 이는 사실이 아니며, Gofer는 이 순서를 고려하지 않는다. 버전 번호가 없을 시 Gofer는 두 개의 서버를 살펴보고 발견되는 가장 최근의 패키지 버전을 로딩한다.

따라서 스크립트를 아래와 같이 다시 작성할 수 있다.

```
Gofer new
    url: 'http://smalltalkhub.com/mc/PharoExtras/CollectionArithmetic/main';
    url: 'http://smalltalkhub.com/mc/PharoExtras/Sound/main';
    url: 'http://smalltalkhub.com/mc/JLaval/Phratch/main';
    package: 'Collections-Arithmetic';
    package: 'Sound';
    package: 'Settings-Sound';
    package: 'SoundScores';
    package: 'SoundMorphicUserInterface';
    package: 'Phratch';
    load
```

특정 서버로부터 패키지를 로딩해야 함을 명시하고 싶다면 다수의 스크립트를 작성해야 한다.

```
Gofer new
    url: 'http://smalltalkhub.com/mc/PharoExtras/CollectionArithmetic/main';
    package: 'Collections-Arithmetic';
    load.
Gofer new
    url: 'http://smalltalkhub.com/mc/PharoExtras/Sound/main';
    package: 'Sound';
    package: 'Settings-Sound';
    package: 'SoundScores';
    package: 'SoundMorphicUserInterface';
    load.
Gofer new
    url: 'http://smalltalkhub.com/mc/JLaval/Phratch/main';
    package: 'Phratch';
    load
```

그러한 스크립트는 패키지의 최신 버전을 로딩하므로, 새 패키지 버전이 공개될 경우 부적절하더라도 로딩되기 때문에 사실상 취약하다는 점을 주목하라. 대개는 우리가 의존하는 외부 구성요소의 버전을 제어하고 현재 개발에 최신 버전을 이용하는 것이 훌륭한 실습이 되겠다.

이제 그러한 문제는 설정을 표현하고 로딩하는 툴, Metacello로 해결할 수 있겠다.

기타 프로토콜

Gofer는 로컬 디렉터리로부터 로딩을 비롯해 FTP 또한 지원한다. 몇 가지만 변경하여 앞과 동일한 메시지를 사용하겠다.

FTP에서는 'ftp'를 헤딩으로 이용해 URL을 명시해야 한다.

```
Gofer new
    url: 'ftp://wtf-is-ftp.com/code';
    ...
```

로컬 디렉터리를 작업하기 위해서는 directory: 메시지 다음에 디렉터리의 절대 경로를 사용해야 한다. 사용할 디렉터리는 /home/pharoer/hacking/MCPackages에서 찾을 수 있음을 명시한다.

```
Gofer new
    directory: '/home/pharoer/hacking/MCPackages';
```

마지막으로 별표 기호를 (keen star가 오역 같아 kleene star로 번역하였습니다.) 이용해 저장소와 그 모든 하위 폴더에서 패키지를 검색할 수 있다.

```
Gofer new
    directory: '/home/pharoer/hacking/MCPackages/*';
    ...
```

Gofer 인스턴스가 매개변수화되면 여러 액션을 실행하도록 메시지를 전송할 수 있는데, 가능한 액션 목록을 아래 제공하며, 그 중 일부는 추후 설명하겠다.

load	명시된 패키지를 로딩한다.
update	로딩된 패키지 버전을 업데이트한다.
merge	떨어져 있는 버전과 현재 로딩된 버전을 병합한다.
localChanges	bases 버전과 현재 수정된 버전 간 변경 내용의 목록을 표시한다.
remoteChanges	현재 수정된 버전과 서버에 공개된 버전 간 변경 내용을 표시한다.
cleanup	Cleanup 패키지 : 시스템의 오래된 정보가 제거된다.
commit / commit:	떨어진 서버에 패키지를 저장한다 – 메시지 로그 이용.
revert	이전에 로딩된 패키지를 재로딩한다.
recompile	패키지를 재컴파일한다.
unload	이미지로부터 패키지를 언로딩한다.
fetch	원격 서버로부터 원격 패키지 버전을 로컬 캐시로 다운로드한다.
push	로컬 캐시로부터 원격 서버로 버전을 업로드한다.

원격 서버 작업하기

Monticello는 분산형 버저닝 제어 시스템이기 때문에 때때로 원격 서버에 공개된 버전들과 MC 로컬 캐시에 국부적으로 공개된 버전들을 동기화하는 것이 유용하다. 여기서는 그러한 업무를 지원하는 주요 연산을 소개하겠다.

병합, 업데이트, 복구 연산. merge 메시지는 원격 버전과 작업 사본(현재 로딩된) 간 병합을 실행한다. 작업 사본에 일어나는 변경은 원격 버전의 코드와 병합된다. 병합 후 작업 사본이 더러워지고, 재공개(republish)되어야 하는 것이 보통이다. 새 버전은 현재 변경 내용과 원격 버전의 변경 내용을 포함할 것이다. 충돌이 발생할 경우 사용자는 경고를 수신하고, 그렇지 않은 경우 연산은 조용히 발생할 것이다.

```
Gofer new
    smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
    package: 'PBE2GoferExample';
    merge
```

update 메시지는 이미지에 원격 버전을 로딩한다. 작업 사본의 수정내용은 손실된다.

revert 메시지는 로컬 버전을 리셋하는데, 현재 버전을 다시 로딩하는 것을 예로 들 수 있다. 이후 작업 사본의 변경 내용은 손실된다.

commit과 commit: 연산. 패키지를 병합하거나 변경했다면 이를 저장하길 원할 것이다. 이러한 작업을 위해 commit과 commit: 메시지를 이용할 수 있다. 후자는 주석을 필요로 하는데, 훌륭한 실습에 해당한다.

```
Gofer new
    "We save the package in the repository"
    smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
    package: 'PBE2GoferExample';
    "We comments the changes and save"
    commit: 'I try to use the message commit: '
```

localChanges 와 remoteChanges 연산. 버전을 로딩하거나 저장하기 전에 국부적으로 혹은 서버 상에 일어난 변경 내용을 검증하는 것이 유용하게 작용한다. localChanges 메시지는 마지막으로 로딩된 버전과 작업 사본 간 변경 내용을 표시한다. remoteChanges는 작업 사본과 서버에 마지막으로 공개된 버전의 차이를 보여준다. 두 가지 모두 변경 내용의 목록을 리턴한다.

```
Gofer new
    smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
    package: 'PBE2GoferExample';
    "We check that we will publish only our changes by comparing local changes versus
        the packages published on the server"
    localChanges
```

browseLocalChanges와 browserRemoteChanges 메시지를 이용하면 일반적인 코드 브라우저를 이용해 변경 내용을 살펴보는 것이 가능하다.

```
Gofer new
    smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
    "we add the latest version of PBE2GoferExample"
    package: 'PBE2GoferExample';
    "we browse the latest version published on the server"
    browseRemoteChanges
```

unload 연산. unload 메시지는 이미지로부터 패키지를 언로딩한다. Monticello 브라우저를 이용하면 패키지를 삭제할 수는 있지만 그러한 연산이 패키지와 연관된 클래스의 코드까지 제거하지는 않으며 사실상 패키지를 파괴할 뿐이라는 사실을 주목해야 한다. 패키지를 언로딩하면 패키지와 그것이 포함하는 클래스를 파괴한다.

아래 코드는 패키지와 그 클래스를 현재 이미지에서 언로딩한다.

```
Gofer new
    package: 'PBE2GoferExample';
    unload
```

이런 방식으로는 Gofer를 언로딩할 수 없음을 주목하라. Gofer gofer unload는 효과가 없다.

fetch 와 push 연산. Monticello는 분산형 버저닝 시스템이기 때문에 원격 서버에 강제로 공개하지 않고 자신이 원하는 버전을 모두 국부적으로 저장하는 편이 나으며, 특히 오프라인으로 작업 시 더 그러하다. 이제 모든 로컬 및 원격으로 공개된 패키지를 동기화하는 작업은 지루하다. fetch와 push 메시지를 이용하면 될 일이다.

fetch 메시지는 자신의 로컬 서버에 누락된 패키지를 원격 서버로부터 복사한다. 패키지는 Pharo에 로딩되지 않는다. 인출(fetch)된 이후 원격 서버가 붕괴(break down)하더라도 패키지를 로딩할 수 있다.

```
Gofer new
    smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
    package: 'PBE2GoferExample';
    fetch
```

이제 자신의 패키지를 국부적으로 로딩하고 싶다면 검색(lookup)은 로컬 캐시를 고려하고 본 장의 처음에 제시된 바와 같이 오류를 비활성화 시키도록 셋업해야 함을 명심한다 (disableRepositoryErrors와 enablePackageCache 메시지 참고).

push 메시지는 inverse 연산을 실행한다. 국부적으로 이용 가능한 패키지를 원격 서버로 공개한다. 당신이 국부적으로 공개한 모든 패키지는 이후 서버로 삽입된다(pushed).

```
Gofer new
    smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
    package: 'PBE2GoferExample';
    push
```

습관상 우리는 사용한 프로젝트나 프로젝트의 모든 버전에 대한 사본을 항상 로컬 캐시에 보관한다. 이 방법을 이용하면 어떤 네트워크 실패가 발생하든 그로부터 자율적일 수 있고, 패키지는 정기적 백업으로 저장된다.

두 개의 메시지를 이용하면 로컬 및 원격 저장소를 동기화하는 sync 스크립트를 작성하기가 쉽다.

```
Gofer new
    smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
    package: 'PBE2GoferExample';
    push.
Gofer new
    smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
    package: 'PBE2GoferExample';
    fetch
```

물론 앞에서 언급했듯이 다수의 패키지를 삽입하고 인출할 수 있다.

```
Gofer new
    smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
    package: 'PBE2GoferExample';
    package: 'PBE2GoferExampleSecondPackage';
    push.
Gofer new
    smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
    package: 'PBE2GoferExample';
    package: 'PBE2GoferExampleSecondPackage';
    fetch
```

응답 자동화하기

때로는 패키지 설치 시 비밀번호와 같은 정보를 요청한다. 빌드 서버를 체계적으로 이용할 경우 패키지가 그러한 정보의 요청을 중단할지도 모르지만 이러한 질문에 대해 스크립트 내에서 어떠한 응답을 제공하는지 아는 것이 중요하다. 이러한 작업을 지원하는 메시지로 valueSupplyingAnswers: 를 들 수 있다.

```
[ Gofer new
    squeaksource: 'Seaside30';
    package: 'LoadOrderTests';
    load ]
    valueSupplyingAnswers: {
        {'Load Seaside'. True}.
        {'SqueakSource User Name'. 'pharoUser'}.
        {'SqueakSource Password'. 'pharoPwd'}.
        {'Run tests'. false}.
    }
```

해당 메시지는 앞에 소개한 예제들이 보여주듯이 블록으로 전송되면 질문과 응답의 목록을 제공한다.

설정 로딩

Gofer는 Metacello 설정 로딩을 지원하기도 한다. 이는 configurationOf:, loadVersion:, loadDevelopment, loadStable과 같은 메시지 집합을 제공하여 설정을 처리하도록 한다.

NativeBoost의 개발 버전을 로딩하는 예제를 들어보자. 여기서 당신이 해야 할 일은 NativeBoost 프로젝트를 명시하고, ConfigurationofNativeBoost를 로딩하여 개발 버전을 실행하기만 하면 된다.

```
Gofer new
    smalltalkhubUser: 'Pharo' project: 'NativeBoost';
    configuration;
    loadDevelopment
```

저장소 이름이 설정의 이름과 일치하지 않으면 configurationOf: 를 이용해 configuration 클래스명을 제공해야 한다.

몇 가지 유용한 스크립트

Gofer는 allResolved 메시지를 통해 주어진 저장소 내 모든 패키지를 얻을 수 있는 훌륭한 기능을 제공한다.

스크립트 8.1: 저장소 내 패키지 개수 얻기.

```
(Gofer new
    smalltalkhubUser: 'Pharo' project: 'NativeBoost';
    allResolved) size
```

아래 스크립트는 패키지 버전을 패키지별로 그룹화하여 dictionary를 리턴한다. 가장 많이 사용되는 패키지를 이해한다면 도움이 될 것이다.

스크립트 8.2: 버전을 패키지명에 따라 그룹화하기.

```
(((Gofer new
    smalltalkhubUser: 'Pharo' project: 'NativeBoost';
    allResolved)
    groupedBy: [ :each | each packageName])
```

스크립트 8.3: SS3 상의 Kozen 프로젝트에 관한 패키지 리스트 얻기.

```
((Gofer new
    squeaksource3: 'Kozen';
    allResolved)
    groupedBy: [ :each | each packageName]) keys
```

패키지 인출하기

주어진 저장소의 모든 패키지를 인출하는 스크립트를 소개하겠다. 이는 모든 파일을 잡고 버전을 국부적으로 얻는 데에 유용하다.

스크립트 8.4: 저장소의 모든 패키지를 인출하기 (Pharo로부터)

```
| go |
go := Gofer new squeaksource3: 'Pharo20'.
go allResolved
    do: [ :each |
    self crLog: each packageName.
    go package: each packageName;
        fetch]
```

스크립트 8.5: Pharo 2.0 저장소로부터 모든 재팩토링 패키지를 인출하기.

```
| go |
go := Gofer new.
go squeaksource3: 'Pharo20'.
(go allResolved select: [ :each | 'Refactoring*' match: each packageName])
    do: [ :pack |
        self crLog: pack packageName.
        go package: pack packageName; fetch]
```

로컬 파일 공개하기

아래의 스크립트는 자신의 로컬 캐시로부터 주어진 저장소로 파일을 공개한다.

스크립트 8.6: Pharo 1.4를 이용해 패키지 파일을 새 저장소로 공개하는 방법.

```
| go |
go := Gofer new.
go repository: (MCHttpRepository
    location: 'http://ss3.gemtalksystems.com/ss/Pharo14'
    user: 'pharoUser'
    password: 'pharoPwd').

((FileSystem workingDirectory / 'package-cache')
    allEntries
        select: [ :each | '*.mcz' match: each])
            do: [ :f | go version: ('.' join: (f findTokens: $.) allButLast); push]
```

다음 스크립트는 새 파일시스템 라이브러리를 이용하며, 버전이 아니라 패키지명을 어떻게 얻는지도 보여주고자 한다. 스크립트는 mcz 파일을 공개하는 데에도 집중한다. 특정 패키지를 선택적으로 공개하도록 확장할 수도 있다.

스크립트 8.7: Pharo 20을 이용해 패키지 파일을 새 저장소에 공개하는 방법. (Pharo의 버전을 확인하기 바랍니다.)

```
| go |
go := Gofer new.
go repository: (MCHttpRepository
    location: 'http://ss3.gemtalksystems.com/ss/rb-pharo'
    user: 'pharoUser'
    password: 'pharoPwd').

(((FileSystem disk workingDirectory / 'package-cache')
    allFiles select: [:each | '*.mcz' match: each basename])
        groupedBy: [:each | (each base copyUpToLast: $-) ])
            keys do: [:name | go package: name; push]
```

스크립트 8.8: SmalltalkHub 상의 Moose Team 로 Fame을 공개하는 방법.

```
|go repo|
repo := MCSmalltalkhubRepository
    owner: 'Moose'
    project: 'Fame'
    user: 'pharoUser'
    password: 'pharoPwd'.

go := Gofer new.
go repository: repo.
(((FileSystem disk workingDirectory / 'package-cache')
    allFiles select: [:each | '*Fame*.mcz' match: each basename])
        groupedBy: [:each | (each base copyUpToLast: $-) ]) keys
            do: [:name | go package: name; push]
```

모든 것을 한 페이지에

많은 생각이 필요 없는 간단한 스크립트를 선호하므로 Monticello 저장소들 간 이동하는 전체 버전을 소개하겠다.

스크립트 8.9: Monticello의 어떤 저장소에서 다른 저장소로 동기화하기 위한 스크립트

```
| source goferSource destination goferDestination files destinationFiles |

source := MCHttpRepository location: 'http://www.squeaksource.com'.
destination := MCSmalltalkhubRepository
    owner: 'TheOwner'
    project: 'YourPackage'
    user: 'YourName'
    password: ''.

goferSource := Gofer new repository: source.
goferDestination := Gofer new repository: destination.

files := source allVersionNames.
"select the relevant mcz packages"
goferSource allResolved
    select: [ :resolved | files anySatisfy: [ :each | resolved name = each ] ]
    thenDo: [ :each | goferSource package: each packageName ].
"download all mcz on your computer"
goferSource fetch.

"check what files are already at destination"
destinationFiles := destination allVersionNames.
"select only the mcz that are not yet at destination"
files
    reject: [ :file | destinationFiles includes: file ]
    thenDo: [ :file | goferDestination version: file ].
"send everything to SmalltalkHub"
goferDestination push.

"check if we have exactly the same files at source and destination"
self assert: destination allVersionNames sorted equals: files sorted.
```

요약

Gofer는 패키지의 관리를 스크립팅하기 위한 강력하고도 안정적인 구현을 제공한다. 프로젝트 크기가 증가할수록 Metacello의 사용을 고려해야 할 것이다 (제 9장 참고).

이번 장에서는 Gofer로 패키지를 스크립팅하는 방법을 소개하였다.

- load 메서드는 url: 와 package: 메서드와 함께 주어진 소스로부터 패키지를 로딩하도록 해준다.
- url: 메서드는 FTP와 로컬 디렉터리 접근을 지원한다.
- API는 몇 가지 유용한 단축키를 제공한다. smalltalkhubUser:project: 는 `http://www.smalltalkhub.com` , squeaksource3: 는 `http://ss3.gemtalksystems.com/ss/` , gemsource: 는 `http://seaside.gemstone.com/ss/` 의 단축키에 해당한다.
- load를 호출하기 전에 package: 메서드를 여러 번 호출하여 여러 개의 패키지를 로딩할 수 있다.
- Gofer 인스턴스가 매개변수화되고 나면 update, merge, push 등의 많은 관련된 연산을 실행할 수 있다.

제 9 장

Metacello를 이용해 프로젝트 관리하기

Dale Henrichs *(dale.henrichs@gemstone.com)* 작성. 그리고
textit **Mariano Martinez Peck** (marianopeck@gmail.com) 참여

프로젝트를 로딩하려는데 없어진 줄도 몰랐던 패키지 때문에 오류가 발생했던 적이 있는가? 아니면 그보다 더 심각하게, 패키지가 있지만 올바르지 않은 버전이었던 경우가 있는가? 개발자들이 당신과 다른 컨텍스트에서 작업하는데 개발자들에게는 프로젝트가 양호하게 로딩되다가 자신에게만 이러한 상황이 발생하는 경우도 흔하다.

이러한 문제에 대한 프로젝트 개발자들의 해결책은 프로젝트를 구성하는 패키지들 간 의존성을 명시적으로 관리하도록 패키지 의존성 관리 시스템을 사용하는 것을 들 수 있다. 본 장은 Metacello, 즉 Pharo의 패키지 관리 시스템을 이용하는 방법과 이를 사용 시 이점에 대해 소개하겠다.

서론

Metacello는 Monticello용 패키지 관리 시스템이라고 말을 한다. 하지만 이게 정확히 어떤 의미일까? 패키지 관리 시스템은 소프트웨어 패키지 그룹을 설치, 업그레이드, 구성, 제거하는 과정을 자동화하는 툴의 집합체이다. Metacello는 패키지를 그룹화하여 사용자를 위해 작업을 간소화하고 의존성을 관리하여 가령 어떤 구성요소의 어떤 버전을 로딩해야 패키지 전체 집합이 일관될 수 있는지 관리한다.

패키지 관리 시스템은 패키지를 설치하는 일관된 방식을 제공한다. 뿐만 아니라 패키지 관리 시스템은 가끔 인스톨러(installer)로 불리곤 하는데, 이는 부정확하다. 오해를 야기할 수 있는데, 패키지 관리 시스템은 소프트웨어의 설치에 그치지 않고 설치 이상으로 많은 일을 실행하기 때문이다. 패키지 관리 시스템은 Envy (VisualAge Smalltalk), Maven (Java), apt-get/aptitude(Debian과 Ubuntu)와 같은 다른 컨텍스트에서 사용해보았을 것이다.

패키지 관리 시스템의 주요 특징 중 하나로, 어떤 패키지도 올바르게 로딩한다는 점을 들 수 있는데, 어떤 것도 수동으로 설치할 필요가 없다. 이것이 가능하려면 각 의존성, 의존성들의 의존성 등이 패키지의 설명에 명시되어야 하고, 이러한 설명은 패키지 관리 툴이 그들을 올바른 순서로 로딩시키기에 충분한 정보를 포함해야 한다.

Metacello의 장점을 보여주는 예로 PharoCore 이미지를 들 수 있는데, 의존성의 문제 없이 어떤 프로젝트의 어떤 패키지든 로딩할 수 있다. 물론 Metacello는 마법(magic)을 사용하지 않으며, 마법은 패키지 개발자들이 의존성을 올바로 정의하는 경우에 한에서만 효과가 있다.

각 작업마다 하나의 툴

Pharo는 소프트웨어 패키지를 관리하는 세 개의 툴을 제공하는데, 모두 밀접하게 연관되어 있으나 각각이 고유의 목적을 지닌다. 툴은 소스 코드의 버전을 관리하는 Monticello, Monticello에 대한 스크립팅 인터페이스 Gofer, 패키지 관리 시스템인 Metacello로 구성된다.

Monticello: 소스 코드 버저닝. 소스 코드 버저닝은 유일한 버전을 특정 소프트웨어 상태로 할당하는 과정이다. 이는 새 버전을 커밋(commit)하고, 타인이 커밋한 버전으로 업데이트하며, 버전들 간 차이를 살펴보고, 오래된 버전으로 복구(revert)하는 작업 등을 실행할 수 있도록 해준다.

Pharo는 Monticello 패키지를 관리하는 Monticello 소스 코드 버저닝 시스템을 이용한다. Monticello는 각 패키지마다 위의 연산을 모두 실행하도록 해주지만 패키지들 간 의존성을 쉽게 명시하거나, 패키지의 안정된 버전을 식별하거나, 패키지를 의미 있는 단위로 그룹화하는 방식은 Monticello에서 제공하지 않는다. 이와 관련해 제 7장에서 설명한 바가 있다.

Gofer: Monticello의 스크립팅 인터페이스. Gofer는 Monticello에 포함된 작은 툴로서, Monticello 패키지의 그룹을 로딩, 업데이트, 병합, 구별, 복구, 커밋, 재컴파일, 언로딩한다. Gofer는 또 이러한 연산들이 가능한 한 깔끔하게 실행되도록 보장한다. 상세한 정보는 제 8장을 참고한다.

Metacello: 패키지 관리. Metacello는 프로젝트의 개념을 연관된 Monticello 패키지의 집합으로 소개한다. 이는 프로젝트, 프로젝트의 의존성, 메타데이터를 관리하는 데 사용된다. Metacello는 패키지들 간 의존성을 관리하기도 한다.

Metacello 특징

Metacello는 Monticello의 중요한 기능과 일치하며, 아래와 같은 개념을 바탕으로 한다.

선언적 프로젝트 설명. Metacello 프로젝트는 Monticello 패키지의 리스트로 구성된 버전을 versions으로 명명하였다. 의존성은 필요한 프로젝트의 명명된 버전과 관련해 명시적으로 표현된다. 필요한 프로젝트라 함은 또 다른 Metacello 프로젝트에 대한 참조다. 총괄적으로 이러한 설명을 모두 프로젝트 메타데이터라고 부른다.

프로젝트 메타데이터는 버저닝된다. Metacello 프로젝트 메타데이터는 클래스 내 인스턴스 메서드로서 표현된다. 그러한 메타데이터를 코드로서 관리하면 대부분 패키지 관리 시스템이 사용하는 XML과 비교 시 엄청난 장점을(power) 가져온다. Metacello 프로젝트 메타데이터 자체가 Monticello 패키지로서 보관 가능하고, 그에 따라 버전 제어의 대상이 된다. 그 결과 프

로젝트 메타데이터로의 동시적 업데이트를 쉽게 관리할 수 있는데, 코드 베이스의 병렬 버전과 마찬가지로 메타데이터의 병렬 버전(parallel version)도 병합이 가능하기 때문이다.

Metacello의 특징은 다음과 같다.
크로스 플랫폼: Metacello는 Monticello를 지원하는 모든 플랫폼에서 실행되고, 현재로선 Pharo, Squeak, GLASS가 해당된다.
조건적 패키지 로딩: 프로젝트를 다수의 플랫폼에서 실행시키기 위해 Metacello 는 플랫폼 특정적 Monticello 패키지의 조건적 로딩을 지원한다.
설정(configurations): Metacello는 프로젝트의 설정을 관리한다. 대규모 프로젝트에는 각 플랫폼에 요구되는 여러 프로젝트와 패키지 집합으로 된 다수의 variations가 있는 것이 보통이다. 유일한 설정마다 버전 문자열이 표시된다.

Metacello는 두 가지 유형의 엔티티, baselines 와 versions 의 정의를 (메서드로서 표현) 지원한다.
baseline. baseline은 프로젝트의 기본 구조를 정의한다. baseline은 프로젝트를 구성하는 데 요구되는 프로젝트와 패키지를 열거한다. baseline은 패키지가 로딩되어야 하는 순서를 정의하고 패키지가 로딩되는 저장소를 정의한다.

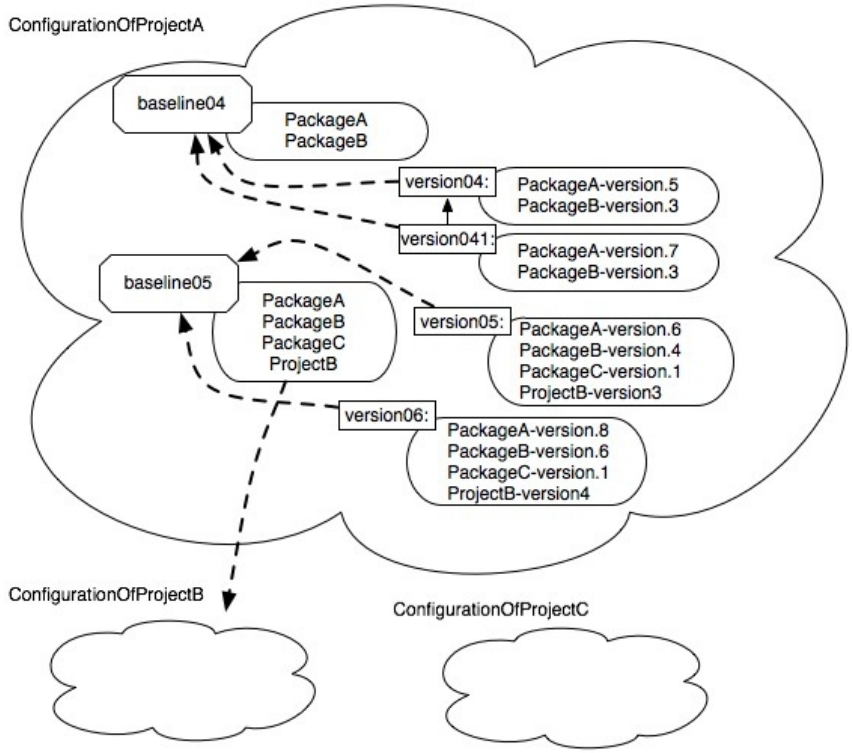


그림 9.1: 설정 : 의존성과 함께 한 버전과 baseline의 그룹.

Versions(버전). 버전은 로딩해야 하는 각 프로젝트와 패키지의 정확한 버전을 식별한다. 버전은 baseline 버전을 기반으로 한다. baseline 버전 내 각 패키지마다 Monticello 파일명(예: Metacello-Base-dkh.152)이 명시된다. baseline 버전 내 각 프로젝트마다 Metacello 버전 번호가 명시된다.

그림 9.1의 ConfigurationOfProjecta 는 2개의 baseline(baseline 0.4와 0.5)과 4개의 버전(버전 0.4, 0.4.1, 0.5, 0.6)을 포함한다. 버전 0.4는 baseline 0.4를 기반으로 하며, 각 패키지마다 버전을 명시한다 (PacageAversion.5과 PackageB-version.3). 버전 0.4.1 또한 baseline 0.4를 기반으로 하지만 PackageA에 대한 다른 버전을 명시한다 (Package-version.7).

Baseline 0.5는 3개의 패키지(PackageA, PackageB, PackageC)로 구성되며, 외부 프로젝트(ProjectB)에 의존한다. 새 패키지(PackageC)와 프로젝트 의존성(ProjectB)이 프로젝트에 추가되어 새 구조를 반영하는 새 baseline 버전이 생성되어야 한다. 버전 0.5는 baseline 0.5를 기반으로 하며, 패키지의 버전(PackageA-version.6, PackageB-version.4, PackageC-version.1)과 의존적 프로젝트의 버전(ProjectB-version3)을 명시한다.

간단한 사례 연구

이 예제에서는 버전으로만 표현된 간단한 구성으로 시작해 점차적으로 baseline을 추가하고자 한다. 실제로는 baseline을 정의한 다음에 버전을 정의하는 편이 낫다.

그렇다면 CoolBrowser라고 불리는 소프트웨어 프로젝트를 관리하는 데에 Metacello를 이용한다고 치자. 첫 번째 단계는 MetacelloConfigTemplate 클래스를 복사한 후 클래스명을 오른쪽 마우스로 클릭하여 copy 를 선택하거나 Monticello 브라우저의 +Config 을 이용해 ConfigurationOfCoolBrowser로 명명함으로써 프로젝트에 대한 Metacello 설정을 생성하는 것이다 (제 7장). 설정(configuration)은 프로젝트에 현재 이용할 수 있는 설정(baseline과 버전 집합)을 설명하는 클래스로서 앞에서는 메타데이터라고 불렀던 것에 해당한다. 설정은 프로젝트의 여러 버전을 표현하여 Pharo의 다른 환경이나 버전에서 프로젝트를 로딩할 수 있도록 한다. 관습상 Metacello 설정의 이름은 ConfigurationOf를 앞에 붙여 구성된다.

클래스 정의는 다음과 같다.

```
Object subclass: #ConfigurationOfCoolBrowser
    instanceVariableNames: 'project'
    classVariableNames: 'LastVersionLoad'
    poolDictionaries: ''
    category: 'Metacello-MC-Model'
```

ConfigurationOfCoolBrowser에 인스턴스 측과 클래스 측 메서드가 있음을 눈치챌 것인데 이들이 어떻게 사용되는지는 후에 살펴보도록 하겠다. 이 클래스는 Object로부터 상속된다는 사실도 주목하라. Metacello 설정은 Metacello 자체를 포함해 어떤 전제조건도 없이 로딩이 가능하므로 Metacello 설정은 공통 슈퍼클래스에 의존할 수 없다.

이제 CoolBrowser 프로젝트에 여러 버전, 즉 1.0, 1.0.1, 1.4, 1.67과 같은 버전이 있다고 가정해보자. Metacello를 이용해 설정(configuration) 메서드, 각 프로젝트 버전의 내용을 설명하는 인스턴스 측 메서드를 생성한다. 아래와 같이 메서드에 <version:> pragma로 주석이 달려있는 한 버전 메서드에 대한 메서드명은 중요하지 않다. 하지만 버전 메서드가 versionXXX: 로 명명되는 관습도 있는데, 여기서 XXX는 틀린 문자(예: '.')가 없는 버전 번호를 의미한다.

CoolBrowser가 CoolBrowser-Core와 CoolBrowser-Tests라는 두 개의 패키지를 포함한다고 치자 (그림 9.2). 설정 메서드(여기서는 버전)는 아마 다음과 같은 모습일 것이다.

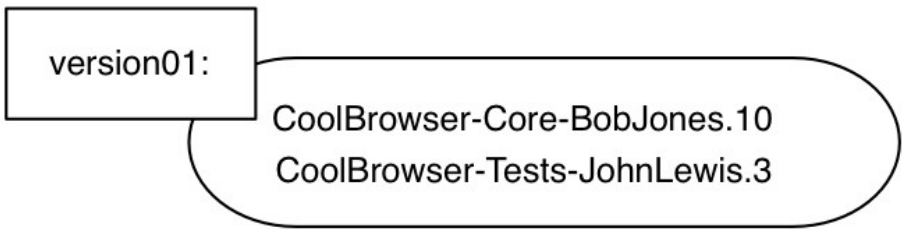


그림 9.2: 간단한 버전.

```
ConfigurationOfCoolBrowser>>version01: spec
    <version: '0.1'>

    spec for: #common do: [
    spec blessing: #release.
        spec repository: 'http://www.example.com/CoolBrowser'.
        spec
            package: 'CoolBrowser-Core' with: 'CoolBrowser-Core-BobJones.10';
            package: 'CoolBrowser-Tests' with: 'CoolBrowser-Tests-JohnLewis.3' ]
```

version01:spec 메서드는 spec 객체에 프로젝트의 0.1 버전에 대한 설명을 빌드한다. 버전 0.1에 대한 공통 코드는 (for:do: 메시지를 이용해 명시) CoolBrowser-Core와 CoolBrowser-Tests로 명명된 특정 패키지의 버전으로 구성된다. 이들은 package:packageName with: versionName 메시지를 이용해 명시된다. 이러한 버전들은 `http://www.example.com/CoolBrowser` Monticello 저장소에서 이용 가능하며, repository: 메시지를 이용해 명시된다. blessing: 메서드는 이것이 출시(released) 버전이며 명세는 추후에도 변경되지 않을 것임을 의미한다. 버전이 안정화되지 않았을 때에는 #development를 사용해야 한다.

이제 좀 더 자세히 살펴보도록 하자.

- 메서드 선택자 바로 다음에 pragma 정의, <version:'0.1'>가 보일 것이다. version: pragma는 해당 메서드에 생성된 버전이 CoolBrowser 프로젝트의 0.1 버전과 연관되어야 함을 의미한다. 이것이 바로 메서드명이 중요하지 않다고 언급한 이유다. Metacello는 정의되는 버전을 식별하는 데에 메서드명이 아니라 pragma를 이용한다.
- 메서드의 인자, spec은 메서드에서 유일한 변수로서 4개의 메시지, for:do:, blessing:, package:with:, repository: 의 수신자로서 사용된다.
- 블록이 메시지의 (for:do:, package:with:, …) 인자로서 전달될 때마다 새 객체가 스택으로 삽입되고 블록 내 메시지들은 스택의 젤 위에 위치한 객체로 전송된다.
- #common 기호는 해당 프로젝트 버전이 모든 플랫폼에 공통적임을 의미한다. #common 외에도 Metacello가 실행되는 플랫폼마다 (#pharo, #squeak, #gemstone, #squeakCommon, #pharo, #pharo1.3.x 등등) 사전에 정의된 속성들이 존재한다. Pharo에서 metacelloPlatformAttributes 메서드가 당신이 사용할 수 있는 속성 값을 정의한다.

비밀번호에 관하여. 때로는 Monticello 저장소가 사용자명과 비밀번호를 요구하기도 한다. 이런 경우, repository: 대신 repository:username:password: 메시지를 이용할 수 있다.

```
spec repository: '\url{http://www.example.com/private'} username: 'foo' password: 'bar'
```

명세(specification) 객체. spec 객체는 주어진 버전에 관한 모든 정보를 표현하는 객체다. 버전은 숫자에 불과한 반면 명세는 객체다. spec 메시지를 이용해 명세로 접근할 수 있다 (보통은 불필요하다).

```
(ConfigurationOfCoolBrowser project version: '0.1') spec
```

이는 '0.1' 버전을 정의하는 메서드 정보를 정확히 포함하는 객체(MetacelloMCVersion-Spec 클래스의 인스턴스)를 응답한다.

새 버전 생성하기. 우리 프로젝트의 0.2 버전이 패키지 버전 CoolBrowser-Core-BobJones.15와 CoolBrowser-Tests-JohnLewis.8, 그리고 CoolBrowser-Addons-JohnLewis.3의 버전을 가진 새 패키지 CoolBrowser-Addons로 구성된다고 가정해보자. 이러한 새 설정은 다음과 같이 version02: 로 명명된 메서드를 생성하여 명시할 수 있겠다.

```
ConfigurationOfCoolBrowser>>version02: spec
    <version: '0.2'>

    spec for: #common do: [
        spec repository: 'http://www.example.com/CoolBrowser'.
        spec
            package: 'CoolBrowser-Core' with: 'CoolBrowser-Core-BobJones.15';
            package: 'CoolBrowser-Tests' with: 'CoolBrowser-Tests-JohnLewis.8';
            package: 'CoolBrowser-Addons' with: 'CoolBrowser-Addons-JohnLewis.3']
```

다수의 저장소를 관리하는 방법. 다수의 저장소를 spec으로 추가할 수도 있다. repository: 표현식을 여러 번 명시하면 될 일이다.

```
ConfigurationOfCoolBrowser>>version02: spec
    ...

    spec for: #common do: [
        spec repository: 'http://www.example.com/CoolBrowser'.
        spec repository: 'http://www.anotherexample.com/CoolBrowser'.
        ...
        ]
```

repositoryOverrides 메시지를 사용하는 수도 있다.

```
self whateverVersion repositoryOverrides: (self whateverRepo); load
```

이러한 메시지들은 의존적 설정으로 재귀적으로 퍼지지 않음을 주목하라.

설정 명명하기. 앞에서 우리는 설정 클래스를 명명하는 규칙을 살펴보았다. 우리 예제에서는 ConfigurationOfCoolBrowser가 해당한다. 설정 클래스와 이름이 동일한 Monticello 패키지를 생성하고, 클래스를 그 패키지로 넣는 방법도 있다. 따라서 이번 예제에선 하나의 클래스 ConfigurationOfCoolBrowser만 포함하는 패키지 ConfigurationOfCoolBrowser를 생성해 볼 것이다.

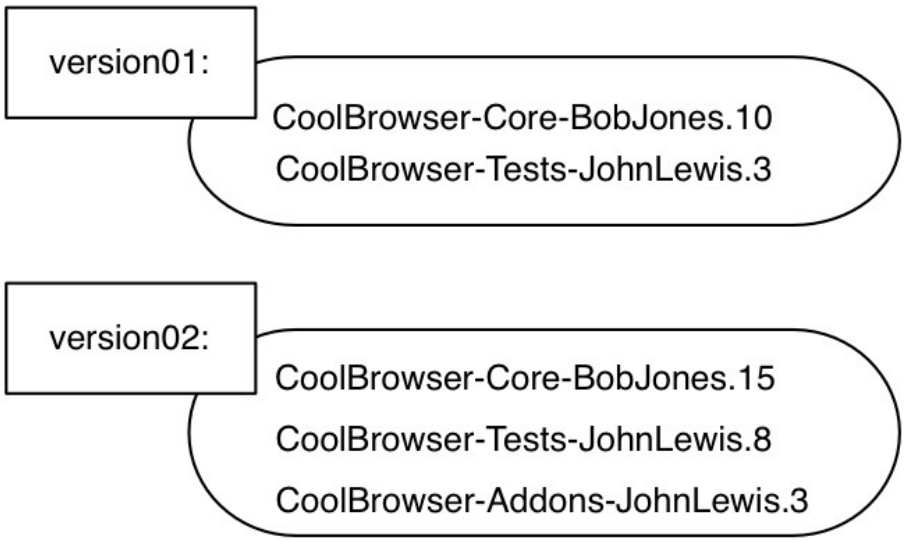


그림 9.3: 프로젝트의 두 가지 버전.

패키지명과 설정 가능한 클래스명을 동일하게 만들고, ConfigurationOf 문자열로 시작하도록 만듦으로서 이용 가능한 프로젝트를 열거하는 저장소를 쉽게 스캔할 수 있다. 설정이 자신만의 Monticello 저장소에 보관되어도 매우 편리할 것이다.

Metacello 설정 로딩하기

물론 Metacello에서 프로젝트 설정을 명시하는 이유는 정확히 특정 설정을 자신의 이미지에 로딩하여 패키지 버전의 일관된 집합을 갖는 데에 있다. 버전을 로딩하기 위해서는 버전에 load 메시지를 전송해야 한다. CoolBrowser의 버전들을 로딩하는 예를 들어보자.

```
(ConfigurationOfCoolBrowser project version: '0.1') load.
(ConfigurationOfCoolBrowser project version: '0.2') load.
```

뿐만 아니라 각 표현식의 결과를 인쇄할 경우 로딩 순서로 된 패키지 목록이 표시된다는 점도 주목해야 하는데, Metacello는 어떤 패키지가 로딩되는지 뿐만 아니라 그 순서까지 관리하기 때문이다. 설정을 디버깅하기가 수월해질 것이다.

선택적 로딩. 기본적으로 load 메시지는 버전과 연관된 모든 패키지를 로딩한다 (후에 살펴보겠지만 이는 default 라는 특정 그룹을 정의하여 변경할 수 있다). 프로젝트에 패키지의 하위집합을 로딩하고 싶다면 관심 있는 패키지들의 이름을 load: 메서드에 대한 인자로서 열거해야 한다.

```
(ConfigurationOfCoolBrowser project version: '0.2') load:
    { 'CoolBrowser-Core' .
    'CoolBrowser-Addons' }.
```

설정 디버깅하기. 설정을 실제로 로딩하지 않고 로딩을 시뮬레이트하고 싶다면 load(혹은 load:) 대신 record(혹은 record:)를 사용해야 한다. 이후 시뮬레이션 결과를 얻기 위해서는 아래와 같이 loadDirective 메시지를 전송해야 한다.

```
((ConfigurationOfCoolBrowser project version: '0.2') record:
    { 'CoolBrowser-Core' .
    'CoolBrowser-Addons' }) loadDirective.
```

load 와 record 외에도 fetch(그리고 fetch:)라는 유용한 메서드가 있다. 설명한 바와 같이 record는 로딩되어야 하는 Monticello 파일과 그 순서를 단순히 기록하는 일만 한다. fetch는 필요한 Monticello 파일로 모두 접근하여 다운로드한다. 확실히 말해두지만 implementation에서 load는 fetch를 먼저 실행한 다음 doLoad를 실행한다.

패키지 간 의존성 관리하기

프로젝트는 주로 여러 개의 패키지로 구성되고, 이 패키지들은 종종 다른 패키지로 의존성을 갖는다. 특정 패키지는 다른 패키지의 특정 버전에 의존하기 쉽다. 의존성을 올바로 처리하는 것이 매우 중요하며 Metacello의 주요 이점에 해당하기도 한다. 의존성에는 두 가지 유형이 있다.

내적 의존성. 프로젝트 내에는 여러 개의 패키지가 포함되어 있으며, 그 중 일부는 동일한 프로젝트의 여러 패키지에 의존한다.

프로젝트 간 의존성. 프로젝트가 다른 프로젝트에 의존하거나, 다른 프로젝트로부터 비롯된 일부 패키지에 의존하는 것은 흔한 일이다. 가령 Pier(meta-described 내용 관리 시스템)는 Magritte(메타데이터 모델링 프레임워크)와 Seaside(웹 애플리케이션 개발용 프레임워크)에 의존한다.

지금은 내적 의존성에 중점을 둘 것인데, 그림 9.4에서 설명한 바와 같이 CoolBrowser-Tests와 CoolBrowser-Addons 패키지 모두 CoolBrowser-Core에 의존한다고 가정하자. 버전 0.1과 0.2에 대한 명세는 이러한 의존성을 포착하지 않았다. 새 설정은 다음과 같다.

```
ConfigurationOfCoolBrowser>>version03: spec
    <version: '0.3'>

    spec for: #common do: [
        spec repository: 'http://www.example.com/CoolBrowser'.
        spec
            package: 'CoolBrowser-Core' with: 'CoolBrowser-Core-BobJones.15';
            package: 'CoolBrowser-Tests' with: [
                spec
                    file: 'CoolBrowser-Tests-JohnLewis.8';
                    requires: 'CoolBrowser-Core' ];
            package: 'CoolBrowser-Addons' with: [
                spec
                    file: 'CoolBrowser-Addons-JohnLewis.3';
                    requires: 'CoolBrowser-Core' ]].
```

version03: 에서 requires: 지시어를 이용해 의존성 정보를 추가하였다.

또한 패키지의 특정 버전을 참조하는 file: 메시지를 소개하였다. CoolBrowser-Tests와 CoolBrowser-Addons는 그들이 로딩되기 전에 로딩되어야 하는 CoolBrowser-Core를 필요로 한다. 그들이 의존하는 Cool-Browser-Core의 정확한 버전을 명시하지는 않았음을 주목하라. 이는 문제를 야기할 수 있으나 그 결함은 조만간 다룰 것이니 걱정하지 말길 바란다!

이 버전을 이용하면 구조적 정보(필요한 패키지와 저장소)와 버전 정보(정확한 번호 버전)를 결합한다. 시간이 지나면서 버전 정보는 자주 변경될 것이지만 구조적 정보는 다소 같은 상태로 유지될 것이라고 예상할 수 있다. 이를 포착하기 위해 Metacello는 Baselines란 개념을 도입한다.

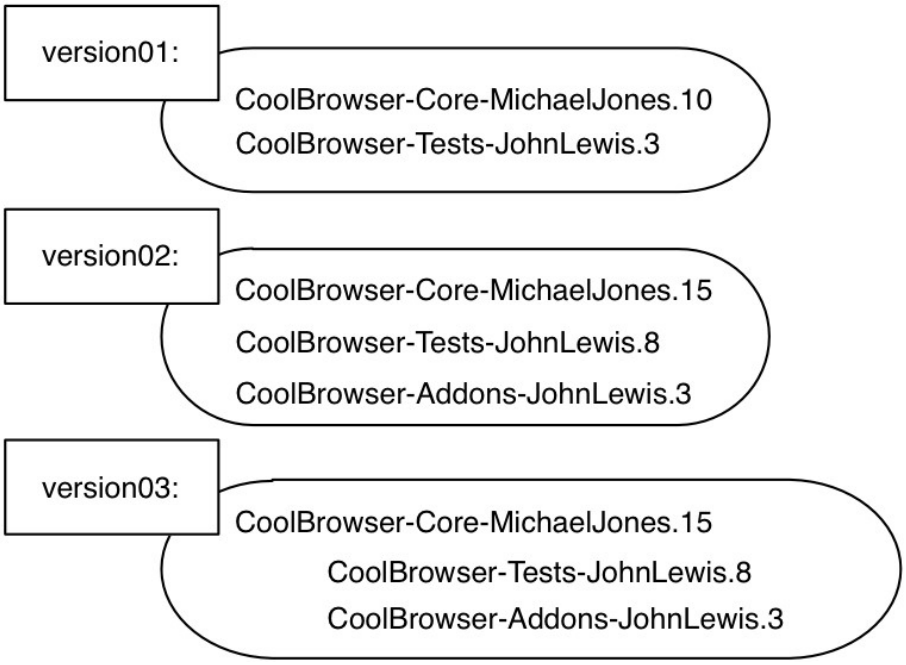


그림 9.4: 버전 0.3은 동일한 프로젝트 내 패키지들 간 내적 의존성을 표현한다.

Baselines

baseline이란 패키지나 프로젝트 간 구조적 의존성에 관한 프로젝트의 아키텍처 또는 뼈대를 의미한다. baseline은 패키지명만 이용해 프로젝트의 구조를 정의한다. 구조가 변경되면 baseline은 업데이트되어야 한다. 구조적 변경이 없다면 변경은 baseline에서 패키지의 특정 버전을 고르는 것으로 제한된다.

이제 우리 예제를 계속해보자. 가장 먼저 baseline을 이용해 수정할 것인데, baseline에 대한 메서드를 하나 생성하겠다. 메서드명과 버전 pragma는 어떤 형태든 취할 수 있음을 명심한다. 하지만 가독성을 위해서는 둘 다 'baseline'을 추가해야 한다. 의무적인 것은 blessing: 메시지의 인자인데, 이는 baseline을 정의한다.

```
ConfigurationOfCoolBrowser>>baseline04: spec "convention"
    <version: '0.4-baseline'> "convention"

    spec for: #common do: [
        spec blessing: #baseline. "mandatory to declare a baseline"
        spec repository: 'http://www.example.com/CoolBrowser'.
        spec
            package: 'CoolBrowser-Core';
            package: 'CoolBrowser-Tests' with: [ spec requires: 'CoolBrowser-Core'];
            package: 'CoolBrowser-Addons' with: [ spec requires: 'CoolBrowser-Core']]
```

baseline04: 메서드는 여러 버전에 의해 사용되는지도 모르는 0.4-baseline의 구조를 정의한다. 예를 들어 아래에 정의된 버전 0.4는 그림 9.5에서 보이는 바와 같이 이러한 구조를 이용한다. baseline은 저장소, 패키지, 그 패키지들 간 의존성을 명시하지만 패키지의 구체적 버전을 명시하지는 않는다.

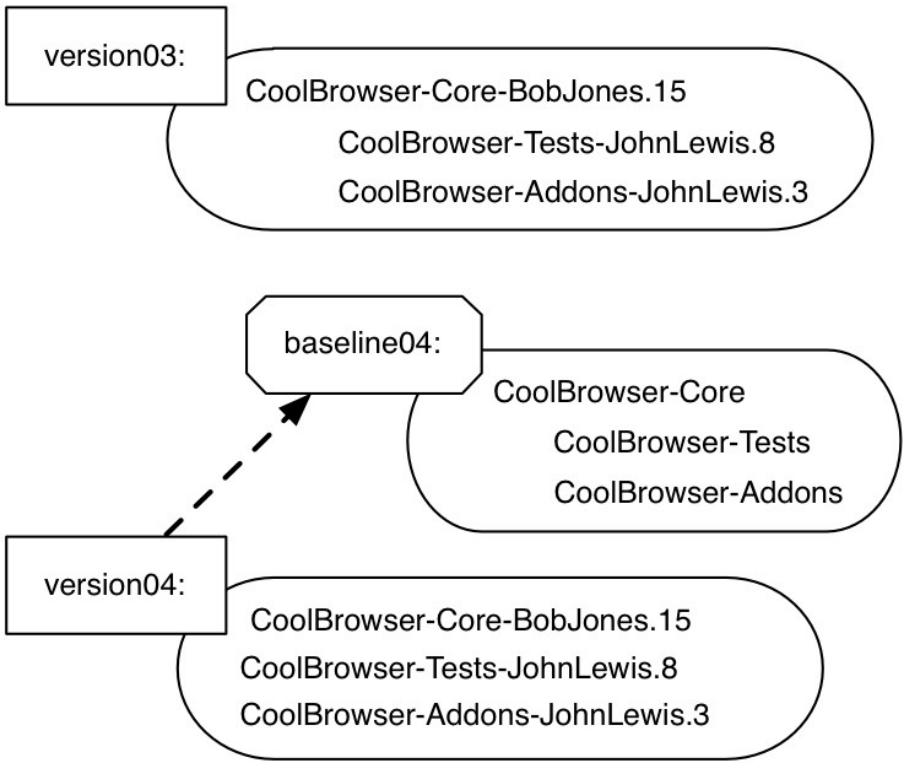


그림 9.5: 버전 0.4는 이제 패키지 간 의존성을 표현하는 baseline을 가져온다(import).

baseline과 관련해 버전을 정의하기 위해 아래와 같이 pragma <version:imports:>를 사용한다.

```
ConfigurationOfCoolBrowser>>version04: spec
    <version: '0.4' imports: #('0.4-baseline')>

    spec for: #common do: [
        spec
            package: 'CoolBrowser-Core' with: 'CoolBrowser-Core-BobJones.15';
            package: 'CoolBrowser-Tests' with: 'CoolBrowser-Tests-JohnLewis.8';
            package: 'CoolBrowser-Addons' with: 'CoolBrowser-Addons-JohnLewis.3'
    ].
```

version04: 메서드에서 우리는 패키지의 구체적인 버전을 명시한다. version:imports: pragma는 해당 버전 (버전 '0.4') 이 기반으로 하는 버전의 목록을 명시한다. 구체적 버전이 명시되면 그것이 baseline을 사용한다는 사실과 상관없이 이전과 같은 방식으로 로딩된다.

```
(ConfigurationOfCoolBrowser project version: '0.4') load.
```

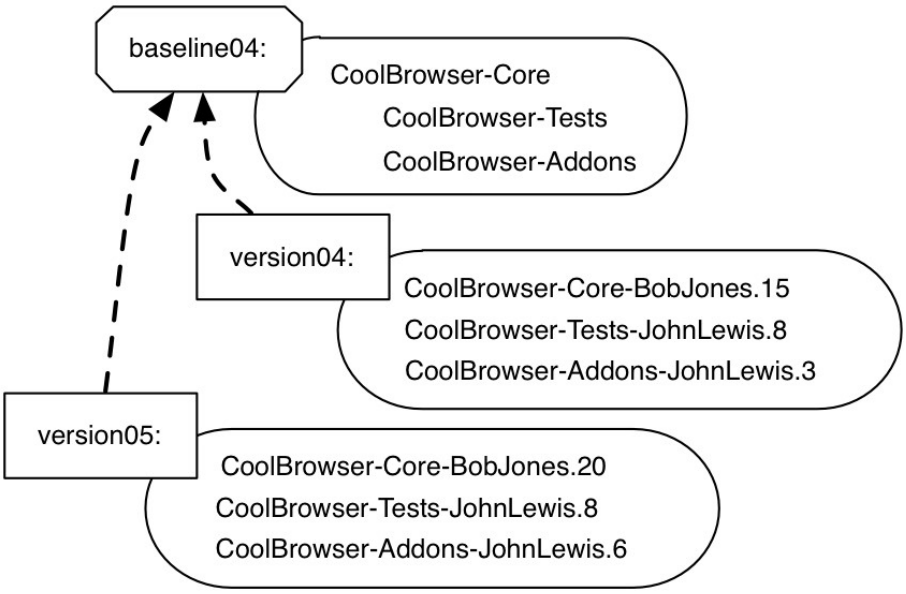


그림 9.6: 두 번째 버전 (0.5) 은 버전 0.4와 같은 baseline을 가져온다.

Baseline 로딩하기

버전 0.4-baseline이 명시적인 패키지 버전 정보를 포함하지 않는다 하더라도 로딩하는 방법은 있다!

```
(ConfigurationOfCoolBrowser project version: '0.4-baseline') load.
```

로더가 버전 정보가 없는 패키지를 마주치면 저장소로부터 패키지의 가장 최신 정보를 로딩하고자 시도한다.

때때로, 그 중에서도 여러 개발자가 프로젝트를 작업할 때는 모든 개발자들의 최신 작업으로 접근하기 위해 baseline 버전을 로딩하는 편이 유용하다. 이러한 경우 baseline 버전은 정말로 "최첨단" 버전이 된다.

새 버전 선언하기. 이제 프로젝트의 새 버전, 즉 버전 0.4와 구조는 같지만 패키지의 다른 버전들을 포함하는 버전 0.5를 생성한다고 가정하자. 동일한 baseline을 가져옴으로써 이러한 계획을 담아낼 수 있는데, 이 관계는 그림 9.6에 설명되어 있다.

```
ConfigurationOfCoolBrowser>>version05: spec
    <version: '0.5' imports: #('0.4-baseline')>

    spec for: #common do: [
    spec
        package: 'CoolBrowser-Core' with: 'CoolBrowser-Core-BobJones.20';
        package: 'CoolBrowser-Tests' with: 'CoolBrowser-Tests-JohnLewis.8';
        package: 'CoolBrowser-Addons' with: 'CoolBrowser-Addons-JohnLewis.6' ].
```

큰 프로젝트에 대한 baseline을 생성 시 보통은 어느 정도 시간과 노력이 요구되는데 모든 패키지의 모든 의존성 뿐만 아니라 다른 것들도 포착해야 하기 때문이며, 그러한 것들은 후에 살펴보도록 하겠다. 하지만 baseline이 정의되고 나면 프로젝트의 새 버전을 생성하는 과정이 크게 단순화되고 소요되는 시간도 줄어든다.

그룹

CoolBrowser 프로젝트가 증가하여 개발자가 CoolBrowser-Addons에 대한 테스트를 몇 가지 작성한다고 가정해보자. 이는 그림 9.7과 같이 CoolBrowser-Addons와 CoolBrowser-Tests에 의존하는 CoolBrowser-AddonsTests라는 새 패키지를 구성한다.

테스트의 유무와 상관없이 프로젝트를 로딩하길 원하는경우, 아래와 같이 모든 테스트 패키지를 명시적으로 열거하는 대신,

```
(ConfigurationOfCoolBrowser project version: '0.6')
    load: #('CoolBrowser-Tests' 'CoolBrowser-AddonsTests').
```

아래와 같은 간단한 표현식으로 모든 테스트를 로딩할 수 있다면 편리하겠다.

```
(ConfigurationOfCoolBrowser project version: '0.6') load: 'Tests'.
```

Metacello는 그룹이란 개념을 제공한다. 그룹은 항목(item)의 집합체로, 각 항목은 패키지, 프로젝트, 또는 심지어 다른 그룹이 될 수도 있다.

그룹은 다양한 목적으로 항목의 집합을 명명하도록 해주기 때문에 유용하다. 사용자에게 core만 설치하거나 add-on와 개발 기능이 있는 core를 설치할 수 있는 기회를 제공하여 적절한 그룹을 정의하는 일을 수월하게 만들 수 있길 원할지도 모르겠다. 그림 9.7의 예제로 돌아가 새 baseline, 즉 6개 그룹을 정의하는 0.6-baseline 을 어떻게 정의하는지 살펴보자. 이 예제에서 우리는 CoolBrowser-Tests와 CoolBrowser-AddonsTests를 구성하는 Tests라는 그룹을 생성한다.

그룹을 정의하기 위해 group: groupName with: groupElements 메서드를 이용한다. with: 인자는 패키지명, 프로젝트, 다른 그룹, 또는 이러한 것들의 집합체가 되기도 한다. 그림 9.7에

해당하는 코드는 다음과 같다.

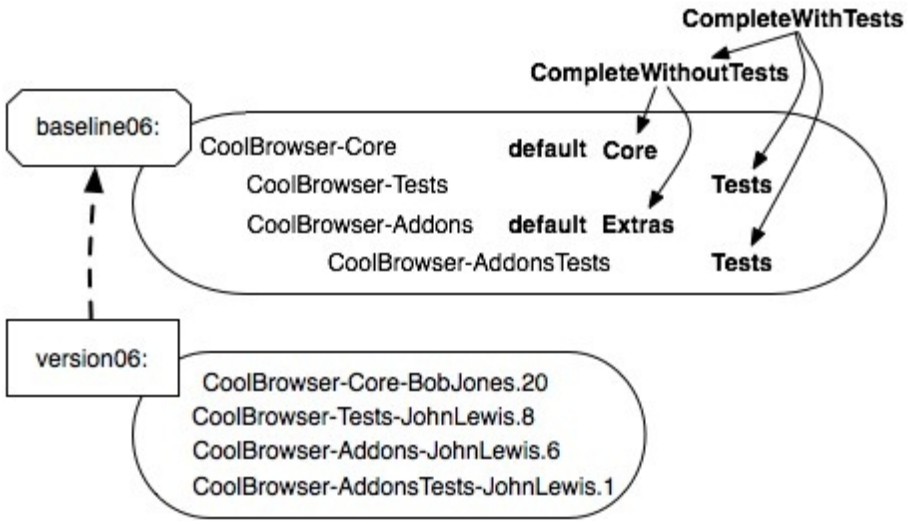


그림 9.7: 6개 그룹, baseline: default, Core, Extras, Tests, CompleteWithoutTests, CompleteWithTests이 있는 baseline.

```
ConfigurationOfCoolBrowser>>baseline06: spec
    <version: '0.6-baseline'>
    spec for: #common do: [
        spec blessing: #baseline.
        spec repository: 'http://www.example.com/CoolBrowser'.
        spec
            package: 'CoolBrowser-Core';
            package: 'CoolBrowser-Tests' with: [ spec requires: 'CoolBrowser-Core' ];
            package: 'CoolBrowser-Addons' with: [ spec requires: 'CoolBrowser-Core' ] ;
            package: 'CoolBrowser-AddonsTests' with: [
                spec requires: #('CoolBrowser-Addons' 'CoolBrowser-Tests' ) ].
        spec
            group: 'default' with: #('CoolBrowser-Core' 'CoolBrowser-Addons');
            group: 'Core' with: #('CoolBrowser-Core');
            group: 'Extras' with: #('CoolBrowser-Addons');
            group: 'Tests' with: #('CoolBrowser-Tests' 'CoolBrowser-AddonsTests');
            group: 'CompleteWithoutTests' with: #('Core' 'Extras');
            group: 'CompleteWithTests' with: #('CompleteWithoutTests' 'Tests')
        ].
```

그룹은 baseline에서 정의된다. 그룹은 baseline 버전에서 정의할 것인데, 그룹이 구조적 구성 요소이기 때문이다. 기본 그룹은 다음 절들에 걸쳐 사용될 것임을 주목하라. 여기서 기본 그룹은 load 메서드가 사용되면 'CoolBrowser-Core'와 'CoolBrowser-Addons' 패키지가 로딩될 것임을 언급한다.

이제 CoolBrowser-AddonsTests라는 새 패키지가 추가된다는 점만 제외하면 해당 baseline을 이용해 0.5 버전과 동일한 버전 0.6을 정의할 수 있다.

```
ConfigurationOfCoolBrowser>>version06: spec
    <version: '0.6' imports: #('0.6-baseline')>

    spec for: #common do: [
        spec
            package: 'CoolBrowser-Core' with: 'CoolBrowser-Core-BobJones.20';
            package: 'CoolBrowser-Tests' with: 'CoolBrowser-Tests-JohnLewis.8';
            package: 'CoolBrowser-Addons' with: 'CoolBrowser-Addons-JohnLewis.6' ;
          package: 'CoolBrowser-AddonsTests' with: 'CoolBrowser-AddonsTests-JohnLewis.1' ].
```

예제. 그룹을 정의했다면 프로젝트나 패키지의 이름을 사용하는 곳이라면 어디서든 그룹명을 사용할 수 있다. load: 메서드는 패키지명, 프로젝트명, 그룹명, 또는 그러한 항목의 컬렉션명을 매개변수로 취한다. 따라서 아래의 문이 모두 가능하다.

```
(ConfigurationOfCoolBrowser project version: '0.6') load: 'CoolBrowser-Core'.
    "Load a single package"

(ConfigurationOfCoolBrowser project version: '0.6') load: 'Core'.
    "Load a single group"

(ConfigurationOfCoolBrowser project version: '0.6') load: 'CompleteWithTests'.
    "Load a single group"

(ConfigurationOfCoolBrowser project version: '0.6')
    load: #('CoolBrowser-Core' 'Tests').
    "Loads a package and a group"

(ConfigurationOfCoolBrowser project version: '0.6')
    load: #('CoolBrowser-Core' 'CoolBrowser-Addons' 'Tests').
    "Loads two packages and a group"

(ConfigurationOfCoolBrowser project version: '0.6')
    load: #('CoolBrowser-Core' 'CoolBrowser-Tests').
    "Loads two packages"

(ConfigurationOfCoolBrowser project version: '0.6') load: #('Core' 'Tests').
    "Loads two groups"
```

그룹 default와 'ALL'. default 그룹은 특별한 그룹이다. load 메시지는 default 그룹의 멤버들을 로딩하는 반면 ALL 그룹을 로딩하면 모든 패키지들이 로딩된다. 게다가 default는 기본값으로 ALL을 로딩한다!

```
(ConfigurationOfCoolBrowser project version: '0.6') load.
```

이는 CoolBrowser-Core와 CoolBrowser-Addons만 로딩한다.

default 그룹이 있다면 프로젝트의 모든 패키지를 어떻게 로딩할 것인가? 아래와 같이 미리 정의된 ALL 그룹을 사용한다.

```
(ConfigurationOfCoolBrowser project version: '0.6') load: 'ALL'.
```

Core, Tests, 그리고 default에 관하여

설정에는 최소 두 개의 그룹, 즉 Core(실제 코드)와 Tests(연관된 테스트)가 있는 것이 보통이다. 문제는 default 그룹이 무엇을 로딩하느냐가 된다 (매개변수로 어떤 것도 명시하지 않을 경우 로딩되는 것임을 명심하라).

spec group: 'default' with: #('Core')라고 말하는 것은 우리는 기본적으로 tests를 로딩하지 않음을 말하는 것과 같다.

이제 어떤 default도 명시하지 않을 경우 기본 값으로 모든 것을 취하기 때문에 우리 예제의 spec group: 'default' with: #('Core' 'Tests')와 같을 것이다.

기본적으로 tests도 로딩하는 편이 낫다고 생각한다. 이것이 바로 default 그룹에 Tests 그룹을 명시적으로 넣거나 default를 아예 명시하지 않는 이유다.

프로젝트 간 의존성

패키지가 다른 패키지에 의존하는 것과 같이 프로젝트 또한 다른 프로젝트에 의존할 수 있다. 예를 들어, 내용 관리 시스템(CMS)에 해당하는 Pier는 Magritte와 Seaside에 의존한다. 프로젝트는 하나 또는 이상의 프로젝트의 엔티티, 다른 프로젝트로부터 패키지 그룹, 다른 프로젝트로부터 하나 또는 두 개의 패키지에 의존할 수 있다.

Metacello 설명 없이 프로젝트에 의존하기

프로젝트 X로부터 패키지 A가 프로젝트 Y의 프로젝트 B를 의존하고, 프로젝트 Y는 Metacello를 이용해 설명되지 않았다고 가정하자. 이런 경우 의존성을 아래와 같이 설명할 수 있겠다.

```
"In a baseline method"
spec
    package: 'PackageA' with: [ spec requires: #('PackageB')];
    package: 'PackageB' with: [ spec
        repository: 'http://www.smalltalkhub.com/ProjectB' ].
```

```
"In the version method"
package: 'PackageB' with: 'PackageB-JuanCarlos.80'.
```

어느 정도는 작동한다. 단, 이 접근법에서는 프로젝트 B가 Metacello 설정에 의해 설명되지 않기 때문에 B의 의존성이 관리되지 않는다는 결함이 있다. 즉, B 패키지의 어떤 의존성도 로딩되지 않을 것이다. 따라서 이런 경우 시간을 들여 프로젝트 B에 대한 설정을 생성할 것을 권한다.

Metacello 설정이 있는 프로젝트에 의존하기

이제 우리가 의존하는 프로젝트가 Metacello를 이용해 설명되는 경우를 생각해보자. CoolBrowser 프로젝트로부터 패키지를 이용하는 CoolToolSet라는 새 프로젝트를 소개해보자. 그 설정 클래스를 ConfigurationOfCoolToolSet이라고 부른다. CoolToolSet에는 CoolToolSet-Core와 CoolToolSet-Tests라고 불리는 두 개의 패키지가 있다고 가정하자. 이러한 패키지들은 CoolBrowser의 패키지에 의존한다.

CoolToolSet의 버전 0.1은 baseline을 가져오는 일반 버전에 불과하다.

```
ConfigurationOfCoolToolSet>>version01: spec
    <version: '0.1' imports: #('0.1-baseline')>
    spec for: #common do: [
        spec
            package: 'CoolToolSet-Core' with: 'CoolToolSet-Core-AlanJay.1';
            package: 'CoolToolSet-Tests' with: 'CoolToolSet-Tests-AlanJay.1'.].
```

당신이 의존하는 프로젝트가 관습을 따른다면 (예: ConfigurationOfCoolBrowser 패키지 내의 ConfigurationOfCoolBrowser 클래스) baseline의 정의는 간단하다. 기본적으로는 당신이 로딩하길 원하는 버전(versionString: 을 이용)과 프로젝트 저장소(repository: 이용)만 명시하면 된다.

```
ConfigurationOfCoolToolSet >>baseline01: spec
    <version: '0.1-baseline'>
    spec for: #common do: [
        spec repository: 'http://www.example.com/CoolToolSet'.
        spec project: 'CoolBrowser ALL' with: [
            spec
                repository: 'http://www.example.com/CoolBrowser';
                loads: #('Core' 'Tests');
                versionString: '2.5' ]
        spec
            package: 'CoolToolSet-Core' with: [ spec requires: 'CoolBrowser ALL' ];
            package: 'CoolToolSet-Tests' with: [ spec requires: 'CoolToolSet-Core' ]].
```

프로젝트 참조를 CoolBrowserALL이라고 명명했다. 프로젝트 참조 이름은 임의로 정해지며 원하는 대로 선택이 가능하지만 해당하는 프로젝트 참조에 의미가 통하는 이름으로 정하길 권한다. CoolToolSet-Core 패키지에 대한 명세에서 CoolBrowser ALL이 필요함을 명시하였다. 후에 설명하겠지만 project:with: 메시지는 로딩하길 원하는 프로젝트의 정확한 버전을 명시할 수 있도록 해준다.

load: 메시지는 로딩할 패키지 또는 그룹을 명시한다. load: 의 매개변수는 load의 매개변수와 같을 수도 있으며, 패키지명, 그룹명, 이러한 것들의 컬렉션명이 해당하겠다. load: 은 선택적으로 호출이 가능함을 주목해야 하는데, 기본 값과 다른 무언가를 로딩하길 원할 때 필요할 것이다.

이제 아래와 같이 CoolToolSet 를 로딩할 수 있다.

```
(ConfigurationOfCoolToolSet project version: '0.1') load.
```

비관습적 프로젝트

당신이 의존하는 프로젝트가 기본 관습을 따르지 않는다면 설정을 식별하기 위해 더 많은 정보를 제공해야 할 것이다. 설정이 만일 권장하는 ConfigurationOfCoolBrowser 대신 CoolBrowser-Metacello라는 Monticello 패키지에 보관된 ConfigurationOfCoolBrowser 클래스에 보관된다고 가정해보자.

```
ConfigurationOfCoolToolSet >>baseline01: spec
    <version: '0.1-baseline'>
    spec for: #common do: [
        spec repository: 'http://www.example.com/CoolToolSet'.
        spec project: 'CoolBrowser ALL' with: [
            spec
                className: 'ConfigurationOfCoolBrowser';
                loads: #('ALL' );
                file: 'CoolBrowser-Metacello';
                repository: 'http://www.example.com/CoolBrowser' ].
        spec
            package: 'CoolToolSet-Core' with: [ spec requires: 'CoolBrowser ALL' ];
            package: 'CoolToolSet-Tests' with: [ spec requires: 'CoolToolSet-Core' ]].
```

- className:는 프로젝트 메타데이터를 포함하는 클래스명을 명시하는데, 이번 경우 ConfigurationOfCoolBrowser가 되겠다.
- file:와 repository: 메시지는 ConfigurationOfCoolBrowser가 이미지에 없을 경우 이를 검색하고 로딩해야 한다는 정보를 Metacello에게 제공한다. file: 의 인자는 메타데이터 클래스를 포함하는 Monticello 패키지의 이름이며, repository: 의 인자는 패키지를 포함하는 Monticello 저장소의 URL이다. Monticello 저장소가 보호된 경우 대신 repository:username:password: 메시지를 사용해야 한다.

이제 다음과 같이 CoolToolSet를 로딩할 수 있다.

```
(ConfigurationOfCoolToolSet project version: '0.1') load.
```

다수의 프로젝트에 의존하기

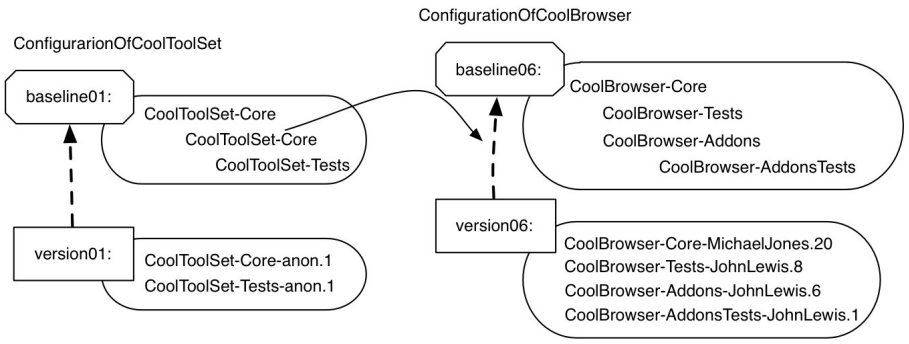


그림 9.8: 설정 간 의존성.

'ALL'을 이용하면 CoolToolSet-Core 이전에 전체 CoolBrowser 프로젝트가 로딩되는 결과를 야기할 것이다. CoolBrowser의 테스트 패키지 상의 의존성을 core 패키지 상의 의존성과 구별하여 명시하고 싶다면 아래 baseline을 정의해야 할 것이다.

```
ConfigurationOfCoolToolSet>>baseline02: spec
    <version: '0.2-baseline'>
    spec for: #common do: [
        spec blessing: #baseline.
        spec repository: 'http://www.example.com/CoolToolSet'.
        spec
            project: 'CoolBrowser default' with: [
                spec
                    className: 'ConfigurationOfCoolBrowser'; ''this is optional''
                    loads: #('default'); ''this is optional''
                    repository: 'http://www.example.com/CoolBrowser' ].
            project: 'CoolBrowser Tests' with: [
                spec
                    loads: #('Tests' );
                    repository: 'http://www.example.com/CoolBrowser' ].
        spec
            package: 'CoolToolSet-Core' with: [ spec requires: 'CoolBrowser default' ];
            package: 'CoolToolSet-Tests' with: [
                spec requires: #('CoolToolSet-Core' 'CoolBrowser Tests') ].].
```

해당 baseline은 두 개의 프로젝트 참조를 생성하는데, CoolBrowser default로 명명된 참조는 default 그룹을 로딩하고, 'CoolBrowser Tests'로 명명된 참조는 CoolBrowser의 설정의 'Tests' 그룹을 로딩한다. 우리는 CoolToolSet-Core가 CoolBrowser default를 필요로 하고 CoolToolSet-Tests가 CoolTestSet-Core와 CoolBrowser Tests를 필요로 하도록 선언한다. 의존적 프로젝트들의 컬렉션과 함께 requires: 의 사용도 주목한다.

이제 아래처럼 core 패키지만 로딩하거나,

```
(ConfigurationOfCoolToolSet project version: '0.2') load: 'CoolToolSet-Core'.
```

테스트만 로딩하는 것이 가능해졌다 (이 또한 core를 로딩할 것이다).

```
(ConfigurationOfCoolToolSet project version: '0.2') load: 'CoolToolSet-Tests'.
```

내적 의존성과 마찬가지로 baseline 0.2-baseline (그리고 0.1-baseline에서도 마찬가지로)은 설정이 의존하는 프로젝트 버전을 명시하지 않는다. 대신 버전 메서드에서 project:with: 메시지를 이용해 명시한다.

```
ConfigurationOfCoolToolSet>>version02: spec
    <version: '0.2' imports: #('0.2-baseline' )>
    spec for: #common do: [
        spec blessing: #beta.
        spec
            package: 'CoolToolSet-Core' with: 'CoolToolSet-Core-AlanJay.1';
            package: 'CoolToolSet-Tests' with: 'CoolToolSet-Tests-AlanJay.1';
            project: 'CoolBrowser default' with: '1.3';
            project: 'CoolBrowser Tests' with: '1.3'].
```

특정 패키지 로딩하기

버전 메서드는 baseline 메서드 외에도 어떤 패키지를 로딩할 것인지 명시할 수 있다. ConfigurationOfSoup를 예로 들어, 버전 1.2에 'XML-Parser'와 'XML-Tests-Parser' 패키지를 로딩하길 원한다고 가정하자.

```
ConfigurationOfSoup>>version10: spec
    <version: '1.0' imports: #('1.0-baseline')>

    spec for: #common do: [
        spec
            project: 'XMLSupport'
            with: [spec
                loads: #('XML-Parser' 'XML-Tests-Parser');
                versionString: '1.2.0'].

    spec
        package: 'Soup-Core' with: 'Soup-Core-sd.11';
        package: 'Soup-Tests-Core' with: 'Soup-Tests-Core-sd.3';
        package: 'Soup-Help' with: 'Soup-Help-StephaneDucasse.2' ].
```

당신이 할 수 있는 일은 로딩하고자 하는 프로젝트의 패키지를 명시하기 위해 프로젝트 참조에서 load: 메시지를 사용하는 것이다. 이는 당신이 프로젝트 참조에 정보를 팩토링하고 모든 버전에서 중복할 필요가 없기 때문에 바람직한 해결책이다.

```
ConfigurationOfSoup>>version10: spec
    <version: '1.0' imports: #('1.0-baseline')>

    spec for: #pharo do: [
        spec project: 'XMLSupport' with: [
            spec
                versionString: #stable;
                loads: #('XML-Parser' 'XML-Tests-Parser');
                repository: 'http://ss3.gemstone.com/ss/xmlsupport' ].

    spec
        package: 'Soup-Core' with: 'Soup-Core-sd.11';
        package: 'Soup-Tests-Core' with: 'Soup-Tests-Core-sd.3';
        package: 'Soup-Help' with: 'Soup-Help-StephaneDucasse.2' ].
```

Baseline 내 버전. 권장하진 않지만 baseline으로부터 버전을 명시하지 못하도록 당신을 막을 수 있는 방도는 없다. 프로젝트 참조도 마찬가지다. 따라서 file:, className:, repository: 등의 메시지 외에도 프로젝트 참조에 직접 프로젝트의 버전을 명시하도록 해주는 versionString: 이라는 메시지가 있는데, 예를 들자면 다음과 같다.

```
ConfigurationOfCoolToolSet >>baseline011: spec
    <version: '0.1.1-baseline'>
    spec for: #common do: [
        spec repository: 'http://www.example.com/CoolToolSet'.
        spec project: 'CoolBrowser ALL' with: [
            spec
                className: 'ConfigurationOfCoolBrowser';
                loads: #('ALL' );
                versionString: '0.6' ;
                file: 'CoolBrowser-Metacello';
                repository: 'http://www.example.com/CoolBrowser' ].
        spec
            package: 'CoolToolSet-Core' with: [ spec requires: 'CoolBrowser ALL' ];
            package: 'CoolToolSet-Tests' with: [ spec requires: 'CoolToolSet-Core' ]].
```

버전 메서드 내에 CoolBrowser default와 CoolBrowser Tests 참조에 대한 버전을 정의하지 않은 경우 baseline 내에 명시된 (versionString: 을 사용) 버전이 사용된다. baseline 메서

드에 명시된 버전이 없다면 Metacello는 가장 최근 프로젝트 버전을 로딩한다.
정보 재사용하기. baseline02:을 보면 정보가 두 개의 프로젝트 참조에서 중복됨을 확인할 수 있다. 중복을 제거하기 위해서는 project:copyFrom:with: 메서드를 이용할 수 있다.

```
ConfigurationOfCoolToolSet >>baseline02: spec
    <version: '0.2-baseline'>
    spec for: #common do: [
        spec blessing: #baseline.
        spec repository: 'http://www.example.com/CoolToolSet'.
        spec
            project: 'CoolBrowser default' with: [
                spec
                    loads: #('default');
                    repository: 'http://www.example.com/CoolBrowser';
                    file: 'CoolBrowser-Metacello']
            project: 'CoolBrowser Tests'
                copyFrom: 'CoolBrowser default'
                with: [ spec loads: #('Tests').].
        spec
            package: 'CoolToolSet-Core' with: [ spec requires: 'CoolBrowser default' ];
            package: 'CoolToolSet-Tests' with: [
                spec requires: #('CoolToolSet-Core' 'CoolBrowser Tests') ].].
```

의존성 세분성(granularity)에 관하여

패키지에 의존하는 일, 프로젝트에 의존하는 일, 이를 표현하는 다른 방법들 간 차이를 논하고자 한다. Fame의 baseline1.1을 생각해보자.

```
baseline11: spec
    <version: '1.1-baseline'>
    spec for: #'common' do: [
        spec blessing: #'baseline'.
        spec description: 'Baseline 1.1 first version on SmalltalkHub, copied from baseline
        1.0 on SqueakSource'.
        spec repository: 'http://www.smalltalkhub.com/mc/Moose/Fame/main'.
        spec
            package: 'Fame-Core';
            package: 'Fame-Util';
            package: 'Fame-ImportExport' with: [spec requires: #('Fame-Core' ) ];
            package: 'Fame-SmalltalkBinding' with: [spec requires: #('Fame-Core' ) ];
            package: 'Fame-Example';
            package: 'Phexample' with: [spec repository: 'http://smalltalkhub.com/mc/
        PharoExtras/Phexample/main' ];
            package: 'Fame-Tests-Core' with: [spec requires: #('Fame-Core' 'Fame-
        Example' 'Phexample' ) ].
        spec
            group: 'Core' with: #('Fame-Core' 'Fame-ImportExport' 'Fame-Util' 'Fame-
        SmalltalkBinding' );
            group: 'Tests' with: #('Fame-Tests-Core' ) ].
```

baseline에서 package:'Phexample'with:[spec repository: 'http://smalltalkhub.com/mc/PharoExtras/Phexample/mai
]; 는 명시된 저장소에 하나의 패키지가 발견됨을 나타낸다. 하지만 그러한 접근법은 바람직하지 못한데, 이 방식대로 이용한다면 저장소로부터 패키지를 다운로드할 뿐, Metacello가 관계되지 않는다. 예를 들어, PhexampleCore가 의존성을 갖고 있더라도 로딩되지 않을 것이란

의미다.

아래 baseline1.2 예제에서 우리는 위에 제시한 프로젝트 간 의존성을 표현하겠다.

```
baseline12: spec
    <version: '1.2-baseline'>
    spec for: #'common' do: [
        spec blessing: #'baseline'.
        spec description: 'Baseline 1.2 to make explicit that Fame depends on HashTable
        and Phexample (now on smalltalkHub and with working configurations'.
        spec repository: 'http://www.smalltalkhub.com/mc/Moose/Fame/main'.
        spec project: 'HashTable' with: [
            spec
                versionString: #stable;
                repository: 'http://www.smalltalkhub.com/mc/Moose/HashTable/main' ].
        spec project: 'Phexample' with: [
            spec
                versionString: #stable;
                repository: 'http://www.smalltalkhub.com/mc/Phexample/main' ].
        spec
            package: 'Fame-Core' with: [spec requires: 'HashTable'];
            package: 'Fame-Util';
            package: 'Fame-ImportExport' with: [spec requires: #('Fame-Core' ) ];
            package: 'Fame-SmalltalkBinding' with: [spec requires: #('Fame-Core' ) ];
            package: 'Fame-Example';
            package: 'Fame-Tests-Core' with: [spec requires: #('Fame-Core' 'Fame-
         Example') ].
        spec
            group: 'Core' with: #('Fame-Core' 'Fame-ImportExport' 'Fame-Util' 'Fame-
        SmalltalkBinding' );
            group: 'Tests' with: #('Fame-Tests-Core' ) ].
```

이제 프로젝트 간 의존성을 표현하면 이것이 Phexample에 의존하는 Fame-Tests-Core 패키지이며 정보의 손실이라는 사실을 잊게 된다.

그러한 정보를 유지하기 위해 PhexampleCore 예제에서 명명한 새 프로젝트를 우리의 설정에서 정의하면 PhexampleCore로부터 Fame-Tests-Core가 아래와 같이 의존적임을 표현할 수 있을 것이다.

```
spec
    project: 'PhexampleCore'
    with: [ spec
        versionString: #stable;
        Executing code before and after installation 171
        loads: #('Core');
        repository: 'http://www.smalltalkhub.com/mc/Phexample/main' ].
    ....
    'Fame-Tests-Core' with: [spec requires: #('Fame-Core' 'Fame-Example' '
        PhexampleCore' ) ].
```

설치 전후에 코드 실행하기

때때로 패키지 또는 프로젝트가 로딩되기 전이나 후에 일부 코드를 실행할 필요가 있음을 발견할 것이다. 가령 System Browser를 설치하고 있다면 기본적으로 이를 로딩한 후에 등록하는 것이 좋을 것이다. 아니면 설치 이후 어느 정도 워크스페이스를 열기를 원할 수도 있다.

Metacello는 preLoadDoIt: 과 postLoadDoIt: 메시지를 이용해 이 기능을 제공한다. 이러한 메시지에 대한 인자는 아래에 표시된 설정 클래스 상에 정의된 메서드의 선택자이다. 현재로선 단일 패키지 또는 전체 프로젝트에 대해 pre-script와 post-script를 정의할 수 있다.

위의 예제를 계속하자면,

```
ConfigurationOfCoolBrowser>>version08: spec
    <version: '0.8' imports: #('0.7-baseline')>
    spec for: #common do: [
        spec
            package: 'CoolBrowser-Core' with: [
                spec
                    file: 'CoolBrowser-Core-BobJones.20';
                    preLoadDoIt: #preloadForCore;
                    postLoadDoIt: #postloadForCore:package: ];
                    ....
            package: 'CoolBrowser-AddonsTests' with: 'CoolBrowser-AddonsTests-
    JohnLewis.1' ].

ConfigurationOfCoolBrowser>>preloadForCore
    Transcript show: 'This is the preload script. Sorry I had no better idea'.

ConfigurationOfCoolBrowser>>postloadForCore: loader package: packageSpec
    Transcript cr;
        show: '#postloadForCore executed, Loader: ', loader printString,
            ' spec: ', packageSpec printString.
    Smalltalk at: #SystemBrowser ifPresent: [:cl | cl default: (Smalltalk classNamed:
        #CoolBrowser)].
```

눈치챘겠지만 preLoadDoit: 과 postLoadDoIt: 메서드는 로딩 전이나 후에 로딩될 선택자를 수신한다. postloadForCore:package: 메서드가 두 개의 매개변수를 취한다는 사실도 눈치챌 것이다. Pre/post load 메서드는 0, 1, 또는 2개 인자를 취할 수 있다. load는 첫 번째 선택적 인자이고, 로딩된 packageSpec은 두 번째 선택적 인자다. 자신의 필요에 따라 그러한 인자들 중 원하는 것으로 선택할 수 있다.

이러한 pre/post load 메서드 스크립트는 버전 메서드 뿐만 아니라 baseline에서도 사용할 수 있다. 스크립트가 버전에 의존하는 경우 버전에 넣어둘 수 있다. 여러 버전에 걸쳐 변경되지 않는다면 정확히 같은 방식으로 baseline 메서드에 넣어둘 수도 있다.

앞에서 언급하였듯 pre/post는 패키지 수준일 수도 있지만 프로젝트 수준에서 이루어질 수도 있다. 예를 들면 아래와 같은 설정을 가질 수 있다.

```
ConfigurationOfCoolBrowser>>version08: spec
    <version: '0.8' imports: #('0.7-baseline')>
    spec for: #common do: [
        spec blessing: #release.
        spec preLoadDoIt: #preLoadForCoolBrowser.
        spec postLoadDoIt: #postLoadForCoolBrowser.
        spec
            package: 'CoolBrowser-Core' with: [
                spec
                    file: 'CoolBrowser-Core-BobJones.20';
                    preLoadDoIt: #preloadForCore;
                    postLoadDoIt: #postloadForCore:package: ];
                package: 'CoolBrowser-Tests' with: 'CoolBrowser-Tests-JohnLewis.8';
                package: 'CoolBrowser-Addons' with: 'CoolBrowser-Addons-JohnLewis.6';
                package: 'CoolBrowser-AddonsTests' with: 'CoolBrowser-AddonsTests-
                    JohnLewis.1' ].
```

이 예제에서 우리는 프로젝트 수준에서 pre/post load 스크립트를 추가하였다. 다시 말하지만 선택자는 0, 1, 또는 2개의 인자를 수신 가능하다.

플랫폼 특정적인 패키지

설정이 로딩된 플랫폼에 따라 여러 패키지가 로딩되길 원한다고 가정해보자. Cool Browser 예제를 배경으로 할 때 CoolBrowser-Platform이라는 패키지를 가질 수 있다. 여기서 추상적 클래스, API 등을 정의할 수 있겠다. 그리고 CoolBrowser-PlatformPharo, CoolBrowser-PlatformGemstone 등의 패키지를 가질 수 있다.

Metacello는 사용된 플랫폼의 패키지를 자동으로 로딩한다. 하지만 이것이 가능하려면 아래 예제에서 보이는 바와 같이 for:do: 메서드를 이용해 플랫폼 특정적인 정보를 명시할 필요가 있다. 여기서 우리는 플랫폼에 따라 여러 패키지 버전이 로딩될 것이라고 정의한다. 당신이 스크립트를 실행하는 시스템에 따라 공통 패키지에 더해 플랫폼 특정적 패키지도 로딩될 것이다.

```
ConfigurationOfCoolBrowser>>version09: spec
    <version: '0.9' imports: #('0.9-baseline')>
    spec for: #common do: [
        ...
        spec
            ...
                package: 'CoolBrowser-AddonsTests' with: 'CoolBrowser-AddonsTests-
                    JohnLewis.1' ].
    spec for: #gemstone do: [
        spec package: 'CoolBrowser-Platform' with: 'CoolBrowser-PlatformGemstone-
            BobJones.4'.].
    spec for: #pharo do: [
        spec package: 'CoolBrowser-Platform' with: 'CoolBrowser-PlatformPharo-
            JohnLewis.7'.].
```

버전 뿐만 아니라 baseline 명세 정보도 명시해야 한다.

```
ConfigurationOfCoolBrowser>>baseline09: spec
    <version: '0.9-baseline'>
    spec for: #common do: [
        spec blessing: #baseline.
        spec repository: 'http://www.example.com/CoolBrowser'.
    spec
        package: 'CoolBrowser-Core';
        package: 'CoolBrowser-Tests' with: [ spec requires: 'CoolBrowser-Core' ];
        package: 'CoolBrowser-Addons' with: [ spec requires: 'CoolBrowser-Core' ];
        package: 'CoolBrowser-AddonsTests' with: [
            spec requires: #('CoolBrowser-Addons' 'CoolBrowser-Tests' ) ].
    spec
        group: 'default' with: #('CoolBrowser-Core' 'CoolBrowser-Addons' );
        group: 'Core' with: #('CoolBrowser-Core' 'CoolBrowser-Platform' );
        group: 'Extras' with: #('CoolBrowser-Addon');
        group: 'Tests' with: #('CoolBrowser-Tests' 'CoolBrowser-AddonsTests' );
        group: 'CompleteWithoutTests' with: #('Core', 'Extras' );
        group: 'CompleteWithTests' with: #('CompleteWithoutTests', 'Tests' )].
    spec for: #gemstone do: [
        spec package: 'CoolBrowser-Platform' with: 'CoolBrowser-PlatformGemstone'].
    spec for: #pharo do: [
        spec package: 'CoolBrowser-Platform' with: 'CoolBrowser-PlatformPharo'].
```

Core 그룹에 CoolBrowser-Platform 패캐지를 추가함을 주목하라. 이 패키지는 여느 것과 마찬가지로 획일적으로 관리함을 확인할 수 있다. 따라서 상당한 유연성을 가진다. 런타임에서 CoolBrowser를 로딩하면 Metacello는 로딩이 발생하는 dialect가 무엇인지 자동으로 감지하고 해당 dialect에 특정적인 패키지를 로딩한다. for:do:는 dialects 뿐만 아니라 그들의 구체적인 버전에도 적용되는데, 아래를 예로 들겠다.

```
ConfigurationOfCoolBrowser>>baseline09: spec
    <version: '0.9-baseline'>
    spec for: #common do: [
        spec blessing: #baseline.
        spec repository: 'http://www.example.com/CoolBrowser'.
        spec
            package: 'CoolBrowser-Core';
            package: 'CoolBrowser-Tests' with: [ spec requires: 'CoolBrowser-Core' ];
            package: 'CoolBrowser-Addons' with: [ spec requires: 'CoolBrowser-Core' ];
            package: 'CoolBrowser-AddonsTests' with: [
                spec requires: #('CoolBrowser-Addons' 'CoolBrowser-Tests' ) ].
        spec
            group: 'default' with: #('CoolBrowser-Core' 'CoolBrowser-Addons' );
            group: 'Core' with: #('CoolBrowser-Core' 'CoolBrowser-Platform' );
            group: 'Extras' with: #('CoolBrowser-Addon');
            group: 'Tests' with: #('CoolBrowser-Tests' 'CoolBrowser-AddonsTests' );
            group: 'CompleteWithoutTests' with: #('Core', 'Extras' );
            group: 'CompleteWithTests' with: #('CompleteWithoutTests', 'Tests' )].
    spec for: #gemstone do: [
        spec package: 'CoolBrowser-Platform' with: 'CoolBrowser-PlatformGemstone'].
    spec for: #pharo do: [
        spec package: 'CoolBrowser-Platform' with: 'CoolBrowser-PlatformPharo'].
```

로딩 순서. 당신이 만일 플랫폼 속성이 (#common #squeakCommon #pharo #'pharo2.x' #'pharo2.0.x') (이 정보는 ConfigurationOf project attributes를 실행해 얻을 수 있다) 위치한 시스템에 있고, #common, #pharo, #pharo2.0.x 와 같이 세 개의 section을 명시하였다면,

이러한 section들은 차례로 로딩될 것이다.

```
ConfigurationOfCoolBrowser>>baseline09: spec
    <version: '0.9-baseline'>
    spec for: #common do: [
    spec blessing: #baseline.
    spec repository: 'http://www.example.com/CoolBrowser'.
    spec
        package: 'CoolBrowser-Core';
        package: 'CoolBrowser-Tests' with: [ spec requires: 'CoolBrowser-Core' ];
        package: 'CoolBrowser-Addons' with: [ spec requires: 'CoolBrowser-Core' ];
        package: 'CoolBrowser-AddonsTests' with: [
            spec requires: #('CoolBrowser-Addons' 'CoolBrowser-Tests' ) ].
        spec
        group: 'default' with: #('CoolBrowser-Core' 'CoolBrowser-Addons' );
        group: 'Core' with: #('CoolBrowser-Core' 'CoolBrowser-Platform' );
        group: 'Extras' with: #('CoolBrowser-Addon');
        group: 'Tests' with: #('CoolBrowser-Tests' 'CoolBrowser-AddonsTests' );
        group: 'CompleteWithoutTests' with: #('Core', 'Extras' );
        group: 'CompleteWithTests' with: #('CompleteWithoutTests', 'Tests' )].
    spec for: #gemstone do: [
        spec package: 'CoolBrowser-Platform' with: 'CoolBrowser-PlatformGemstone'].
    spec for: #pharo do: [
        spec package: 'CoolBrowser-Platform' with: 'CoolBrowser-PlatformPharo'].
    spec for: #pharo2.0.x do: [
        spec package: 'CoolBrowser-Addons' with: 'CoolBrowser-Core20'].
```

마지막으로 for:do: 메서드는 플랫폼 특정적 패키지를 명시할 때 뿐만 아니라 여러 dialect와 관련된 것이라면 어디든 사용 가능함을 주목하라. 설정으로부터 원하는 것은 무엇이든 해당 블록에 넣을 수 있다. 그렇기 때문에 각 dialect마다 그룹, 패키지, 저장소 등을 정의, 변경, 맞춤설정할 수 있는 것이다. 아래를 일례로 들어보겠다.

```
ConfigurationOfCoolBrowser>>baseline010: spec
    <version: '0.10-baseline'>
    spec for: #common do: [
        spec blessing: #baseline.].
    spec for: #pharo do: [
        spec repository: 'http://www.pharo.com/CoolBrowser'.
        spec
            ...
        spec
            group: 'default' with: #('CoolBrowser-Core' 'CoolBrowser-Addons' );
            group: 'Core' with: #('CoolBrowser-Core' 'CoolBrowser-Platform' );
            group: 'Extras' with: #('CoolBrowser-Addon');
            group: 'Tests' with: #('CoolBrowser-Tests' 'CoolBrowser-AddonsTests' );
            group: 'CompleteWithoutTests' with: #('Core', 'Extras' );
            group: 'CompleteWithTests' with: #('CompleteWithoutTests', 'Tests' )].
    spec for: #gemstone do: [
        spec repository: 'http://www.gemstone.com/CoolBrowser'.
        spec
            package: 'CoolBrowser-Core';
            package: 'CoolBrowser-Tests' with: [ spec requires: 'CoolBrowser-Core' ];
        spec
            group: 'default' with: #('CoolBrowser-Core' 'CoolBrowser-Addons' );
            group: 'Core' with: #('CoolBrowser-Core' 'CoolBrowser-Platform' )].
```

이 예제에서 Pharo의 경우 Gemstone과 다른 저장소를 사용한다. 하지만 의무적인 것은 아닌데, 두 가지 모두 저장소는 같지만 버전, post/pre code 실행, 의존성 등에 차이가 있기

때문이다.

게다가 addons와 테스트는 Gemstone에서 이용할 수 있기 때문에 그러한 패키지와 그룹은 포함되지 않는다. 따라서 for:#common:do: 내부에서 우리가 한 일은 특정 dialect에 대한 다른 for:do: 내부에서도 실행할 수 있겠다.

중요한 개발: symbolic 버전

대규모로 발전하는 애플리케이션이라면 특정 버전과 함께 어떤 설정의 버전을 사용해야 할지 확인하기란 항상 까다롭다. ConfigurationOfOmniBrowser가 이런 문제의 일례를 보여주는데, Pharo1.0 one-click 이미지에선 1.1.3 버전이 사용되는데 버전 1.1.3은 Pharo1.2에서 로딩할 수 없고, 버전 1.1.5는 Pharo1.1용, 버전 1.2.3은 Pharo1.2용으로 Pharo1.0에선 로딩할 수 없으며, Pharo2.0에 이용할 수 있는 버전은 없다. Metacello를 이용해 무엇을 할 수 있는지 살펴보기 위해 이 예제로 설명하겠다.

Metacello는 기존 리터럴 버전(1.1.3, 1.1.5, 1.2.3)과 관련해 버전을 설명하기 위해 symbolic 버전의 개념을 도입한다. symbolic 버전은 symbolicVersion: pragma를 이용해 명시된다. 여기서는 Pharo의 버전마다 OmniBrowser 에 대한 안정된 버전을 정의하였다.

```
OmniBrowser>>stable: spec
    <symbolicVersion: #stable>
    spec for: #'pharo1.0.x' version: '1.1.3'.
    spec for: #'pharo1.1.x' version: '1.1.5'.
    spec for: #'pharo1.2.x' version: '1.2.3'.
```

symbolic 버전은 리터럴 버전을 사용할 수 있는 곳이라면 어디든 사용 가능하다. 아래와 같은 load 표현식부터,

```
(ConfigurationOfXMLParser project version: \#stable) load
(ConfigurationOfXMLParser project version: \#stable) load: 'Tests'
```

baseline 버전의 프로젝트 참조까지 가능하다.

```
baseline10: spec
    <version: '1.0-baseline'>
    spec for: #squeakCommon do: [
        spec blessing: #baseline.
        spec repository: 'http://seaside.gemstone.com/ss/GLASSClient'.
    spec
        project: 'OmniBrowser' with: [
        spec
            className: 'OmniBrowser';
            versionString: #stable;
            repository: 'http://www.squeaksource.com/MetacelloRepository' ].
    spec
        package: 'OB-SUnitGUI' with: [
            spec requires: #('OmniBrowser') ];
        package: 'GemTools-Client' with: [
            spec requires: #('OB-SUnitGUI') ];
        package: 'GemTools-Platform' with: [
            spec requires: #('GemTools-Client') ]].
```

여기서 #stable은 (어리석게도) baseline을 로딩하려 할 경우 얻게 될 최첨단 로딩(bleeding edge loading) 행위를 오버라이드함을 주목하라 (baseline을 로딩 시 최첨단 버전을 로딩함을 기억하라). 여기서 우리는 자신의 플랫폼에 대한 OmniBrowser의 (최신 버전이 아니라) 안정적 버전이 확실히 로딩되도록 한다. 다음 절에서는 이와 다른 symbolic 버전을 다루겠다.

표준 symbolic 버전

몇 가지 표준 symbolic 버전이 정의된다.
stable. 특정 플랫폼과 그러한 플랫폼의 버전에 대한 안정적인 리터럴 버전을 명시하는 symbolic 버전. 안정적 버전은 로딩에 사용되어야 하는 버전이다. bleedingEdge 버전을 제외하고 (기본적으로 사전 정의된 버전) stable 또는 development 버전 정보를 추가하도록 자신의 설정을 편집해야 할 것이다. 나는 플랫폼에 승인된 버전을 원한다. 패키지의 안정적 버전에 의존할 경우 패키지가 변경될 수 없다는 의미는 아니기 때문에 주의를 기울여야 할 때다. 사실 패키지 구현자가 자신의 시스템과 호환이 되지 않는 새 버전을 생성할 수도 있다.
development. 개발 시 사용할 리터럴 버전을 명시하는 symbolic 버전 (예: development를 사용 시). 일반적으로 development 버전은 개발자들이 bleedingEdge에서 stable로 프로젝트를 전환하는 등의 배포 전 (pre-release) 활동을 관리하기 위해 개발자들에 의해 사용된다. 나는 플랫폼에 승인된 버전을 원하지만 개발 모드였으면 좋겠다 라는 의미다.
bleedingEdge. 최신 mcz 파일과 프로젝트 버전을 명시하는 symbolic 버전. 기본적으로 bleedingEdge symbolic 버전은 이용 가능한 최신 baseline 버전으로서 정의된다. bleedingEdge에 대한 기본 명세는 모든 프로젝트에 대해 정의된다. bleedingEdge 버전은 주로 자신들이 어떤 일을 하는지 아는 개발자들을 위한 것이다. 올바로 bleedingEdge 버전이 기능하는지는 제쳐두고 그것이 로딩될 것인지도 보장할 수 없다. 나는 가장 최근에 공개된 파일을 원한다.

symbolicVersion: pragma 의 형식으로 symbolic 버전을 명시할 때는 symbolic 버전의 stable에 대한 아래의 정의처럼 또 다른 symbolic 버전을 사용해도 좋다.

```
stable: spec
    <symbolicVersion: #stable>
    spec for: #gemstone version: '1.5'.
    spec for: #'squeak' version: '1.4'.
    spec for: #'pharo1.0.x' version: '1.5'.
    spec for: #'pharo1.1.x' version: '1.5'.
    spec for: #'pharo1.2.x' version: #development.
```

아니면 development symbolic 버전의 정의에서처럼 특수 symbolic 버전 notDefined: 를 사용하는 방법도 있겠다.

```
development: spec
    <symbolicVersion: #development>
    spec for: #common version: #notDefined.
    spec for: #'pharo1.1.x' version: '1.6'.
    spec for: #'pharo1.2.x' version: '1.6'.
```

여기서는 common 태그에 대한 버전이 없다고 나타난다. notDefined로 해결되는 symbolic 버전을 이용 시 MetacelloSymbolicVersionNotDefinedError가 시그널링된다.

개발 symbolic 버전의 경우 원하는 버전은 무엇이든 사용할 수 있다 (다른 symbolic 버전도 포함). 아래 코드가 보여주듯이 특정 버전, baseline(baseline이 명시한 최신 버전을 로딩시킬), 또는 안정적 버전을 명시할 수 있다.

```
development: spec
    <symbolicVersion: #'development'>
    spec for: #'common' version: '1.1'
development: spec
    <symbolicVersion: #'development'>
    spec for: #'common' version: '1.1-baseline'
development: spec
    <symbolicVersion: #'development'>
    spec for: #'common' version: #stable
```

경고. 안정적이란 말은 오해를 불러일으키기 쉬운 용어이므로 주의를 기울일 것을 다시 요청한다. 당신이 의존하는 시스템의 개발 시 다른 안정적 버전을 가리키도록 '안정적'이란 의미를 변경할 수 있기 때문에 당신이 정확히 같은 버전을 항상 로딩하게 될 것이란 뜻은 아니며, 그러한 안정적 버전이 당신의 코드와 호환되지 않을 가능성이 있다는 의미다. 따라서 자신의 코드를 배포할 때는 다른 변경 내용의 영향을 받지 않도록 특정 버전을 사용해야 한다.

프로젝트 Blessing과 로딩

패키지 또는 프로젝트는 가령 개발, 알파, 베타, 릴리즈와 같은 라이프 사이클이나 소프트웨어 개발 과정 동안에 여러 단계를 거친다. 때로는 프로젝트의 상태를 칭하고 싶을 때가 있다.

blessing은 load logic에 의해 고려된다. 아래 표현식의 결과가

```
ConfigurationOfCoolBrowser project latestVersion.
```

이렇듯이 항상 마지막 버전이 되는것은 아니다. 그 이유는 latestVersion은 blessing이 #development, #broke, #blessing에 해당하지 않는 최신 버전을 응답하기 때문이다. 예를 들어, 최신 #development 버전을 찾기 위해서는 아래 표현식을 실행해야 한다.

```
ConfigurationOfCoolBrowser project latestVersion: \#development.
```

그럼에도 불구하고 아래와 같이 lastVersion을 이용해 blessing과 상관없이 마지막 버전을 얻을 수도 있다.

```
ConfigurationOfCoolBrowser project lastVersion.
```

일반적으로 불안정한 버전에는 #development blessing를 사용해야 한다. 버전이 안정화되고 나면 다른 blessing을 적용시켜야 한다.

아래 표현식은 최신 #baseline 버전에 대한 모든 패키지의 최신 버전을 로딩할 것이다.

```
(ConfigurationOfCoolBrowser project latestVersion: \#baseline) load.
```

최신 #baseline 버전은 가장 최신의 프로젝트 구조를 반영해야 하기 때문에 앞의 표현식을 이용하면 프로젝트의 완전한 bleeding edge 버전을 로딩한다.

힌트

일부 패턴은 Metacello와 작업할 때 생겨나기도 한다. 한 가지 예를 들어보겠다. Baseline 버전을 생성하고 baseline 내 모든 프로젝트에 대해 #stable 버전을 사용하라. 리터럴 버전에서 명시적 버전을 이용하면, 함께 작업하는 것으로 알려진 프로젝트 집합에 대해 명시적인 반복 가능 명세를 얻을 수 있다.

예로, pharo 1.2.2-baseline는 아래와 같은 명세를 포함할 것이다.

```
spec
    project: 'OB Dev' with: [
        spec
            className: 'ConfigurationOfOmniBrowser';
            versionString: #stable;
            ...];
    project: 'ScriptManager' with: [
        spec
            className: 'ConfigurationOfScriptManager';
            versionString: #stable;
            ...];
    project: 'Shout' with: [
        spec
            className: 'ConfigurationOfShout';
            versionString: #stable;
            ...];
        ....].
```

Pharo 1.2.2-baseline를 로딩 시 로딩되어야 할 프로젝트마다 #stable 버전을 야기할 것이지만…시간이 지나면서 #stable 버전은 변경되고, 패키지들 간 비호환성이 발견되기 시작함을 기억하라. #stable 버전을 이용하면 #bleedingEdge를 이용할 때보다 더 나은 결과를 야기할 것인데, #stable 버전은 작동하는 것으로 알려져 있기 때문이다.

Pharo 1.2.2(리터럴 버전)은 아래와 같은 명세를 가질 것이다.

```
spec
    project: 'OB Dev' with: '1.2.4';
    project: 'ScriptManager' with: '1.2';
    project: 'Shout' with: '1.2.2';
    ....].
```

이제 이러한 버전들은 서로 작업하는 것으로 알려져 있음을 확신할 수 있을 것이다 (단체로 테스트를 합격한 셈이다). 향후 5년 간은 Pharo 1.2.2를 로딩하면서 매번 정확히 같은 패키지를 얻을 것이지만, #stable 버전은 시간이 지나면 없어질지도 모른다.

이제 막 PharoCore1.2 이미지를 가져와 Pharo dev 코드를 로딩하고자 한다면 Pharo의 #stable 버전을 로딩해야 한다 (어쩌면 오늘은 1.2.2를, 내일은 1.2.3을). 누군가 작업 중인 환경을 중복하고자 한다면 그들에게 Pharo의 버전을 묻고 나서 버그와 같은 것들을 복제하기 위해 명시적 버전을 로딩해야 한다.

패키지 구조 변경은 어떻게 처리하는가?

Pharo13과 Pharo14에 애플리케이션을 생성하고자 하는데 애플리케이션을 변경했거나 패키지가 베이스 시스템으로 통합되었다는 이유로 애플리케이션이 버전 하나 당 패키지를 하나만 갖고 있다고 가정해보자.

이에 대한 해결책은 아래와 같이 의존성을 정의하고 symbolic 태그를 마커(marker)로서 사용하는 것이다.

```
spec for: #'pharo' do: [
    spec package: 'that depends upon zinc' with: [
        "the common required packages for your package"
    ].

spec for: #'pharo1.3.x' do: [
    spec project: 'Zinc' with: [
        spec
            className: 'ConfigurationOfZinc';
            versionString: #'stable';
            repository: 'http://www.squeaksource.com/MetacelloRepository' ].
    spec package: 'that depends upon zinc' with: [
        spec requires: #('Zinc') ].
].
```

자신의 baseline에 stable 버전을 사용할 경우 자신의 버전 명세에 특별한 일을 할 필요가 없다.

로드 타입

Metacello는 자체의 "로드 타입(load type)"을 통해 패키지가 로딩되는 방식을 명시하도록 해준다. 이 문서를 작성하는 시점에선 두 가지의 로드 타입, atomic과 linear가 가능하다.

Atomic 로딩은 패키지들이 따로 로딩될 수 없도록 분할(partitioned)되었을 때 사용된다. 각 패키지로부터 정의는 Monticello 패키지 로더에 의해 하나의 거대한 로드로 모인다. 클래스 측 initialize 메서드와 pre/post 코드 실행은 개별적 패키지가 아니라 패키지 전체 집합을 대상으로 실행된다.

Linear 로드를 사용할 경우 각 패키지는 순서대로 로딩된다. 클래스 측 initialize 메서드와 pre/post 코드 실행은 특정 패키지가 로딩된 전 또는 후에 실행된다.

의존성을 관리한다고 해서 순서대로 패키지가 로딩될 것을 의미하지는 않음을 인식하는 것이 중요하다. 패키지 A가 패키지 B를 의존한다고 해서 B가 A보다 먼저 로딩될 것이란 의미는 아니다. 그저 A를 로딩한 후에 B를 로딩하길 원할 경우 그렇게 진행되도록 보장할 뿐이다.

이와 관련된 문제는 메서드 오버라이드와 관련해 발생한다. 패키지가 다른 패키지의 메서드를 오버라이드할 경우 순서는 보존되지 않는데, 로딩될 순서를 확신할 수 없고 그에 따라 메서드의 어떤 버전이 결국 로딩될 것인지 확신할 수 없기 때문에 문제가 된다.

Atomic 로딩을 이용하면 패키지 순서가 손실되고 앞서 언급한 문제도 발생한다. 하지만 linear 모드를 이용하면 각 패키지가 순서대로 로딩된다. 뿐만 아니라 메서드 오버라이드도 보존된다.

Lienar 모드에서 발생 가능한 문제를 설명하기 위해 가령 프로젝트 A가 두 개의 프로젝트, B와 C에 의존한다고 가정해보자. B는 프로젝트 D 버전 1.1을 의존하고, C는 프로젝트 D 버전 1.2(동일한 프로젝트의 다른 버전)에 의존한다. 첫 번째 질문은, 'A는 결국 D의 어떤 버전을 갖게 될까'가 된다. 기본적으로(이는 project 메서드에서 operator: 메서드를 이용해 변경 가능하다) Metacello는 최신 버전, 즉 버전 1.2를 로딩할 것이다.

하지만 로드 타입과 관련해, atomic 로딩에서는 1.2 버전만 로딩된다. Linear 로딩에서는 두 가지 버전 모두 (의존성 순서에 따라) 로딩이 가능하지만 1.2가 결국 로딩될 것이다. 다시 말해

1.1 버전이 먼저 로딩된 다음 1.2가 로딩될지 모른다는 의미다. 이조차도 문제가 되기도 하는데, 오래된 버전의 패키지나 프로젝트가 우리가 사용 중인 Pharo 이미지에서 로딩되지 않을 수 있기 때문이다.

이러한 연유로 하여 linear가 기본 모드가 된다. 사용자는 특별한 경우에, 그리고 전적으로 확신할 때 atomic 로딩을 사용해야 한다.

마지막으로 명시적으로 로드 타입을 설정하길 원한다면 project 메서드를 통해야 할 것인데, 아래에 예를 들어보겠다.

```
ConfigurationOfCoolToolSet >>project
    ^ project ifNil: [ | constructor |
        "Bootstrap Metacello if it is not already loaded"
        self class ensureMetacello.
        "Construct Metacello project"
        constructor := (Smalltalk at: #MetacelloVersionConstructor) on: self.
        project := constructor project.
        project loadType: #linear. '"or #atomic"'
        project ]
```

조건부 로딩

사용자는 프로젝트를 로딩 시 주로 특정 상태, 가령 이미지에 다른 특정 패키지의 존재 유무에 따라 특정 패키지를 로딩할 것인지 말 것인지 결정하길 원한다. 자신의 이미지에 Seaside를 로딩하고 싶다고 치자. Seaside는 OmniBrowser에 의존하는 툴을 갖고 있으며, 웹 서버의 인스턴스를 관리하는 데에 사용된다. 이렇게 작은 툴로 할 수 있는 일은 코드를 이용해 실행할 수도 있다. 그러한 툴을 로딩하기 위해선 OmniBrowser가 필요하다. 하지만 다른 사용자들에겐 해당 패키지가 필요하지 않을지도 모른다. 대안적 방법으로 여러 그룹, 즉 그러한 패키지를 포함하는 그룹과 포함하지 않은 그룹을 제공하는 방법이 있다. 문제는 최종 사용자가 이를 인지하여 여러 상황에서 여러 그룹을 로딩해야 한다는 데에 있다. 조건부 로딩을 이용하면 OmniBrowser가 이미지에 있을 때에만 Seaside 툴을 로딩할 수 있다. 이는 Metacello가 자동으로 실행할 것이므로 특정 그룹을 명시적으로 로딩할 필요가 없다.

우리의 CoolToolSet가 더 많은 기능을 제공하기 시작한다고 가정하자. 먼저 core를 두 개의 패키지, 'CoolToolSet-Core'와 'coolToolSet-CB'로 나눈다. CoolBrowser는 한 이미지에만 존재가 가능하며 다른 이미지에선 존재해선 안 된다. 우리는 CoolBrowser가 있을 경우에만 기본 값으로 'CoolToolSet-CB' 패키지를 로딩하길 원한다.

Metacello에서 앞서 언급한 조건문(conditionals)은 앞 절에서 살펴본 프로젝트 속성을 이용해 확보할 수 있다. 이러한 조건문은 project 메서드에 정의되는데, 아래를 예로 들어보겠다.

```
ConfigurationOfCoolBrowser >>project
    | |
    ^ project ifNil: [ | constructor |
        "Bootstrap Metacello if it is not already loaded"
        self class ensureMetacello.
        "Construct Metacello project"
        constructor := (Smalltalk at: #MetacelloVersionConstructor) on: self.
        project := constructor project.
        projectAttributes := ((Smalltalk at: #CBNode ifAbsent: []) == nil
            ifTrue: [ #( #'CBNotPresent' ) ]
            ifFalse: [ #( #'CBPresent' ) ]).
        project projectAttributes: projectAttributes.
        project loadType: #linear.
        project ]
```

코드에서 볼 수 있듯이 우리는 CBNode 클래스(CoolBrowser의 클래스)가 존재하는지, 그리고 우리가 설정한 특정 프로젝트 속성에 의존하는지 확인한다. 자신만의 조건문을 정의하고 자신이 원하는 프로젝트 속성 수량을 설정하도록 (속성의 배열을 정의할 수 있다) 허용할 정도로 유연하다. 이제 문제는 이러한 프로젝트 속성을 어떻게 사용하느냐가 된다. 아래 baseline에서 예제를 살펴보겠다.

```
ConfigurationOfCoolToolSet >>baseline02: spec
    <version: '0.2-baseline'>
    spec for: #common do: [
    spec blessing: #baseline.
    spec repository: 'http://www.example.com/CoolToolSet'.
    spec project: 'CoolBrowser default' with: [
        spec
            className: 'ConfigurationOfCoolBrowser';
            versionString: '1.0';
            loads: #('default' );
            file: 'CoolBrowser-Metacello';
            repository: 'http://www.example.com/CoolBrowser' ];
        project: 'CoolBrowser Tests'
            copyFrom: 'CoolBrowser default'
            with: [ spec loads: #('Tests').].
    spec
        package: 'CoolToolSet-Core';
        package: 'CoolToolSet-Tests' with: [
            spec requires: #('CoolToolSet-Core') ];
        package: 'CoolToolSet-CB';

    spec for: #CBPresent do: [
        spec
            group: 'default' with: #('CoolToolSet-CB' )
            yourself ].

    spec for: #CBNotPresent do: [
        spec
            package: 'CoolToolSet-CB' with: [ spec requires: 'CoolBrowser default'];
                yourself ].
        ].
```

프로젝트 속성은 기존 메서드 for:do: 를 통해 사용됨을 눈치챘을 것이다. 해당 메서드 내에서는 그룹, 의존성 등을 정의하는 등 당신이 원하는 대로 할 수 있다. 우리 예제의 경우는, CoolBrowser가 존재한다면 기본 그룹으로 'CoolToolSet-CB'를 추가하기만 하면 된다. CoolBrowser가 없다면 'CoolBrowser default'가 'CoolToolSet-CB'에 대한 의존성에 추가된다.

이번 사례에서 우리는 기본 그룹으로 추가하길 원치 않으므로 삼가한다. 사용자가 원한다면 해당 패키지도 명시적으로 로딩할 수 있겠다.

다시 말하지만 for:do: 안에서 원하는 것을 마음껏 실행할 수 있음을 주목하라.

프로젝트 버전 속성

설정에는 여러 개의 선택적 속성이 있는데, 작성자, 설명, blessing, 타임스탬프를 들 수 있겠다. 우리 프로젝트의 새 0.7 버전으로 예를 들어보겠다.

```
ConfigurationOfCoolBrowser>>version07: spec
    <version: '0.7' imports: #('0.7-baseline')>
    spec for: #common do: [
        spec blessing: #release.
        spec description: 'In this release...'.
        spec author: 'JohnLewis'.
        spec timestamp: '10/12/2009 09:26'.
        spec
            package: 'CoolBrowser-Core' with: 'CoolBrowser-Core-BobJones.20';
            package: 'CoolBrowser-Tests' with: 'CoolBrowser-Tests-JohnLewis.8';
            package: 'CoolBrowser-Addons' with: 'CoolBrowser-Addons-JohnLewis.6';
            package: 'CoolBrowser-AddonsTests' with: 'CoolBrowser-AddonsTests-JohnLewis.1'
        ].
```

각 속성을 상세히 설명해보겠다.

Description(설명): 버전에 대한 텍스트 설명. 버그 수정이나 새 기능 리스트, changelog 등을 포함할 수 있다.

Author(작성자): 버전을 생성한 작성자의 이름. OB-Metacello 툴이나 MetacelloToolbox를 사용할 시 작성자 필드는 이미지에 정의된 현재 작성자를 반영하도록 자동으로 업데이트된다.

TimeStamp: 버전이 완성된 당시의 날짜와 시간. OB-Metacello 툴이나 MetacelloToolbox를 사용 시 타임스탬프 필드는 현재 날짜와 시간을 반영하도록 자동으로 업데이트 된다. 타임스태프는 String이어야 한다는 사실을 주목한다.

이번 장을 마무리하기 위해 해당 정보를 질의하는 방법을 보여주고자 한다. 이는 Metacello 버전에 정의한 정보 대부분을 질의할 수 있음을 보여준다. 예를 들어서 아래 표현식을 평가할 수 있겠다.

```
(ConfigurationOfCoolBrowser project version: '0.7') blessing.
(ConfigurationOfCoolBrowser project version: '0.7') description.
(ConfigurationOfCoolBrowser project version: '0.7') author.
(ConfigurationOfCoolBrowser project version: '0.7') timestamp.
```

요약

Metacello는 Pharo에서 중요한 부분이다. 이는 당신의 프로젝트를 조정하도록 해준다. 또한 새 버전으로 그리고 어떤 패키지를 위해 이동하길 원할 때를 제어하도록 해준다. 이는 중요한 아키텍처 뼈대이다.

Metacello Memento

```
ConfigurationOfCoolToolSet>>baseline06: spec "could be called differently just a convention"
    <version: '0.6-baseline'> "Convention. Used in the version: method"
    spec for: #common do: [ "#common/#pharo/#gemstone/#pharo'1.4'"
        spec blessing: #baseline. "Important: identifies a baseline"
        spec repository: 'http://www.example.com/CoolToolSet'.

        "When we depend on other projects"
        spec project: 'CoolBrowser default' with: [
            spec
                className: 'ConfigurationOfCoolBrowser'; "Optional if convention followed"
           versionString: #bleedingEdge; "Optional. Could be #stable/#bleedingEdge/specific version"
                loads: #('default'); "which packages or groups to load"
                file: 'CoolBrowser-Metacello'; "Optional when same as class name"
                repository: 'http://www.example.com/CoolBrowser' ];
            project: 'CoolBrowser Tests'
                copyFrom: 'CoolBrowser default' "Just to reuse information"
                with: [ spec loads: #('Tests').]. "Just to reuse information"

        "Our internal package dependencies"
        spec
            package: 'CoolToolSet-Core';
            package: 'CoolToolSet-Tests' with: [ spec requires: #('CoolToolSet-Core') ];
            package: 'CoolBrowser-Addons' with: [ spec requires: 'CoolBrowser-Core' ] ;
            package: 'CoolBrowser-AddonsTests' with: [
                spec requires: #('CoolBrowser-Addons' 'CoolBrowser-Tests' ) ].

        spec
            group: 'default' with: #('CoolBrowser-Core' 'CoolBrowser-Addons');
            group: 'Core' with: #('CoolBrowser-Core');
            group: 'Extras' with: #('CoolBrowser-Addon');
            group: 'Tests' with: #('CoolBrowser-Tests' 'CoolBrowser-AddonsTests');
            group: 'CompleteWithoutTests' with: #('Core' 'Extras');
            group: 'CompleteWithTests' with: #('CompleteWithoutTests' 'Tests')
    ].
```

```
ConfigurationOfCoolBrowser>>version07: spec "could be called differently just a convention"
    <version: '0.7' imports: #('0.6-baseline')> "Convention. No baseline so this is version"
                                                "do not import baseline from other baselines"
    spec for: #common do: [ "#common/#pharo/#gemstone/#pharo'1.4'"
      spec blessing: #release. "Required #development/#release: release means that it will not change
            anymore"
        spec description: 'In this release .....'.
        spec author: 'JohnLewis'.
        spec timestamp: '10/12/2009 09:26'.
        spec
            package: 'CoolBrowser-Core' with: 'CoolBrowser-Core-BobJones.20';
            package: 'CoolBrowser-Tests' with: 'CoolBrowser-Tests-JohnLewis.8';
            package: 'CoolBrowser-Addons' with: 'CoolBrowser-Addons-JohnLewis.6' ;
            package: 'CoolBrowser-AddonsTests' with: 'CoolBrowser-AddonsTests-JohnLewis.1']
```

```
ConfigurationOfGemToolsExample>>development: spec "note that the selector can be anything"
    <symbolicVersion: #development> "#stable/#development/#bleedingEdge"
    spec for: #common version: '1.0'. "'1.0' is the version of your development version"
    "#common or your platform attributes: #gemstone, #pharo, or #'pharo1.4'"
```

```
ConfigurationOfGemToolsExample>>baseline10: spec
    <version: '1.0-baseline'>
    spec for: #common do: [
        spec blessing: #'baseline'. "required see above"
        spec repository: 'http://seaside.gemstone.com/ss/GLASSClient'.
        spec
            project: 'FFI' with: [
                spec
                    className: 'ConfigurationOfFFI';
             versionString: #bleedingEdge; "Optional. #stable/#development/#bleedingEdge/specificversion"
                    repository: 'http://www.squeaksource.com/MetacelloRepository' ];
            project: 'OmniBrowser' with: [
                spec
                    className: 'ConfigurationOfOmniBrowser';
             versionString: #stable; "Optional. #stable/#development/#bleedingEdge/specificversion"
                    repository: 'http://www.squeaksource.com/MetacelloRepository' ];
            project: 'Shout' with: [
                spec
                    className: 'ConfigurationOfShout';
                    versionString: #stable;
                    repository: 'http://www.squeaksource.com/MetacelloRepository' ];
            project: 'HelpSystem' with: [
                spec
                    className: 'ConfigurationOfHelpSystem';
                    versionString: #stable;
                    repository: 'http://www.squeaksource.com/MetacelloRepository'].
        spec
            package: 'OB-SUnitGUI' with: [spec requires: #('OmniBrowser')];
        package: 'GemTools-Client' with: [ spec requires: #('OmniBrowser' 'FFI' 'Shout' 'OB-SUnitGUI' ).];
            package: 'GemTools-Platform' with: [ spec requires: #('GemTools-Client' ). ];
            package: 'GemTools-Help' with: [
        spec requires: #('HelpSystem' 'GemTools-Client' ). ].
        spec group: 'default' with: #('OB-SUnitGUI' 'GemTools-Client' 'GemTools-Platform' 'GemTools-Help')]. 'GemTools-Help')
```

```
ConfigurationOfGemToolsExample>>version10: spec
    <version: '1.0' imports: #('1.0-baseline' )>
    spec for: #common do: [
        spec blessing: #development.
        spec description: 'initial development version'.
        spec author: 'dkh'.
        spec timestamp: '1/12/2011 12:29'.
    spec
        project: 'FFI' with: '1.2';
        project: 'OmniBrowser' with: #stable;
        project: 'Shout' with: #stable;
        project: 'HelpSystem' with: #stable.
    spec
        package: 'OB-SUnitGUI' with: 'OB-SUnitGUI-dkh.52';
        package: 'GemTools-Client' with: 'GemTools-Client-NorbertHartl.544';
        package: 'GemTools-Platform' with: 'GemTools-Platform.pharo10beta-dkh.5';
        package: 'GemTools-Help' with: 'GemTools-Help-DaleHenrichs.24'. ].
```

Loading. load, load:. load 메서드는 기본 그룹을 로딩하는데, 기본 그룹이 정의되지 않은 경우

모든 패키지가 로딩된다. load: 메서드는 패캐지명, 프로젝트명, 그룹명, 또는 그러한 항목들의 컬렉션명을 매개변수로 취한다.

```
(ConfigurationOfCoolBrowser project version: '0.1') load.
(ConfigurationOfCoolBrowser project version: '0.2') load: \{'CBrowser-Core' . 'CBrowserAddons'\}.
```

Debugging(디버깅). Record, record: loadDirectives. record 메시지는 기본 그룹을 기록하는데, 특정 항목의 그룹을 원할 경우 load처럼 record: 를 사용하면 된다.

```
((ConfigurationOfCoolBrowser project version: '0.2') record:
    { 'CoolBrowser-Core' .
    'CoolBrowser-Addons' }) loadDirective.
```

모든 파일 버전으로 재귀적으로 접근하고자 한다면 아래 예제와 같이 packageDirectives: 를 사용하라.

```
| pkgs loader |
loader := ((Smalltalk globals at: #ConfigurationOfMoose) project version: 'default')
    ignoreImage: true;
    record.

pkgs := OrderedCollection new.
loader loadDirective packageDirectivesDo: [:directive |pkgs add: directive spec file ].
pkgs.
```

권장하는 개발 과정. metacello를 이용해 아래와 같은 개발 단계를 권한다.

```
Baseline      "first we define a baseline"
Version development     "Then a version tagged as development"
Validate the map   "Once it is validated and the project status arrives to the desired status"
Version release     "We are ready to tag a version as release"
Version development     "Since development continue we create a new version"
{\dots}           "Tagged as development. It will be tagged as release and so on"
Baseline        "When architecture or structure changes, a new baseline will appear"
Version development       "and the story will continue"
Version release
```

제 III 부

Frameworks

제 10 장

Glamour

***Tudor Girba** 참여 (tudor@tudorgirba.com)*

브라우저는 복잡한 시스템이나 모델을 이해하는 데 결정적인 도구다. 또한 특정 영역을 탐색하고 상호작용하기 위한 툴이다. 각 문제 영역에는 기본 요소를 분석하고 해석하도록 돕기 위해 생성된 브라우저들이 많이 구비되어 있다. 이러한 브라우저와 관련된 문제는 보통 처음부터 (재) 작성되기 때문에 생성 시 비용이 많이 들고 유지하기가 힘들다. 많은 프레임워크들이 대개는 사용자 인터페이스의 개발을 돕기 위해 존재하지만 브라우저의 생성을 간편화하는 데에는 제한된 지원만 제공하고 있다.

Glamour는 브라우저의 탐색 흐름을 설명하는 데 집중된 프레임워크다. 그 선언 언어 덕분에 Glamour는 데이터에 대한 새 브라우저를 재빠르게 정의하도록 해준다.

이번 장에서는 Glamour 프레임워크의 개요를 살펴보기 위해 예제로 소개한 브라우저의 생성을 먼저 상세히 살펴보고자 한다. 그 다음으로 세부적인 내용에 집중하겠다.

설치 및 첫 번째 브라우저

자신의 Pharo 이미지에 Glamour를 설치하기 위해서는 아래 코드를 실행하라.

```
Gofer new
    smalltalkhubUser: 'Moose' project: 'Glamour';
    package: 'ConfigurationOfGlamour';
    load.
(Smalltalk at: #ConfigurationOfGlamour) perform: #loadDefault.
```

Glamour가 설치되었으니 Glamour의 선언 언어를 이용해 첫 브라우저를 빌드할 준비가 되었다. Apple의 Finder와 같은 파일 브라우저를 빌드하는 건 어떨까? 이 브라우저는 Miller Columns 브라우징 기법을 이용해 일련의 열로 계층 구조적 요소를 표시하여 빌드된다. 이러한 브라우저에서는 항상 앞 열에서 선택한 요소의 내용을 그 다음 열에 반영하고, 파일을 열면 첫 열의 내용이 선택되도록 되어 있는 것을 원리로 한다.

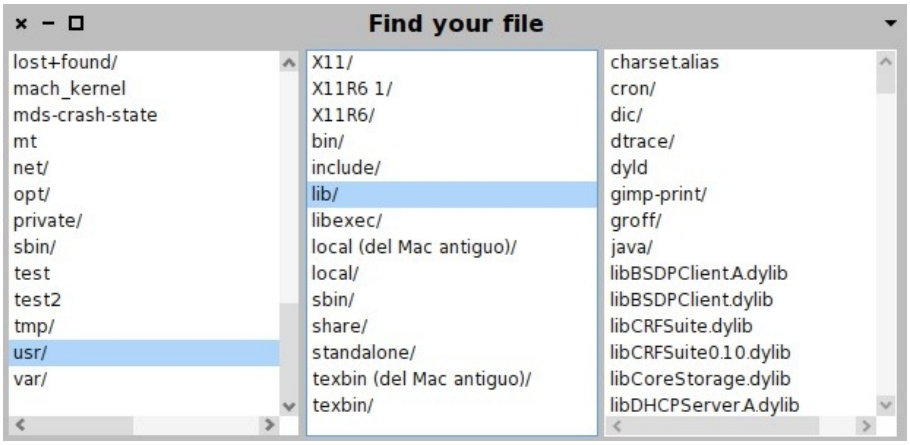


그림 10.1: 모든 스크립트는 http://get.pharo.org 에서 이용 가능하다.

우리 예제처럼 파일 시스템을 탐색하는 경우, 브라우저는 특정 디렉터리의 엔트리(각 파일과 디렉터리)의 리스트를 첫 열에 표시하고, 사용자 선택영역에 따라 다른 열이 추가된다 (그림 10.1 참고).

- 사용자가 디렉터리를 선택하면 다음 열(column)은 그 특정 디렉터리의 엔트리를 표시할 것이다.
- 사용자가 파일을 선택하면 다음 열(column)은 파일의 내용을 표시할 것이다.

재귀성(recursion) 때문에 처음엔 복잡하게 보일 수도 있다. 하지만 Glamour는 Miller Columns 기반의 브라우저를 직관적으로 설명하는 방식을 제공한다. Glamour의 용어에 따르면 이 특별한 브라우저를 파인더(finder)라고 부르는데, 이는 Mac OS X에서 찾을 수 있는 Apple사의 Finder를 지칭한다. Glamour는 GLMFinder 클래스를 이용해 이러한 행위를 제공한다. 해당 클래스는 우리의 관심 영역, 즉 파일을 적절하게 열거하도록 인스턴스화되고 초기화되어야 한다.

```
| browser |
browser := GLMFinder new.
browser show: [:a |
    a list
        display: #children ].
browser openOn: FileSystem disk root.
```

이 단계에서 평문(plain) 파일을 선택하면 오류를 야기한다. 이러한 상황이 발생하는 이유와 수정 방법은 머지않아 이해하게 될 것이다.

이렇게 작은 코드 조각을 이용해 자신의 파일 시스템의 루트에서 발견되는 모든 엔트리의 (파일 또는 디렉터리) 리스트를 얻게 되고, 각 행은 파일이나 디렉터리를 나타낼 것이다. 디렉터리를 하나 클릭하면 해당 디렉터리의 엔트리가 다음 열에 표시될 것이다. 파일시스템 탐색 기능은 Filesystem 프레임워크에 의해 제공되는데, 이와 관련된 내용은 제 3장에서 충분하게 논한 바 있다.

하지만 이 코드에도 문제가 있다. 각 행은 엔트리의 전체 출력 문자열(print string)을 표시하는데 그 방식이 당신이 원하는 방식과 다를 수도 있다는 사실이다. 일반 사용자는 각 엔트리의 이름만 표시될 것이라고 예상한다. 이는 리스트를 맞춤설정하여 쉽게 해결이 가능하다.

```
browser show: [:a |
    a list
        display: #children;
        format: #basename ].
```

이런 방식으로 basename 메시지가 각 엔트리로 전송되어 이름을 얻을 것이다. 이는 파일과 디렉터리의 풀네임 대신 파일명만 표시하여 훨씬 읽기가 쉽다.

또 다른 문제로는, 코드가 파일과 디렉터리를 구별하지 않는다는 점을 들 수 있다. 파일을 클릭하면 브라우저는 파일이 이해하지 못하는 children이란 메시지를 파일로 전송하기 때문에 오류를 수신할 것이다. 이 문제를 수정하려면 선택된 요소가 파일일 경우 포함된 엔트리 리스트를 표시하지 않도록 해야 한다.

```
browser show: [:a |
    a list
        when: #isDirectory;
        display: #children;
        format: #basename ].
```

이러한 방식은 잘 작동하긴 하지만 사용자가 파일을 표현하는 행과 디렉터리를 표현하는 행을 구별하지 못한다는 단점이 있다. 여기서는 선택된 요소가 디렉터리일 경우 파일명 끝에 슬래시를 추가하는 등의 방법을 이용해 해결할 수 있다.

```
browser show: [:a |
    a list
        when: #isDirectory;
        display: #children;
        format: #basenameWithIndicator ].
```

파일일 경우 엔트리 내용을 표시하는 일은 절대로 하지 말아야 한다. 아래는 파일 브라우저의 최종 버전을 제공한다.

```
| browser |
browser := GLMFinder new
    variableSizePanes;
    title: 'Find your file';
    yourself.

browser show: [:a |
    a list
        when: #isDirectory;
        display: [:each | [each children ]
            on: Exception
            do: [Array new]];
        format: #basenameWithIndicator.
    a text
        when: #isFile;
        display: [:entry | [entry readStream contents]
            on: Exception
                do:['Can''t display the content of this file'] ] ].

browser openOn: FileSystem disk root.
```

위의 코드는 기존의 브라우저를 다양한 크기의 패인과 제목을 비롯해 디렉터리 엔트리, 접근 권한 처리, 파일 내용 읽기로 확장한다. 그 결과로 만들어진 브라우저는 그림 10.1에 소개하고 있다.

간략한 서론을 통해 Glamour를 설치하는 방법과 이를 이용해 간단한 파일 브라우저를 생성하는 방법을 소개해보았다.

Presentation, Transmission 그리고 Ports

이번 절은 Glamour 프레임워크의 실질적인 예제와 세부 내용을 제공한다.

실행 예제

다음 지침을 통해 간단한 스몰토크 클래스 탐색기(navigator)를 생성할 것이다. 그러한 탐색기는 많은 스몰토크 브라우저에서 사용되고 주로 4개의 패인으로 구성되는데 그림 10.2에 추상적으로 표시하였다.

클래스 탐색기는 다음과 같은 기능을 한다. 패인 1은 패키지의 트리 또는 리스트를 표시하는데, 각 패키지는 환경의 조직적 구조를 구성하는 클래스들을 포함한다. 패키지를 선택하면 선택된 패키지 내 모든 클래스의 리스트가 패인 2에 표시된다. 클래스를 하나 선택하면 패인 3에 모든 프로토콜(메서드를 그룹화하기 위한 구조체, 메서드 범주라고도 알려짐)을 표시하고, 패인 4에는 클래스의 모든 메서드가 표시된다. 패인 3에서 프로토콜을 하나 선택하면 해당 프로토콜에 속하는 메서드의 하위집합이 패인 4에 표시된다.

브라우저 시작하기

브라우저를 반복적으로 빌드하고 Glamour의 새 구조체를 서서히 소개해보겠다. 우선 패키지의 리스트에 새 브라우저를 열길 원한다. 예제에는 이전 파일 브라우저 이상의 코드를 수반할 것이기 때문에 전용(dedicated) 클래스에 코드 브라우저를 구현하고자 한다.

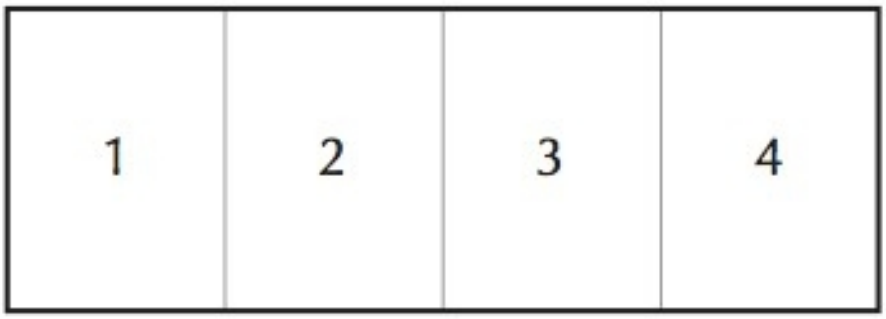


그림 10.2: Pane list.

첫 번째 단계는 몇 가지 initial 메서드가 있는 클래스를 생성하는 것이다.

```
Object subclass: #PBE2CodeNavigator
    instanceVariableNames: 'browser'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PBE2-CodeBrowser'

PBE2CodeNavigator class>>open
    ^ self new open

PBE2CodeNavigator>>open
    self buildBrowser.
    browser openOn: self organizer.

PBE2CodeNavigator>>organizer
    ^ RPackageOrganizer default

PBE2CodeNavigator>>buildBrowser
    browser := GLMTabulator new.
```

PBE2CodeNavigator open 을 실행하면 "a RPackageOrganizer" 텍스트로 된 새 브라우저를 열기만 할 뿐 다른 일은 전혀 하지 않는다. 이제 브라우저를 생성하기 위해 GLMTabulator 클래스를 사용함을 주목한다. GLMTabulator는 명시적 브라우저로서, 패인을 열과 행에 위치시키도록 해준다.

패키지의 리스트를 표시하기 위해 새 패인으로 브라우저를 확장한다.

```
PBE2CodeNavigator>>buildBrowser
    browser := GLMTabulator new.
    Browser
        column: #packages.

    browser transmit to: #packages; andShow: [:a | self packagesIn: a].

PBE2CodeNavigator>>packagesIn: constructor
    constructor list
        display: [:organizer | organizer packageNames sorted];
        format: #asString
```

Glamour 브라우저는 패인, 그리고 패인들 간 데이터 흐름과 관련해 구성된다. 예제로 제시

한 브라우저는 패키지를 표시하는 하나의 패인만 포함한다. 데이터의 흐름은 transmissions를 이용해 명시된다. 이는 리스트 내 항목 선택과 같이, 브라우저의 그래픽 사용자 인터페이스에서 특정 내용이 변경되면 트리거된다. 선택된 패키지에 포함되어 있는 클래스를 표시함으로써 브라우저를 더 흥미롭게 만들고자 한다 (그림 10.3 참고).

```
PBE2CodeNavigator>>buildBrowser
    browser := GLMTabulator new.
    browser
        column: #packages;
        column: #classes.

    browser transmit to: #packages; andShow: [:a | self packagesIn: a].
    browser transmit from: #packages; to: #classes; andShow: [:a | self classesIn: a].

PBE2CodeNavigator>>classesIn: constructor
    constructor list
        display: [:packageName | (self organizer packageNamed: packageName)
            definedClasses]
```

위의 리스팅은 Glamour의 거의 모든 핵심 언어 구조체를 표시한다. 후에 패인의 참조가 가능하길 원하므로 "packages"와 "classes"라는 이름을 따로 부여하고, column: 키워드를 이용해 열(column)에 정렬한다. 이와 유사한 row: 이라는 키워드도 존재하는데, 이는 패인을 행(row)으로 조직할 수 있다.

transmit:, to:, from: 키워드는 transmission, 즉 한 패인에서 다른 패인으로의 정보 흐름을 정의하는 직접 연결을 생성한다. 예제의 경우, packages 패인으로부터 classes 패인으로 연계(link)를 생성한다. from: 키워드는 전송의 origin(출발지)를 나타내고, to: 는 destination(목적지)를 나타낸다. 더 이상 구체적인 내용이 표기되지 않는 이상 Glamour는 출발지를 명시된 패인의 selection(선택 영역)을 가리키는 것으로 가정한다. Origin 패인의 다른 측면들을 명시하는 방법과 여러 개의 origin을 사용하는 방법은 아래에서 설명하겠다.

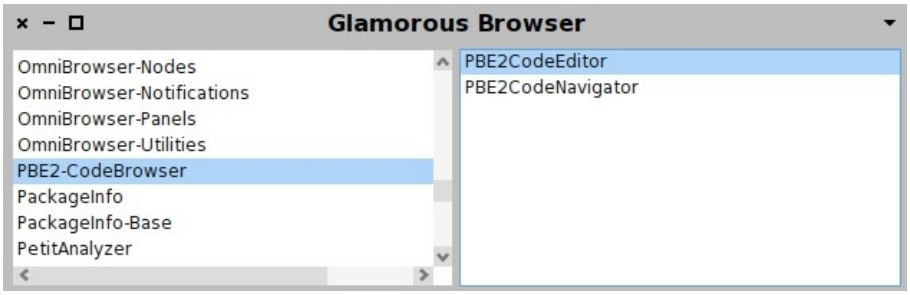


그림 10.3: 2-pain 브라우저. 왼쪽 패인에서 패키지를 선택하면 패키지에 포함된 클래스가 오른쪽 패인에 표시된다.

마지막으로, andShow: 는 연결이 활성화되거나 transmitted(전송)되었을 때 destination 패인에 무엇을 표시할 것인지를 명시한다. 예제에서는 왼쪽에서 선택한 패키지에 포함된 클래스의 리스트를 표시하고자 한다.

display: 키워드는 제공된 블록을 단순히 presentation 내부에 저장한다. 블록은 후에 presentation을 on-screen에 표시해야 하는 경우에만 평가될 것이다. 명시적 디스플레이 블록이

명시되지 않은 경우 Glamour는 일반적인 방식으로 객체를 표시하고자 시도할 것이다. 리스트를 표시하는 예제에서 이는 displayString 메시지가 객체로 전송되어 표준 문자열 표현을 검색할 것임을 뜻한다. 이미 살펴보았듯 이러한 행위는 format: 를 이용해 변경할 수 있다.

display: 를 비롯해 when: 조건을 명시하여 연결의 적용 가능성을 제한하는 것도 가능하다. 기본적으로는 항목이 실제로 선택되어야 한다는 것이 유일한 조건으로, 디스플레이 변수 인자가 null이 아닌 값이어야 한다.

또 다른 Presentation

지금까지 패키지는 시각적으로 평평한 리스트로 표현되었다. 하지만 패키지는 본래 해당하는 클래스 범주로 구조화된다. 이러한 구조를 활용하기 위해 리스트를 패키지의 트리 표현으로 대체하겠다.

```
PBE2CodeNavigator>>packagesIn: constructor
    constructor tree
        display: [ :organizer | (self rootPackagesOn: organizer) asSet sorted ];
        children: [ :rootPackage :organizer | (self childrenOf: rootPackage on: organizer)
            sorted ];
        format: #asString

PBE2CodeNavigator>>classesIn: constructor
    constructor list
    when: [:packageName | self organizer includesPackageNamed: packageName ];
    display: [:packageName | (self organizer packageNamed: packageName)
        definedClasses]

PBE2CodeNavigator>>childrenOf: rootPackage on: organizer
    ^ organizer packageNames select: [ :name | name beginsWith: rootPackage , '-' ]

PBE2CodeNavigator>>rootPackagesOn: organizer
    ^ organizer packageNames collect: [ :string | string readStream upTo: $- ]
```

트리 표현은 트리 내에 주어진 항목의 자식들을 검색하는 방법을 명시하기 위해 하나의 선택자나 블록을 취하는 children: 인자를 이용한다. 각 패키지의 자식들은 이제 트리 표현으로 선택되기 때문에 패키지 계층구조의 루트만 display: 인자로 전달해야 한다.

이 시점에서 Pane 3를 추가하여 메서드 범주를 열거하는 것도 가능하다 (그림 10.4). 아래 리스팅은 지금까지 논한 요소로 구성된다.

```
PBE2CodeNavigator>>buildBrowser
    browser := GLMTabulator new.
    browser
        column: #packages;
        column: #classes;
        column: #categories.
    browser transmit to: #packages; andShow: [:a | self packagesIn: a].
    browser transmit from: #packages; to: #classes; andShow: [:a | self classesIn: a].
    browser transmit from: #classes; to: #categories; andShow: [:a | self categoriesIn: a].

PBE2CodeNavigator>>categoriesIn: constructor
    constructor list
        display: [:class | class organization categories]
```

위의 변경 내용에서 도출된 브라우저는 그림 10.4에 실려 있다.

다수의 Origin

메서드의 리스트를 Pane 4로서 추가하는 데에는 조금 더 많은 조직이 수반된다. 메서드 범주를 선택하면 해당 범주에 속하는 메서드만 표시하길 원할 것이다. 어떤 범주도 선택되지 않을 경우 현재 클래스에 속한 모든 메서드가 표시된다.

이는 메서드 패인으로 하여금 클래스 패인과 범주 패인의 선택 영역에 의존하도록 만든다. 다중 origin은 아래와 같이 from: 키워드를 여러 번 이용해 정의할 수 있다.

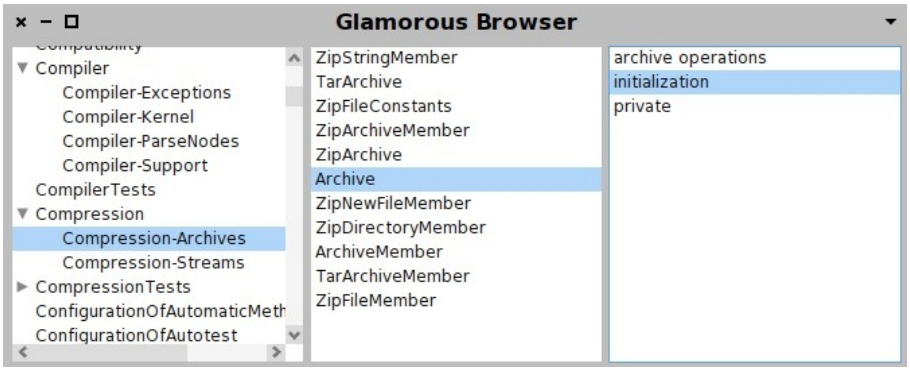


그림 10.4: 선택된 클래스의 메서드 범주 리스트와 패키지를 표시하는 트리가 포함된 향상된 클래스 탐색기.

```
PBE2CodeNavigator>>buildBrowser
    browser := GLMTabulator new.
    browser
        column: #packages;
        column: #classes;
        column: #categories;
        column: #methods.

    browser transmit to: #packages; andShow: [:a | self packagesIn: a].
    browser transmit from: #packages; to: #classes; andShow: [:a | self classesIn: a].
    browser transmit from: #classes; to: #categories; andShow: [:a | self categoriesIn: a].
    browser transmit from: #classes; from: #categories; to: #methods;
        andShow: [:a | self methodsIn: a].

PBE2CodeNavigator>>methodsIn: constructor
    constructor list
        display: [:class :category |
            (class organization listAtCategoryNamed: category) sorted].
    constructor list
        when: [:class :category | class notNil and: [category isNil]];
        display: [:class | class selectors sorted];
        allowNil
```

리스팅에 몇 가지 새로운 프로퍼티가 보인다. 첫째, 다중 origin은 display: 와 when: 절에 사용된 블록의 인자 개수에 반영된다. 둘째, 하나 이상의 표현을 사용하고 있어서, 해당 전송이 fired되면 Glamour는 정의된 순서에 매칭하는 조건의 표현을 모두 표시한다.

첫 번째 표현에서는 모든 인자가 정의되어 (null이 아닌 값) 있을 때 조건이 매칭하는데, 모든 표현에서 기본 값이다. 두 번째 조건은 범주가 정의되어 있지 않고 클래스가 정의되어 있을

때에만 매칭한다. origin이 정의되어 있지 않을 때에도 표현을 표시해야 하는 경우, 표시된 바와 같이 allowNil 을 사용하는 것이 유용하겠다. 따라서 디스플레이 블록에서 범주를 생략할 수 있다.

완성된 클래스 탐색기는 그림 10.5와 같은 모습이다.

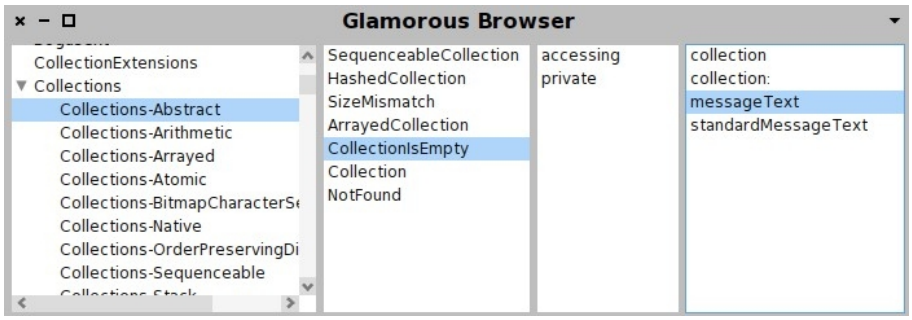


그림 10.5: 완전한 코드 탐색기. 메서드 범주가 선택되지 않은 경우 클래스의 모든 메서드가 표시된다. 선택된 경우 해당 범주에 속하는 메서드만 표시된다.

Ports

transmissions는 패인으로 연결된다고 언급한 적이 있는데, 전적으로 옳은 말은 아니다. 좀 더 정밀하게 말해, transmissions는 ports(포트)라고 불리는 패인의 프로퍼티로 연결된다. 그러한 포트는 패인의 상태에 대한 특정 측면이나 그에 포함된 표현을 나타내는 값과 이름으로 구성된다. 사용자가 명시적으로 포트를 명시하지 않으면 Glamour는 기본적으로 selection 포트를 이용한다. 그 결과 아래와 같은 두 개의 문은 동일하다고 볼 수 있다.

```
browser transmit from: \#packages; to: \#classes; andShow: [:a | {\dots}].
browser transmit from: \#packages port: \#selection; to: \#classes; andShow: [:a | {\dots}].
```

구성하기(composing)와 상호작용

Browsers 재사용하기

Glamour의 강점 중 하나로 리스트나 트리와 같은 primitive 표현 대신 브라우저를 사용한다는 점을 들 수 있다. 이는 브라우저를 구성하고 높은 내포(nest)의 가능성을 제시한다.

다음 예제는 그림 10.6과 같은 editor 클래스를 정의한다. 패인 1부터 4까지는 앞에서 설명한 패인과 같다. 패인 5는 패인 4에서 현재 선택된 메서드의 소스 코드를 표시한다.

PBE2CodeEditor라는 새 클래스가 해당 editor를 구현할 것이다. editor는 패인 1부터 4까지 표현을 이전에 구현된 PED2CodeNavigator로 위임할 것이다. 이를 위해서는 먼저 기존의 탐색기가 구성된 브라우저를 반환하도록 만들어야 한다.

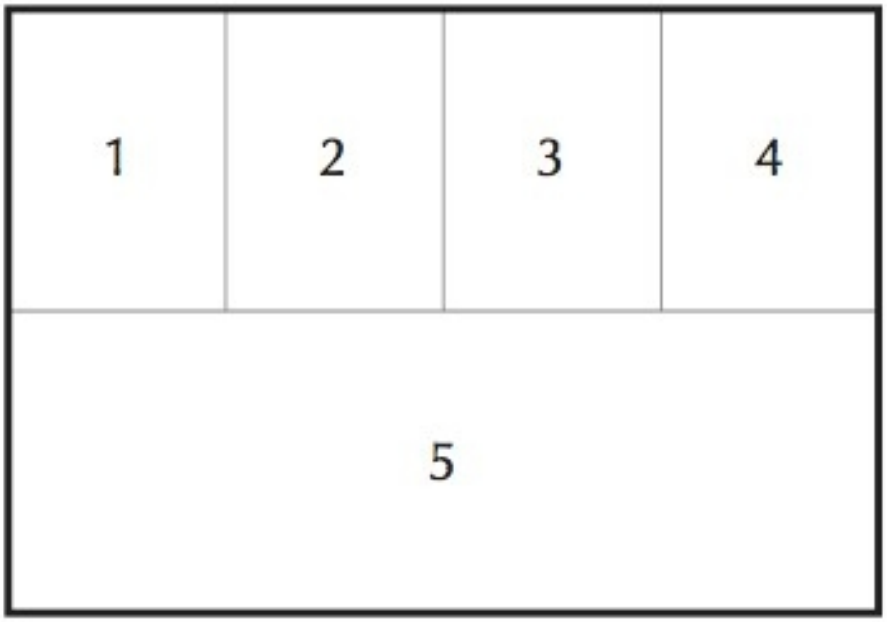


그림 10.6: 5 pane.

```
PBE2CodeNavigator>>buildBrowser
    ...
    ”new line”
    ^ browser
```

다음으로, 아래와 같이 새 editor 브라우저에서 탐색기를 재사용할 수 있다.

```
Object subclass: #PBE2CodeEditor
    instanceVariableNames: 'browser'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PBE2-CodeBrowser'.

PBE2CodeEditor class>>open
    ^ self new open

PBE2CodeEditor>>open
    self buildBrowser.
    browser openOn: self organizer

PBE2CodeEditor>>organizer
    ^ RPackageOrganizer default

PBE2CodeEditor>>buildBrowser
    browser := GLMTabulator new.
    browser
        row: #navigator;
        row: #source.

    browser transmit to: #navigator; andShow: [:a | self navigatorIn: a ].

PBE2CodeEditor>>navigatorIn: constructor
    constructor custom: (PBE2CodeNavigator new buildBrowser)
```

위의 리스팅은 우리가 리스트나 다른 타입의 표현을 사용할 때와 정확히 같은 방식으로 사용함을 보여준다. 사실 브라우저는 표현의 한 가지 타입이다.

PBE2CodeEditor open을 평가하면 탐색기를 윗부분에 삽입하고 아랫부분엔 빈 패인을 가진 브라우저가 열린다. 소스 코드는 아직 표시되지 않는데, 패인 간 어떤 연결도 구축되지 않았기 때문이다. 소스 코드는 텍스트 패인으로 탐색기를 연결함으로써 얻으며, 선택된 메서드명과 그것이 정의된 클래스도 필요하다. 이러한 정보는 navigator 브라우저에서만 정의되기 때문에 먼저 sendToOutside:from: 을 이용해 외부 세계로 내보내야 한다. 이를 위해 아래의 행을 codeNavigator로 추가한다.

```
PBE2CodeNavigator>>buildBrowser
    ...
    browser transmit from: #classes; toOutsidePort: #selectedClass.
    browser transmit from: #methods; toOutsidePort: #selectedMethod.

    ^ browser
```

그러면 포함하는 패인의 selectedClass 포트와 selectedMethods 포트로 클래스와 메서드 내에서 선택한 영역이 전송될 것이다. 이 방법이 아니라면, 코드 에디터에서 navigationIn: 메서드로 아래와 같은 행을 추가할 수도 있는데, Glamour에는 차이가 없다.

```
PBE2CodeEditor>>navigatorIn: constructor
    ”Alternative way of adding outside ports. There is no need to use this
    code and the previous one simultaneously.”

    | navigator |
    navigator := PBE2CodeNavigator new buildBrowser
        sendToOutside: #selectedClass from: #classes -> #selection;
        sendToOutside: #selectedMethod from: #methods -> #selection;
        yourself.

    constructor custom: navigator
```

하지만 해당 인터페이스의 재사용까지 장려하기 위해서는 코드 편집기 내부보다는 코드 navigator 측에서 인터페이스를 분명히 정의하는 편이 더 합리적이라고 생각한다.

따라서 코드 에디터 예제를 아래와 같이 확장한다.

```
PBE2CodeEditor>>buildBrowser
    browser := GLMTabulator new.
    browser
        row: #navigator;
        row: #source.

    browser transmit to: #navigator; andShow: [:a | self navigatorIn: a].
    browser transmit
        from: #navigator port: #selectedClass;
        from: #navigator port: #selectedMethod;
        to: #source;
        andShow: [:a | self sourceIn: a].

PBE2CodeEditor>>sourceIn: constructor
    constructor text
        display: [:class :method | class sourceCodeAt: method]
```

이제 선택된 메서드의 소스 코드는 모두 확인이 가능하며, 앞에서 이미 작성한 클래스 탐색기를 이용하여 modular 브라우저를 생성하였다. 리스팅이 설명한 대로 구성된 브라우저는 그림 10.7에서 소개한다.

액션

영역(domain)의 탐색은 흥미로운 요소를 찾는 데 필수적이다. 하지만 이용 가능한 액션의 집합을 갖는 것은 영역과 상호작용을 허용하는 데에 꼭 필요하다. 액션은 정의되어 표현과 연관될 수 있다. 액션은 키보드 단축키를 누르거나 컨텍스트 메뉴에서 엔트리를 클릭할 때 평가되는 블록이다. 액션은 표현으로 전송되는 act:on: 을 통해 정의된다.

```
PBE2CodeEditor>>sourceIn: constructor
    constructor text
        display: [:class :method | class sourceCodeAt: method ];
        act: [:presentation :class :method | class compile: presentation text] on: $s.
```

on: 으로 전달된 인자는 해당하는 표현이 포커스를 받았을 때 액션을 트리거하는 데 사용되어야 하는 키보드 단축키를 명시하는 문자다. 문자를 메타 키, 가령 콤마, 컨트롤, 알트와 같은 키와 결합해야 하는지 여부는 플랫폼에 따라 다르므로 명시되지 않아도 된다. act: 블록은 상응하는 표현을 첫 번째 인자로서 제공하는데, 결합된 텍스트나 현재 선택 영역과 같은 다양한

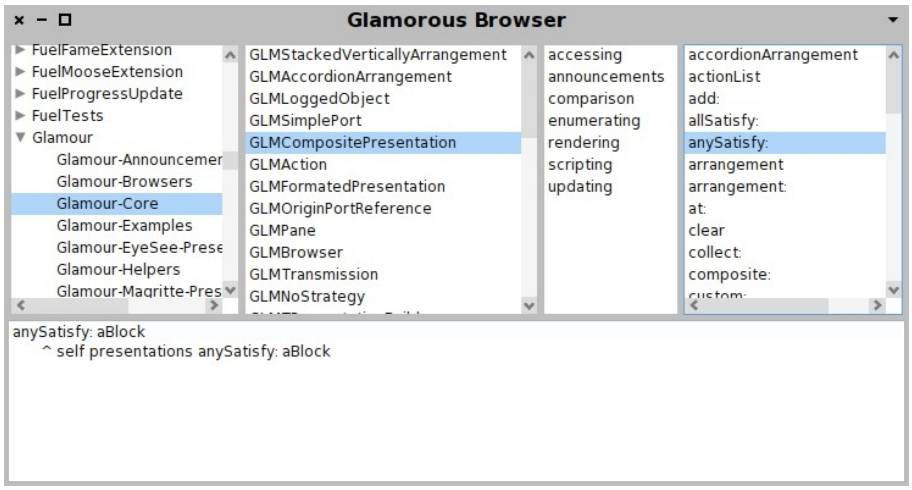


그림 10.7: 선택된 메서드의 소스를 표시하기 위해 앞에서 설명한 클래스를 재사용하는 구성(composed) 브라우저.

프로퍼티를 알아내는 데 사용할 수 있다. 다른 블록 인자들은 from: 이 정의하는 대로 들어오는 origin이며, display: 와 when: 의 인자와 같다.

액션은 컨텍스트 메뉴로 표시되기도 한다. 이를 위해 Glamour는 act:on:entited: 와 act:entitled: 메시지를 제공하는데, 이러한 메시지에서 마지막 인자는 메뉴의 엔트리로 표시되어야 하는 문자열에 해당한다. 예를 들어, 아래 코드 조각은 위의 예제를 확장시켜 현재 메서드를 클래스로 "저장"하기 위한 컨텍스트 메뉴 엔트리를 제공한다.

```
...
    act: [:presentation :class :method | class compile: presentation text]
    on: $s
    entitled: 'Save'
```

컨텍스트 메뉴는 왼쪽에 위치한 텍스트 패인 위의 아래 방향 삼각형(triangle downward-oriented)을 통해 접근 가능하다.

다수의 Presentation

개발자들은 특정 객체의 표현을 하나 이상 제공하고 싶을 때가 있다. 앞에서 제시한 코드 브라우저의 예제에서 가령 클래스를 리스트 뿐만 아니라 그래픽 표현으로서 표시하길 원한다고 가정하자. Glamour는 Mondrian visualization engine (12장에서 소개)을 이용해 생성된 시각화의 표시 및 상호작용을 지원한다. 2차 표현을 추가하기 위해서는 아래와 같이 using: 블록에서 정의하면 된다.

```
PBE2CodeNavigator>>classesIn: constructor
    constructor list
        when: [:packageName | self organizer includesPackageNamed: packageName ];
        display: [:packageName | (self organizer packageNamed: packageName)
            definedClasses];
        title: 'Class list'.

constructor mondrian
    when: [:packageName | self organizer includesPackageNamed: packageName];
    painting: [ :view :packageName |
        view nodes: (self organizer packageNamed: packageName)
            definedClasses.
        view edgesFrom: #superclass.
        view treeLayout];
    title: 'Hierarchy'
```

Glamour는 탭 레이아웃의 도움을 받아 같은 패인에 다수의 표현을 구별한다. Mondian 표현의 모양은 코드 에디터에 내포되어 있는데, 이는 그림 10.8에서 소개한다. title: 절은 표현을 렌더링하는 데에 사용되는 탭의 이름을 설정한다.

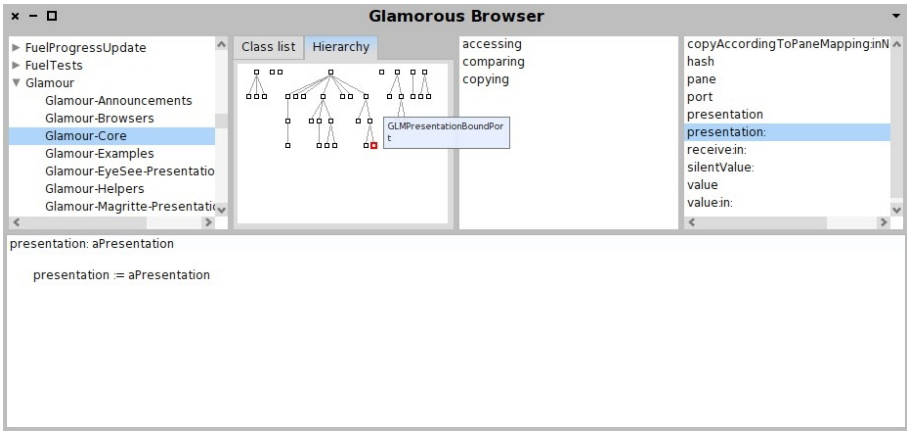


그림 10.8: 간단한 클래스 리스트와 함께 Mondrian 표현을 표시하는 코드 에디터.

기타 브라우저

앞서 언급한 row: 와 column: 키워드를 이용해 커스텀 레이아웃을 생성하는 기능을 본따 명명된 GLMTabulator를 기본적으로 이용해왔다. 사용자는 추가 브라우저를 제공하거나 작성할 수 있다. 브라우저 구현은 두 가지 범주로 다시 나뉘는데, 명시적 패인을 가진 브라우저가 첫 번째 범주로서 사용자가 명시적으로 선언하는 것이며, 두 번째는 암시적 패인(implicit pane)이 되겠다.

GLMTabulator는 명시적 패인을 사용하는 브라우저의 예에 해당한다. 암시적 브라우저를 이용할 경우 패인을 직접적으로 선언하지 않고 브라우저가 패인과 패인 간 연결을 내부적으로 생성한다. 그러한 브라우저의 예로 Finder를 들 수 있는데, 이와 관련된 내용은 10.1절에서 논

한 바 있다. 패인이 알아서 생성되기 때문에 우리는 from:to: 키워드를 사용하지 않아도 되며, 단순히 표현을 명시하기만 하면 된다.

```
browser := GLMFinder new.

browser list
    display: [:class | class subclasses].

browser openOn: Collection
```

위의 리스팅은 브라우저를 생성하고 (그림 10.9에 표시) 열어서 Collection의 서브클래스 리스트를 표시한다. 리스트에서 항목을 선택하면 브라우저가 오른쪽으로 확장되면서 선택된 항목의 서브클래스를 표시한다. 선택할 내용이 남아 있는 한 무한으로 계속될 수 있다.

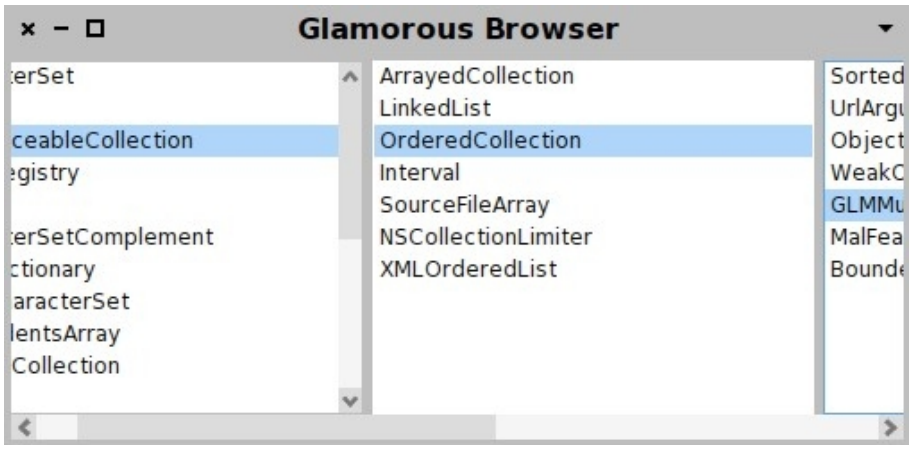


그림 10.9: Miller Columns 스타일 브라우징을 이용한 서브클래스 탐색기.

다른 유형의 브라우저를 살펴보기 위해서는 GLMBrowser 클래스의 계층구조를 탐색해보라.

요약

본 장을 통해 Glamour 브라우저 프레임워크를 간략하게 소개하였다. Glamour는 기본적으로 평범한 객체로 구성된 임의의 영역을 탐색하고 상호 작용할 수 있게 해주는 빌드 툴을 빌드하는 데에 사용된다.

- GLMTabulator는 위젯을 열과 행으로 정렬한 일반 브라우저다.
- 열은 기호 이름을 인자로 한 column: 을 연속으로 전송함으로써 정의된다.
- 데이터는 transmit from: #source; to: #target 을 이용해 설정된 transmissions를 따라 흐른다.
- transmission는 여러 개의 소스를 가질 수 있다.

- 리스트 패인과 텍스트 패인은 브라우저로 list와 test를 각각 전송하여 얻는다. 내용은 display: 를 이용해 설정되고, 항목들은 format: 를 이용해 포맷팅된다.
- Ports는 브라우저의 컴포넌트 인터페이스를 정의한다. 이는 재사용과 삽입(embedding)을 수월하게 해준다.
- 상호작용은 위젯으로 act: 를 전송하여 정의되는 액션과 관련해 정의된다.
- Glamour는 다수의 표현을 지원한다.
- Glamour는 일반 용도의 그래픽 사용자 인터페이스를 빌드하기 위해 만들어진 것이 아니다.

이번 장은 Glamour의 전반적인 개요를 제공하는 것이 목적이 아니라 독자에게 필자의 접근법의 사용법과 그 의도를 소개하기 위해 작성되었다. Glamour 와 그 개념 및 구현을 광범위하게 살펴보기 위해서는 Moose 에서 이를 집중적으로 다룬 장을 참고한다[1].

[1] http://www.themoosebook.org/book

제 11 장

Roassal을 이용한 재빠른 시각화

Vanessa Peña-Araya *참여 (van.c.pena@gmail.com)*

많은 양의 데이터에 의미를 부여하기란 적절한 툴이 없이는 힘들다. 텍스트 출력(textual output)은 그 표현의 유연성(expressiveness)과 상호작용의 지원에 제한이 있는 것으로 알려져 있다.

Roassal은 재빠른 시각화 엔진이다. Roassal은 객체나 그 관계와 관련해 정의된 임의의 데이터를 시각화하고 그것과 상호작용하기 위해 만들어졌다. Roassal은 보통 상호작용적 시각화를 생성하는 데 사용된다. Roassal을 이용한 애플리케이션의 범위는 다양하다. 예를 들어, Moose 공동체는 소프트웨어를 시각화하는 데 Roassal을 사용한다.

이번 장에서는 Roassal의 원리를 사용하고, 자신의 데이터를 빠르게 렌더링하기 위해 Rossal의 표현적인 API의 사용을 설명하겠다. 본 장을 마칠 즈음엔 상호작용적이고 시각적인 표현을 생성할 수 있을 것이다.

Roassal의 개발은 ESUG.org의 후원으로 이루어지고 있다. 자세한 정보는 Roassal 웹사이트를 방문하면 찾을 수 있다.

http://objectprofile.com/#/pages/products/roassal/overview.html

설치 및 첫 시각화

Roassal은 Moose 배포판에[1] 속한다. 따라서 어떠한 설치도 불필요하며, 첫 번째 시각화를 곧바로 진행할 수 있다.

Gofer와 Metacello 덕분에 Roassal를 새로운 Pharo 이미지에 설치하기란 쉽다. 워크스페이스를 열고 아래를 실행하면 된다.

[1] http://www.moosetechnology.org/

```
Gofer new smalltalkhubUser: 'ObjectProfile'
    project: 'Roassal';
    package: 'ConfigurationOfRoassal';
    load.
(Smalltalk at: #ConfigurationOfRoassal) load
```

Rossal은 Pharo 버전 1.4, 2.0, 3.0에서 실행되는 것으로 알려진다.

첫 시각화.

우리가 보일 첫 시각화는 Collection 클래스 계층구조를 나타낸다. 이는 각 클래스를 상자로 정의하고, 각 상자는 그 서브클래스로 연결된다. 각 상자는 표현하는 클래스의 인스턴스 변수 개수와 메서드 개수를 표시한다.

```
view := ROView new.
classElements := ROElement forCollection:
    Collection withAllSubclasses.

classElements
    do: [:c |
        c width: c model instVarNames size.
        c height: c model methods size.
        c + ROBorder.
        c @ RODraggable ].
view addAll: classElements.

associations := classElements collect: [:c |
    (c model superclass = Object)
        ifFalse: [ (view elementFromModel: c
        model superclass) -> c]
    ] thenSelect: [:assoc | assoc isNil not ].
edges := ROEdge linesFor: associations.
view addAll: edges.

ROTreeLayout new on: view elements.
view open
```

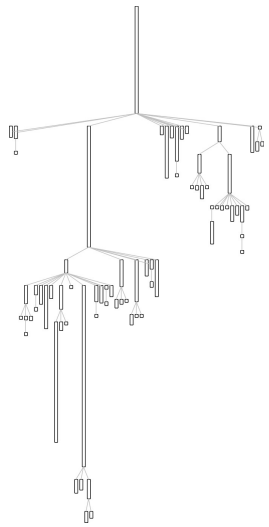


그림 11.1: Roassal 프레임워크를 이용한 첫 시각화.

이러한 시각화를 만드는 방법을 이번 장에서 설명할 것이다. 다음 장에서는 Mondrian의 도메인 특정 언어(DSL)를 이용해 시각화를 생성하는 방법을 자세히 살펴볼 것인데, 그 과정에서 Roassal의 일부인 Mondrian 빌더를 이용할 것이다.

Roassal Easel

Roassal easel은 상호작용적으로 시각화를 스크립팅하기 위한 툴이다. easel을 이용하면 프로그래머가 easel에서 작업을 수행하는 painter가 되어 생성, 조정, 삭제는 몇 개의 (키) 누름으로 가능해진다.

Roassal easel은 Pharo World 메뉴에서 접근할 수 있다. $\mathcal{R}$ 아이콘[2]을 살펴보라.

Easel은 두 개의 독립된 창으로 구성되는데, 하나는 왼쪽 측면에 위치하여 오른쪽에 있는 텍스트 창에 적힌 스크립트를 렌더링한다. 에디터에서 수락을 (Cmd-s, Alt-s/오른쪽 마우스 클릭 후 accept를 누름) 통해 시각화가 업데이트될 것이다. 이는 시스템 브라우저에서 메서드를 수락 시 사용되는 키 누름과 같다. 이를 사용 시 장점은 피드백 루프가 짧다는 것으로, 스크립트의 의미는 항상 하나의 키 누름으로 가능하다.

시각화 창은 다수의 시각화 예제를 포함하는데, 단계별(step-by-step) 지침서도 이에 포함된다. 예제들은 두 가지 범주, ROExample과 ROMondrianExample로 나뉘며, 시각화 창의 윗부분에 examples 버튼을 클릭하여 접근할 수 있다.

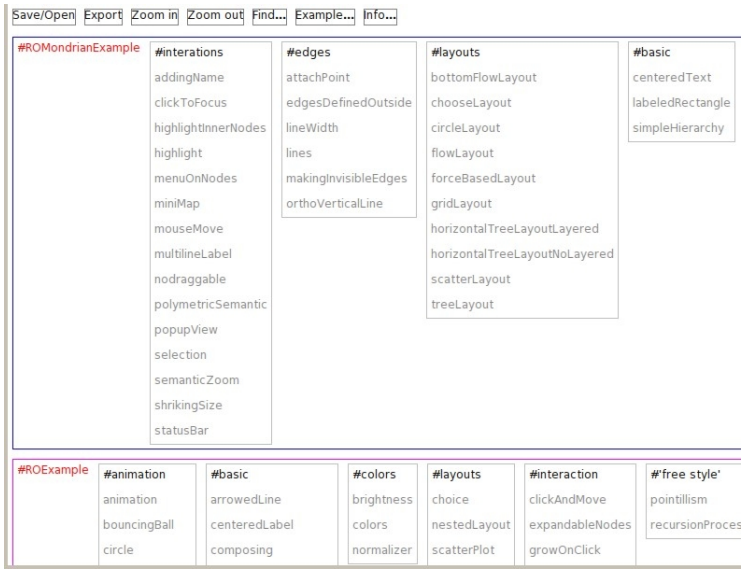


그림 11.2: ROMondrianViewBuilder와 ROExample 범주로 나뉜 Roassal Easel 예제.

ROMondrianExample 범주는 Roassal의 상단에 빌드된 도메인 특정 언어인 Modrian으로 생성된 예제들을 포함한다. 이 예제들은 주로 시각화를 만드는 데 ROMondrianViewBuilder 클래스를 사용한다. ROExample 범주는 Roassal을 직접 설명한다.

Roassal 코어 모델

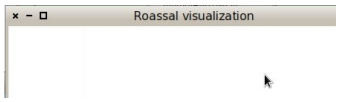
각 시각화의 루트는 렌더링되어야 할 모든 그래픽 구성 요소에 대한 컨테이너 역할을 하는 ROView 클래스의 인스턴스다. 그러한 구성 요소들은 ROAbstractComponent의 서브클래스의 인스턴스로서, 주로 ROElement와 ROEdge의 인스턴스들이다. 주로 그래픽 구성 요소는 domain object에 대한 참조를 유지한다. 몇몇 시각적 프로퍼티들은 (크기 또는 색상) domain

[2]Glamour를 기반으로 한 easel도 World 메뉴의 Moose section에서 제공됨을 주목한다. Glamour를 기반으로 한 easel은 본문에 제시된 easel과 유사하다. 해당 버전에 전용으로 만들어진 표현은 moose 서적, http://themoosebook.org 에서 찾을 수 있을 것이다.

object로부터 직접 추론할 수 있다. 이와 관련된 점은 곧 다시 살펴보겠다. 현재로선 기본적인 연산을 먼저 설명하겠다.

요소 추가하기. 그래픽 구성 요소를 확인하기 위한 첫 번째 단계는 구성 요소를 뷰로 추가한 후 뷰를 여는 것이다. 다음의 코드 조각은 정확히 이러한 일을 수행한다.

```
view := ROView new.
element := ROElement new size: 100.
view add: element.
view open.
```



위 코드는 100 픽셀 크기의 사각형으로 된 단일 요소가 있는 시각화를 생성하여 연다. 하지만 이 코드를 실행하면 시각화에 어떤 변화도 발생하지 않는다. 요소가 효과적으로 뷰에 추가되었지만 어떻게 렌더링되어야 하는지는 요소에게 알려주지 않은 것이다.

Shape 추가하기. 요소의 시각적 측면은 ROShape의 서브클래스의 인스턴스인 shapes에 의해 주어진다. 기본적으로 모든 요소는 어떤 모양(shape)[3]도 갖고 있지 않다. 모양(테두리)을 요소로 추가해보자.

```
view := ROView new.
element := ROElement new size: 100.
element addShape: ROBorder. "added line"
view add: element.
view open.
```

요소로 모양을 추가하는 일은 추가하고자 하는 모양이 있는 addShape: 메시지를 추가하기만 하면 된다는 사실은 놀라운 일도 아니다. 자주 사용되는 연산이기 때문에 + 메시지를 이용해도 같은 효과를 얻을 수 있다. 아니면 element + ROBorder 를 작성할 수도 있겠다.

본 예제에서는 ROBorder 모양을 추가하였다. 이름에서 제시하듯 ROBorder는 ROElement로 사각형 모양의 테두리를 추가한다. 기본 값으로 ROBorder는 검정색이다. 다른 모양들도 이용 가능한데, 맞춤형 라벨, 원, 또는 안이 채워진 사각형이 포함된다. 그러한 모양은 정교한 시각적 측면을 생성하도록 구성되기도 한다. 모양에 대한 개념은 11.3절에서 더 상세히 다루겠다.

이벤트에 반응하기. 현재 우리가 생성한 하나의 요소가 할 수 있는 일은 많지 않다. 해당 요소로 하여금 클릭, 드래그 앤 드롭, 키 누름 등의 사용자 액션을 인식하게 만들기 위해선 이벤트 콜백을 명시할 필요가 있다.

대부분의 사용자 인터페이스 및 그래픽 프레임워크와 마찬가지로 사용자가 실행할 각 액션마다 이벤트를 생성한다. 그러한 이벤트는 ROEvent의 서브클래스의 인스턴스에 해당한다.

[3] 사실 요소는 항상 shape, 즉 RONullShape의 인스턴스를 갖고 있다. 본문에서는 null 객체 디자인 패턴이 사용되었다.

그래픽 요소가 이벤트에 반응하도록 만들기 위해선 블록을 이벤트 클래스로 연관시켜야 하고 그래픽 요소로 부착시켜야 한다.

가령, 예제의 사각형을 사용자 클릭에 반응하도록 만들기 위해선 이벤트 핸들러를 추가해야 하는데, 이벤트가 발생하면 실행될 블록을 예로 들 수 있겠다.

```
view := ROView new.
element := ROElement new size: 100.
element + ROBorder.
"Open an inspector when clicking"
element on: ROMouseClick do: [:event | event inspect].
view add: element.
view open.
```

이제 사각형을 클릭하면 인스펙터가 열릴 것이다. 한편 우리는 addShape: 보다는 + 메시지를 선호하는 편인데, 짧으면서도 충분한 정보를 제공하기 때문이다.

복잡한 응답을 위한 상호작용. 사용자 액션에 직접 응답하는 방법이 보통 그래픽 프레임워크에서 널리 사용되긴 하지만 복잡한 상황을 처리하기엔 너무 단순한 경우가 종종 있다. 마우스 버튼을 누른 채 마우스를 움직이면서 발생하는 드래그 앤 드롭을 고려해보자. 드래그 앤 드롭은 공통 연산(common operation)이지만 꽤 복잡하다. 예를 들어, 픽셀에서 마우스의 이동은 요소의 계획에 반영되어야 하고 시각화는 새로고침(refresh)되어야 한다. 이는 공통 연산이기 때문에 프로그래머가 element on: ROMouseDrag do: [⋯] 와 같은 구조체를 굳이 사용하지 않아도 되도록 해준다.

대신 우리는 이벤트 핸들러의 재사용과 구성을 위한 가벼운 메커니즘으로 interactions를 제공한다. 이동이 불가한 요소를 드래그 가능하게 만들기 위해선 element @ RODraggable 을 이용하면 간단히 해결된다. @ 메서드는 addInteraction: 의 단축키다. 다른 상호작용(interactions)은 11.7 절에서 상세히 다루겠다.

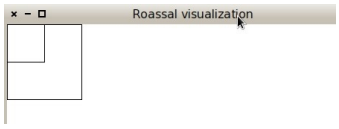
RODraggable 은 Roassal의 모든 상호작용의 루트에 해당하는 ROInteraction의 서브클래스이다. RODraggable은 마우스 드래그에 대한 마우스의 반응을 허용한다. 따라서 우리가 제시한 작은 예제는 아래와 같이 재정의된다.

```
view := ROView new.
element := ROElement new size: 100.
element
    + ROBorder "-> add shape"
    @ RODraggable. "-> add interaction"
view add: element.
view open.
```

추가 요소. 흥미로운 시각화는 많은 수의 요소를 포함하는 경향이 있다. 요소들은 ROView에서 add: 를 연속적으로 호출하여 하나씩 추가하거나, addAll: 를 한 번만 전송하여 한꺼번에 추가할 수 있다. 아래를 살펴보자.

위의 코드는 두 개의 사각형 요소가 있고 상단 좌측 모서리에 origin이 있는 창을 연다. 먼저 크기가 각각 50과 100 으로 된 요소를 두 개 생성하고, addAll: 메시지를 이용해 요소들을 뷰로 추가한다. 두 개의 요소는 테두리를 갖고 있으며 드래그 가능하게 만든다. 이 예제에서 모양

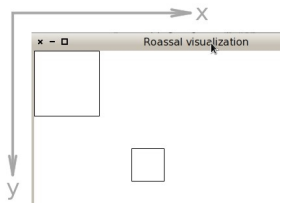
```
view := ROView new.
element1 := ROElement new size: 100.
element2 := ROElement new size: 50.
elements := Array with: element1 with:
    element2.
elements do: [:el | el + ROBorder @
    RODraggable ].
view addAll: elements.
view open.
```



과 상호작용은 뷰를 열기 전에 추가됨을 주목한다. 물론 뷰를 열고 난 후에 실행할 수도 있다. 그래픽 구성 요소들은 한 번만 추가하고 렌더링하면 마음껏 수정이 가능하다.

포인트를 매개변수로 하여 translateBy: 또는 translateTo: 를 전송하면 요소를 전환(translate) 할 수 있다. 매개변수는 단계(step) 나 위치를 픽셀로 표현한다. 축(axe) 은 그림 11.2에서 표시하고 있는데, x-축은 좌측에서 우측 방향으로 증가하며, y-축은 위에서 아래 방향으로 증가한다.

```
view := ROView new.
element1 := ROElement new size: 100.
element2 := ROElement new size: 50.
elements := Array with: element1 with:
    element2.
elements do: [:el | el + ROBorder @
    RODraggable ].
    view addAll: elements.
element2 translateBy: 150@150.
view open.
```



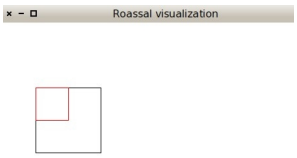
시각화가 하나 이상의 요소를 포함할 경우 각 요소를 자동으로 위치시키는 알고리즘이 있다면 좋겠다. 그러한 알고리즘은 레이아웃(layout) 이라 불린다. Roassal은 공간에 요소들을 위치시킴으로써 정렬시키는 레이아웃을 많이 제공한다. Roassal에서 레이아웃은 ROLayout의 서브클래스이다. 레이아웃은 11.5절에 설명되어 있다.

중첩 요소. ROElement 객체는 ROElement 요소를 포함할 수도 있다. 이러한 포함 관계를 중첩(nesting) 이라 부른다. 중첩은 요소를 트리 모양으로 구조화할 수 있게 해준다. 뿐만 아니라 아래 예제에서 볼 수 있듯이 자식들의 위치가 부모의 위치에 비례한다. 즉, 우리가 부모 노드를 전환하면 자식 노드들도 전환될 것이란 의미다.

중첩 요소들은 default별로 연장이 가능하기 때문에 자식 노드를 전환할 때 그 부모의 범위(bound) 는 새로운 위치에 해당 요소를 포함하도록 확장될 것이다.

각 요소는 resize(크기 조절) 전략을 갖고 있는데, 이는 resizeStrategy 인스턴스 변수에 보관된다. 기본적으로 resize 전략은 ROExtensibleParent의 인스턴스로서, 부모가 그 모든 자식 요소의 크기에 맞춰 자신의 범위를 확장시킬 것이란 의미다. 이용 가능한 resize 전략의 수는 많으며 ROAbstractResizeStrategy 클래스의 서브클래스를 살펴보면 되는데, 그 서브클래스

```
view := ROView new.
parent := ROElement new
    size: 100;
    + ROBorder.
children := ROElement new
    size: 50;
    + ROBorder red.
parent add: children.
view add: parent.
"Translate the parent"
parent translateTo: 50@100.
view open.
```




```
view := ROView new.
parent := ROElement new
    size: 100;
    + ROBorder.
children := ROElement new
    size: 50;
    + ROBorder red.
parent add: children.
view add: parent.
"Translate the children"
children translateTo: 50@100.
view open.
```

각각은 요소들이 사용하게 될 전략을 정의하기 때문이다.

지금까지 상호작용, 모양, 자식 요소를 소개하고, 객체 도메인(object domain)을 갖게 될 가능성을 간략하게 언급하였다. 표로 소개하자면 요소 표현은 그림 11.3과 같다.

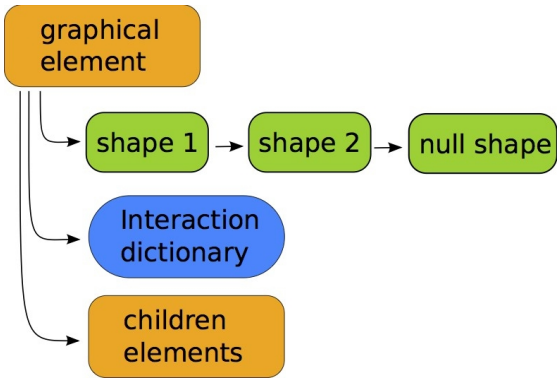


그림 11.3: ROElement 표현.

뷰의 카메라 이동하기. 뷰는 translateBy: 와 translateTo: 메시지에 응답하기도 한다. 위치는 뷰가 변경하는 것처럼 보이지만 사실상 그 카메라가 변경하는 것이다. ROCamera의 인스턴스

에 의해 표현되는 뷰의 카메라 구성 요소는 visualization 객체를 실제로 살펴보는 관점이다. 카메라에 관해서는 11.8절에서 찾아볼 수 있다.

Collection 계층구조 예제

예제로 우리는 본 장의 앞 부분에서 살펴본 Collection 계층구조 시각화를 생성할 것이다. 다음 단계를 통해 빌드할 것이다.

1. 특정 모양이 없는 데이터를 모두 추가하라. 이번 경우 데이터는 모든 서브클래스가 포함된 Collection 클래스다.
2. 특성에 따라 각 클래스를 렌더링하라.
3. 클래스와 그 슈퍼클래스 사이에 연결(link)을 추가하라.
4. 요소를 레이아웃이 있는 계층구조로 배열하라.

이번 절에서는 첫 단계, 계층구조의 각 클래스를 나타내는 모든 요소를 추가하는 것으로 시작하겠다.

이는 컬렉션으로부터 ROElements를 빌드하는 데 도움이 되는 ROElement 클래스로 forCollection: 메시지를 전송하여 쉽게 완료할 수 있다. 해당 메시지의 리턴 값으로부터 각 ROElement는 매개변수로부터 각 요소의 표현이다. 각각으로 border 모양을 추가하고, 쉬운 조작을 위해 드래그 가능하게 만든다. 마지막으로 뷰에서 모든 요소를 보기 위해 기본 레이아웃을 적용한다. 레이아웃이 어떻게 작용하는지는 추후에 더 설명하겠다.

```
view := ROView new.
classElements := ROElement forCollection: Collection withAllSubclasses.
classElements
    do: [:c | c + ROBorder.
        c @RODraggable ].
view addAll: classElements.
ROHorizontalLineLayout new on: view elements.
view open.
```

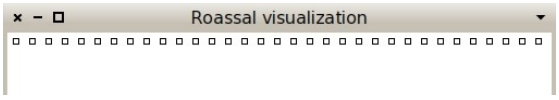


그림 11.4: 클래스를 나타내는 요소 추가.

모양 상세히 열거하기

그래픽 구성 요소는 ROShape의 서브클래스이거나 그러한 클래스의 인스턴스에 해당하는 인자를 가진 + (또는 addShape:) 메시지를 전송하면 모양이 주어진다.

이와 비슷한 메시지로 @ 가 있는데, 이 또한 기본 값을 오버라이드하기 위해 클래스 또는 인스턴스를 인자로 취할 수 있다.

+의 매개변수가 shape인 경우, 색상과 같은 속성이 채워지거나 테두리 색상이 개별적으로 설정될 수 있다. 클래스가 매개변수로서 전달되면 요소는 각 속성마다 기본 값을 가진 그 클래스의 인스턴스로 모양이 주어질 것이다.

이용 가능한 모양으로는 라벨(ROLabel), 테두리(ROBorder), 박스(ROBox), 원(ROEllipse)가 있다. 기본 값으로 ROLabel은 요소의 모델(예: 객체 도메인)에 대한 printString 값을 표시할 것이다. 해당 값은 그림 11.5에서 보이는 바와 같이 커스텀 텍스트를 설정하여 변경 가능하다. ROElement로 ROBorder, ROBox, ROEllipse를 적용할 경우 모양이 요소의 범위(bound)로 조정될 것이다. 색상, 테두리 색상, 테두리 너비와 같은 속성을 shape로 설정하는 것도 가능하다. 이를 그림 11.6, 그림 11.7, 그림 11.8에 표시하겠다.

```
ROElement new
    model: 'foo';
    size: 100;
    + ROLabel.
```

그림 11.5: 기본 값으로 된 ROLabel.

```
ROElement new
    size: 100;
    + ROBorder.
```

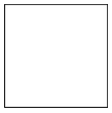


그림 11.6: 기본 값으로 된 ROBorder.

```
ROElement new
    size: 200;
    + (ROBox new
        color: Color green;
        borderColor: Color red;
        borderWidth: 4 ).
```

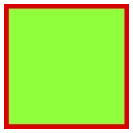


그림 11.7: 맞춤설정된 ROBox.

Shapes 구성하기. 좀 더 정교한 시각적 면을 생성하기 위해 shape들을 구성할 수 있다. 하나 이상의 ROShape로 형성된 요소를 가지려면 원하는 모양으로 + 메시지를 여러 번 전송하면 된다 (그림 11.9)

이는 요소에 연관된 shape들의 사슬을 구축하는데, 그 중 첫 번째 구성 요소는 마지막으로 추가된 모양이고 마지막 요소는 빈 모양(RONullShape)의 인스턴스다.

```
element := ROElement new
    size: 100.
shape := ROEllipse new
    color: Color yellow;
    borderColor: Color blue;
    borderWidth: 2.
element + shape.
```

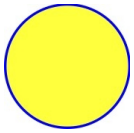


그림 11.8: 맞춤설정된 ROEllipse.

```
| element label border circle |
element := ROElement new
    size: 180.

label := ROLabel new
    text: 'composed shape'.
border := ROBorder new
    color: Color red.
circle := ROEllipse new
    color: Color yellow.
    borderWidth: 0.

element + label.
element + border.
element + circle.
```

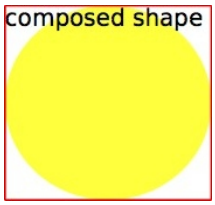


그림 11.9: shape들 구성하기.

Collection 계층구조 예제

이제 Collection 계층구조 예제에서 클래스로 몇 가지 모양을 추가할 것이다. 각 클래스 표현은 클래스의 인스턴스 변수 개수를 나타내는 너비와 그 메서드 개수를 나타내는 높이를 갖게 될 것이다.

```
view := ROView new.
classElements := ROElement forCollection: Collection withAllSubclasses.
classElements do: [:c |
    c width: c model instVarNames size.
    c height: c model methods size.
    c + ROBorder.
    c @ RODraggable ].
view addAll: classElements.
ROHorizontalLineLayout new on: view elements.
view open.
```

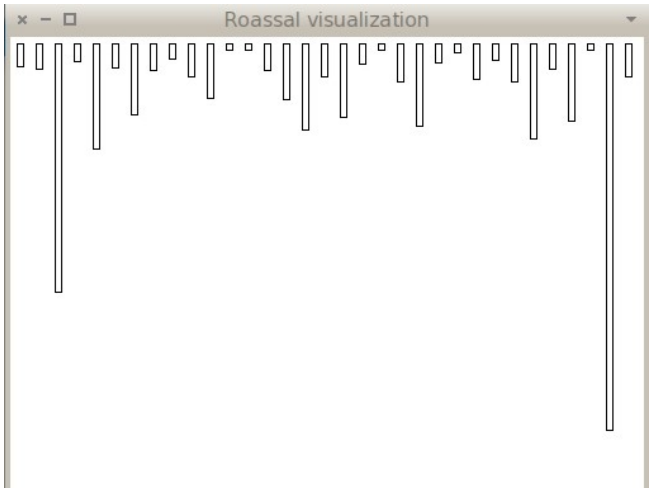


그림 11.10: 각 클래스에 대한 몇 가지 모양 추가하기.

Edges: 요소 연결하기

Roassal을 이용하면 요소들 간 관계를 나타내기 위해 요소들 간 링크를 빌드할 수 있다. 두 요소 간 링크는 ROEdge 클래스의 인스턴스이다. 기본 값으로 edge는 빈 모양에 해당하는 RONullShape의 인스턴스로 형성된다. 이 때문에 edge를 렌더링하기 위해서는 직선 모양으로 형성될 필요가 있는데, 직선 모양은 ROAbstractLine의 어떤 서브클래스든 가능하다. 아래 코드는 두 요소 간 edge의 생성을 묘사한다. 먼저 두 개의 요소를 생성할 것이다. 다음으로 두 요소를 매개변수로서 사용하는 edge를 생성하여 직선 (ROLine의 인스턴스) 모양으로 구성할 것이다. 마지막으로 두 개의 요소와 edge를 뷰로 추가한다.

```
vview := ROView new.
start := (ROElement on: 1) size: 20; + ROB
    order red.
end := (ROElement on: 2) size: 20; + ROBor
    der red.
end translateBy: 50@50.

edge := ROEdge from: start to: end.
edge + ROLine.
view
    add: start;
    add: end;
    add: edge.
view open.
```

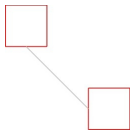


그림 11.11: 간단한 edge.

edge에 모양 추가하기. 표준 모양 외에도 직선 모양의 유형에는 여러 가지가 있는데, 가령 ROOrthoHorizontalLineShape를 예로 들 수 있겠다. 이들은 모두 ROAbstractLine 클래스의 서브클래스로서 ROLine도 마찬가지다. 몇 가지 예를 그림 11.12와 그림 11.13에서 소개하겠다.

```
edge + ROLine.
```

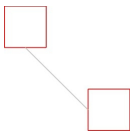


그림 11.12: 간단한 edge.

```
edge + ROOrthoHorizontalLineShape.
```

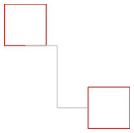


그림 11.13: 가로 방향(horizontally oriented)의 직각 edge.

직선에 화살표 추가하기. 직선은 하나 또는 이상의 화살표를 포함할 수 있다. 화살표는 ROAbstractArrow의 서브클래스의 인스턴스로서, ROArrow 또는 ROHorizontalArrow를 들 수 있다. 직선 모양에 화살표를 추가하기 위해 add: 메시지를 사용하는데, 이는 그림 11.14와 그림 11.15에 표시하겠다.

```
edge + (ROLine new add: ROArrow new).
```

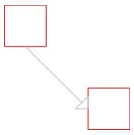


그림 11.14: 화살표가 표시된 edge.

```
edge + (ROOrthoHorizontalLineShape new
    add: ROHorizontalArrow new)
```

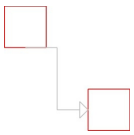


그림 11.15: 가로 방향의 화살표가 있는 직각 edge.

기본 값으로 화살표는 edge 끝에 위치할 것이지만 해당 위치는 add:offset: 을 이용해 맞춤 설정이 가능하다. 오프셋 매개변수는 0과 1 사이에 해당하는 수치여야 하고, 화살표가 위치한 직선 길이의 비율을 나타낸다. 예를 들어, 오프셋이 0.5일 경우 화살표는 그림 11.16과 같이 직선의 중간으로 설정된다.

```
edge + (ROLine new add: ROArrow new
    offset: 0.5).
```

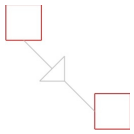


그림 11.16: 중간에 화살표가 있는 edge.

직선이 하나 이상의 화살표를 포함하는 경우 화살표마다 여러 개의 오프셋을 설정할 수 있다.

```
line := ROLine new.
line add: ROArrow new offset: 0.1.
line add: ROArrow new offset: 0.5.
edge + line.
```

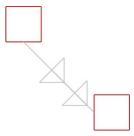


그림 11.17: 화살표가 두 개인 edge.

Collection 계층구조 예제

이제 요소 간 링크를 만드는 방법을 숙지했을 것이다. 다음 코드를 이용해 각 클래스와 그 슈퍼클래스 간 edge를 생성할 수 있다. 이를 위해선 먼저 edge를 빌드하기 위해 연관(association)의 컬렉션을 생성할 필요가 있다. 각 연관은 시작점을 연관 키로 나타내고 끝지점은 연관 값으로 나타낸다. 해당 예제에서 각 연관은 클래스를 표현하는 ROElement부터 시작해 그 슈퍼클래스를 표현하는 ROElement까지 이어진다.

연관을 갖고 있다면 linesFor: 메시지를 전송하여 ROEdge의 인스턴스를 생성한다. 해당 메시지는 연관의 컬렉션을 매개변수로 취하고 edge의 컬렉션을 리턴한다.

```
view := ROView new.
classElements := ROElement forCollection: Collection withAllSubclasses.
view addAll: classElements.
associations := OrderedCollection new.
classElements do: [:c |
    c width: c model instVarNames size.
    c height: c model methods size.
    c + ROBorder.
    c @ RODraggable.
    (c model superclass = Object)
        ifFalse: [ associations add: ((view elementFromModel: c model superclass) -> c)]
    ].
edges := ROEdge linesFor: associations.
view addAll: edges.
ROHorizontalLineLayout new on: view elements.
view open
```

이제 Collection 계층구조에 원하는 모양으로 된 클래스를 갖고 있으며 클래스는 각자의 슈퍼클래스로 연결되었다. 하지만 실제 계층구조가 보이지 않는다. 그 이유는 뷰의 모든 요소를

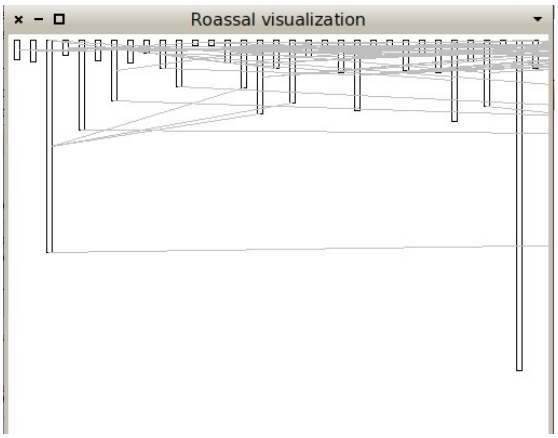


그림 11.18: 각 클래스와 그 슈퍼클래스 간 링크 추가하기.

배열하기에 적절한 레이아웃이 필요하기 때문이다. 다음 절에서는 레이아웃을 요소로 적용하는 방법을 다루고자 한다.

레이아웃

레이아웃은 요소의 컬렉션이 어떻게 자동으로 배열되는지를 정의한다. 레이아웃을 적용하기 위해서는 ROElement의 컬렉션을 매개변수로 한 on: 메시지를 사용해야 한다. 그림 11.19의 예제와 같이 spriteOn: 라는 편의(convenience) 메시지를 이용해 각각의 크기가 50이고 빨간색 테두리를 가지며 드래그가 가능하게 구성된 ROElements의 컬렉션을 생성한다. 이후 그리드(grid)에 요소를 배열하기 위해 레이아웃을 적용하겠다.

```
view := ROView new.
view addAll: (ROElement spritesOn: (1 to:
    4)).
ROGridLayout on: view elements.
view open.
```

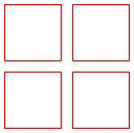
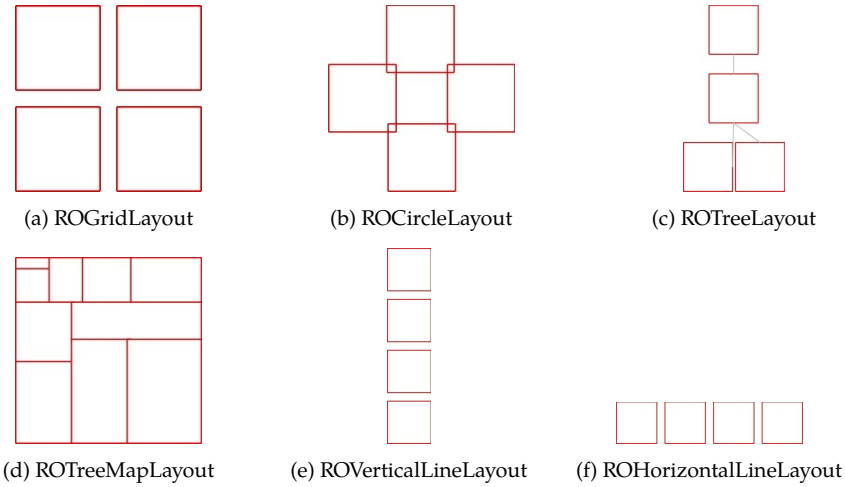


그림 11.19: ROGridLayout가 ROElements의 그룹에 적용된 모습.

그림 11.20은 Roassal에서 이용 가능한 레이아웃을 몇 가지 보여준다. 본문에 실리지 않은 것을 포함해 아래의 레이아웃은 ROLayout의 서브클래스로 찾을 수 있다.

레이아웃이 요소의 컬렉션으로 적용되면서 여러 요소의 집합들은 여러 레이아웃을 가질 수 있다. 아래 예제에서는 두 개의 요소 컬렉션이 두 개의 레이아웃으로 배열된다. 첫 번째는 요소를 수직선을 따라, 두 번째는 수평선을 따라 배열한다. 먼저 수직선을 따라 배열된 요소들을 생성하고, ROVerticalLineLayout을 적용하여 label을 이용해 모양을 형성한다. 이후 두 번째



(a) ROGridLayout (b) ROCircleLayout (c) ROTreeLayout

(d) ROTreeMapLayout (e) ROVerticalLineLayout (f) ROHorizontalLineLayout

그림 11.20: 이용 가능한 레이아웃 몇 개를 요소의 그룹에 적용시켰을 때

그룹에는 ROHorizontalLineLayout을 이용해 똑같이 실행하고 겹침(overlapping)을 피하기 위해 공간을 둔다.

```
|| view verticalElements horizontalElements |
view := ROView new.

verticalElements := ROElement spritesOn: (1 to: 3).
ROVerticalLineLayout on: verticalElements.
verticalElements do: [ :el | el + ROLabel ].

horizontalElements := ROElement spritesOn: (4 to: 6).
ROHorizontalLineLayout on: horizontalElements.

horizontalElements do: [ :el |
    el + ROLabel.
    el translateBy: (60@ 0) ]. "spacing"
view
    addAll: horizontalElements;
    addAll: verticalElements.
view open.
```

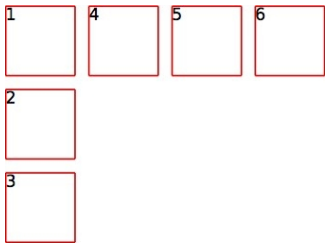


그림 11.21: 여러 개의 레이아웃을 요소들의 여러 집합으로 적용하기.

중첩된 구조 내 레이아웃. 중첩된 요소의 레이아웃은 각 요소의 컨테이너에 비례한다. 아래 예제에서는 두 개의 요소가 생성되는데, 각 요소는 그리드로서 배열된 3개의 자식 요소들을 갖는다. 마지막으로 수평선 레이아웃을 이용해 부모 요소들을 배열하고자 한다.

```
vview := ROView new.
elements := (ROElement spritesOn: (1 to:
    2)).
elements
    do: [:el | el addAll: (ROElement spr
            itesOn: (1 to: 3)).
        "arranging the children nodes"
        ROGridLayout on: el elements.].
view addAll: elements.
ROHorizontalLineLayout on: view elements.
view open.
```

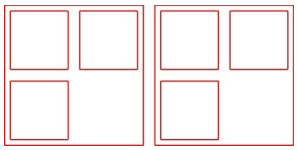


그림 11.22: 여러 개의 레이아웃이 있는 중첩된 요소들.

새 레이아웃 생성하기. Roassal은 많은 수의 레이아웃을 제공한다 (이 글을 작성할 무렵엔 약 23개의 레이아웃이 존재). 하지만 특정 표현을 수용하기 위해 새 레이아웃을 필요로 하는 경우가 발생할지도 모른다. 이번 절에서는 레이아웃을 집중적으로 다룬다. 새 레이아웃을 생성하기 전에 먼저 레이아웃이 어떻게 구조화되는지를 이해해야 할 것이다.

모든 레이아웃 클래스는 ROLayout으로부터 상속된다. 해당 클래스는 클래스 측이나 인스턴스로부터 레이아웃을 적용하는 데에 가장 흔히 사용되는 메서드인 on: 을 정의한다. on: 메서드는 레이아웃을 적용하는 데 주요 메서드인 executeOnElements: 를 호출한다. 이 메서드를 아래 코드에 표시하겠다.

```
ROLayout >> executeOnElements: elements
    "Execute the layout, myself, on the elements"
    maxInterations := elements size.
    self doInitialize: elements.
    self doExecute: elements asOrderedCollection.
    self doPost: elements.
```

executeOnElements: 메서드는 아래 3개의 hook 메서드를 호출한다.

1. **doInitialize:** 레이아웃을 시작하기 전에 실행되는 메서드. 정렬해야 하는 그래프를 준비해야 하는 경우 유용하다.

2. **doExecute:** 레이아웃 알고리즘을 적용한다. 그에 따라 요소들이 이동된다.

3. **doPost:** 레이아웃을 실행한 이후에 실행되는 메서드.

Pre/post-processing 이 정의될 수도 있다. 이는 레이아웃이 다단계이거나 적절한 이벤트를 방출해야 하는 경우 유용하다. 이러한 액션은 ROLayoutBegin과 ROLayoutEnd 이벤트를 이용해 콜백으로서 설정된다. ROLayoutBegin과 ROLayoutEnd는 각각 doInitialize: 와 doPost: 에 의해 발표(announce)된다. 그 사용 예제를 아래 코드에 소개하겠다.

```
| layout t |
t := 0.
layout := ROHorizontalLineLayout new.
layout on: ROLayoutBegin do: [:event | t := t + 1].
layout on: ROLayoutEnd do: [:event | t := t + 1].
layout applyOn: (ROElement forCollection: (1 to: 3)).

self assert: (t = 2).
```

doExecute: 메서드는 특정 알고리즘을 이용해 요소들을 배열한다. 이 메서드는 배치할 요소들의 컬렉션을 매개변수로서 취한다.

이제 ROLayout 클래스의 구조를 알게 되었으니 RODiagonalLineLayout 이라 불리는 새 레이아웃을 정의하여 대각선을 따라 요소를 위치시킬 것이다. ROLayout의 서브클래스를 생성하는 것이 첫 번째 단계가 되겠다.

```
RROLayout subclass: #RODiagonalLineLayout
    instanceVariableNames: 'initialPosition'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Roassal-Layout'
```

인스턴스 변수 initialPosition 는 가상 직선이 어디서 시작하는지, 다시 말하자면 직선의 첫 번째 요소가 어디에 위치할 것인지 정의한다. 해당 변수는 initialize 메서드에서 설정된다.

```
RODiagonalLineLayout >> initialize
    super initialize.
    initialPosition := 0@0.

RODiagonalLineLayout >> initialPosition: aPoint
    initialPosition := aPoint

RODiagonalLineLayout >> initialPosition
    ^ initialPosition
```

레이아웃이 적용되기 전이나 후에 특수 액션을 실행할 필요가 있다면 doInitialize: 또는 doPost: 메서드를 오버라이드할 것이다. 하지만 본문의 예제는 이에 해당하지 않는다. 우리가 겹쳐 써야 하는 메서드는 사실상 작업을 실행하는 doExecute: 로, 이 메서드는 가상의 대각선을 따라 모든 요소를 이동하는 일을 수행한다.

```
RODiagonalLineLayout >> doExecute: elements
    | position |
    position := initialPosition.
    elements do: [:el |
        el translateTo: position.
        position := position + el extent ]
```

아래 코드를 이용해 레이아웃을 테스트해볼 수 있다.

```
| view elements |
view := ROView new.
elements := ROElement spritesOn: (1 to: 3).
view addAll: elements.
RODiagonalLineLayout on: view elements.
view open.
```

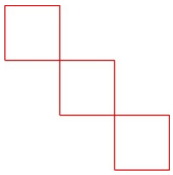


그림 11.23: 요소의 컬렉션에 적용된 Diagonal Line 레이아웃.

Roassal에서 레이아웃에 대한 한 가지 주요점으로 배치할 요소의 크기를 고려하는 것을 들 수 있다. 새 레이아웃을 정의할 때는 자신의 알고리즘이 요소의 크기를 사용하도록 만들 것을 명심하라.

Collection 계층구조 예제

Collection 예제에 대한 계층구조가 필요하므로 적절한 시각화를 얻기 위해선 ROTreeLayout 이 유용하겠다.

```
""Create the elements to be displayed"
view := ROView new.
classElements := ROElement forCollection: Collection withAllSubclasses.
view addAll: classElements.

associations := OrderedCollection new.

classElements do: [:c |

    "Make each element reflect their model characteristics"
    c width: c model instVarNames size.
    c height: c model methods size.

    "we add shape for the element to be seen"
    c + ROBorder.

    "we make it draggable by the mouse"
    c @ RODraggable.

    "Create associations to build edges"
    (c model superclass = Object)
    ifFalse: [ associations add: ((view elementFromModel: c model superclass) -> c)]
    ].

"Add edges between each class and its superclass"
edges := ROEdge linesFor: associations.
view addAll: edges.

"Arrange all the elements as a hierarchy"
ROTreeLayout new on: view elements.
view open
```

그 결과 시각화는 그림 11.24의 모습과 같을 것이다.

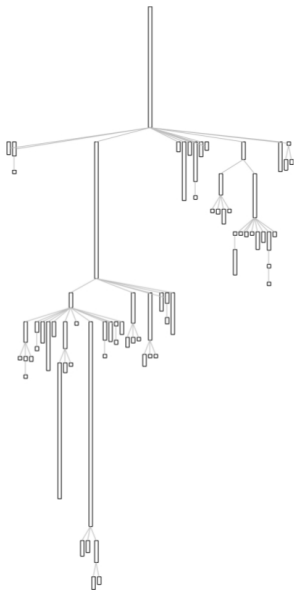


그림 11.24: 너비는 인스턴스 변수의 개수를 표현하고 높이는 메서드의 개수를 표현하는 Collection 클래스 계층구조.

이벤트와 콜백

Roassal은 뷰 자체를 포함해 시각성 내 어떤 시각적 구성 요소든 이벤트를 방출하고 그에 반응하도록 허용한다. Roassal에서 정의되는 이벤트에는 두 가지 종류가 있다. 첫 번째 이벤트는 저수준으로서 마우스 이동이나 클릭, 또는 키 누름과 같은 사용자 액션을 나타낸다. 두 번째 이벤트는 뷰 자체가 트리거한 이벤트를 의미하는데 주로 카메라의 이동, 레이아웃 적용, 뷰의 새로고침이 포함된다. 모든 이벤트는 ROEvent 클래스로부터 상속된다.

이벤트가 작동하는 방식을 확인하기 위해 마우스 클릭에 반응하는 시각화를 예로 들 예정인데, 클릭이 이루어지는 곳으로 요소를 전환한다. 마우스 이벤트를 처리하는 이벤트 클래스로는 ROMouseClick, ROMouseMove, ROMouseEnter, ROMouseLeave가 있으며, 키 누름을 처리하는 이벤트로 ROKeyDown 클래스가 있다.

우선 ROLeftMouseClick 이벤트를 이용해 마우스 왼쪽을 클릭하면 시각화가 반응하도록 만들 것이다. 반응은 이벤트의 위치로 요소를 전환하기 위한 애니메이션을 생성할 것이다.

아래 코드에서와 같이 Roassal 객체가 이벤트에 반응하도록 설정하기 위해 on:do: 메시지를 이용한다. 첫 번째 매개변수는 예상되는 이벤트의 클래스여야 하고, 두 번째 매개변수는 이벤트를 수신 시 실행되어야 하는 액션을 정의하는 블록이어야 한다.

```
view := ROView new.
el := ROElement sprite.
view add: el.
view
    on: ROMouseLeftClick
    do: [ :event | ROLinearMove new for: el to: event position ].
view open.
```

ROLinearMove는 Roassal 상호작용 중 하나에 해당한다. 이름이 제시하듯이 선형적 이동(linear move)으로 전환되어야 하는 요소에 대한 애니메이션을 생성한다. 상호작용에 관한 내용은 다음 절에서 더 설명하겠다.

상호작용 계층구조

그래픽 요소는 콜백이나 상호작용을 설정함으로써 이벤트에 응답한다. 콜백을 어떻게 설정하는지는 이미 제시하였으니 이번 절에서는 상호작용에 관해 상세히 다루겠다.

모든 Roassal 상호작용의 루트 클래스는 ROInteraction이다. 상호작용은 ROInteraction의 서브클래스나 그러한 클래스의 인스턴스를 매개변수로 하여 @ 메시지를 전송함으로써 요소로 설정된다. 요소로 적용 가능한 상호작용은 다양한데, RODraggable 이나 ROGrowable 을 예로 들 수 있겠다. RODraggable 은 요소를 마우스로 드래그 가능하게 해주며, ROGrowable 은 클릭 시 요소의 크기를 증가시킨다.

요소는 하나 이상의 상호작용을 가질 수 있다. 예를 들자면 요소로 RODraggable이나 ROGrowable을 적용할 수도 있는데, 아래 코드를 통해 설명하겠다. 요소를 클릭하여 크게 만들거나 뷰로 드래그하라.

```
| view element |
view := ROView new.
element := ROElement new size: 10.
element
    + ROBox;
    @ RODraggable;
    @ ROGrowable.
view add: element.
view open.
```

마우스를 요소 위에서 클릭하면 표시되는 팝업 요소처럼 일부 상호작용은 준비하기가 조금 복잡한 것이 사실이다.

Roassal에서 이용 가능한 상호작용 중 몇 가지만 본문에 제시하겠다.

ROAbstractPopup

ROAbstractPopup은 팝업을 표시함으로써 요소로 하여금 이벤트 위의 마우스에 반응하도록 해준다. 팝업에는 두 가지 유형이 있는데, 첫 번째는 기본 값으로 요소 모델의 printString 값과 함께 상자를 표시하는 ROPopup, 두 번째는 커스텀 뷰를 표시하는 ROPopupView가 있다.

요소에 팝업을 추가하려면 ROPopup 클래스를 인자로 하여 @ 메시지를 전송하면 된다. 문자열을 매개변수로 한 text: 메시지를 이용해 커스텀 텍스트를 마련하는 것도 가능하다.

아래 예제에서는 임의의 문자열을 그 모델로 하여 ROElement 클래스로 spriteOn: 메시지를 전송함으로써 요소를 생성하고자 한다. 그 결과 요소의 크기는 50이 되고, 빨간 테두리를 가지며, 마우스로 드래그가 가능해진다. 마지막으로 요소에 ROPopup을 추가한다.

```
view := ROView new.
el := ROElement spriteOn: 'baz'.
el @ ROPopup. "Or with custom text -> (ROPopup text: 'this is custom text')"
view add: el.
view open.
```

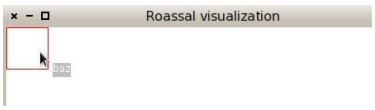


그림 11.25: ROPopup.

ROPopupView는 팝업시킬 뷰의 정의를 필요로 하기 때문에 조금 더 복잡하다. 이러한 상호작용은 표시할 새로운 뷰가 있는 ROpopupView로 view: 메시지를 전송하면 생성할 수 있다. 매개변수는 뷰를 정의하는 블록이 될 수도 있다. 마우스가 요소 위에 있으면 같은 요소를 매개변수로 이용해 블록이 평가되며, 이에 따라 뷰가 동적으로 생성될 수 있다.

아래 예제는 5개의 요소가 있는 뷰를 생성한다. 각 요소 위로 마우스를 갖다 대면 팝업이 표시되면서 반응한다. 팝업 뷰는 마우스가 위치한 요소 모델과 같은 개수의 노드를 가진 뷰를 생성하는 블록으로 정의된다. 예를 들어, 그림 11.7에서 볼 수 있듯이 마우스가 노드 "3" 위로 지나가면 3개의 회색 상자가 있는 팝업창이 나타난다.

```
view := ROView new.
elements := ROElement spritesOn: (1 to: 5).
"create the view to popup"
viewToPopup := [ :el | | v |
    v := ROView new.
    "Add as many elements as the value represented"
    v addAll: (ROElement forCollection: (1 to: el model)).
    v elementsDo: [ :e | e size: 20; + ROBox ].
    ROGridLayout on: v elements.
    v ].
elements do: [ :e | e + ROLabel; @ (ROPopupView view: viewToPopup)].
view addAll: elements.
ROHorizontalLineLayout on: view elements.
view open.
```

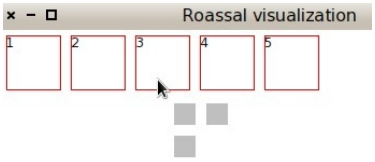


그림 11.26: 마우스 아래의 요소 모델과 같은 개수의 요소를 가진 뷰를 생성하는 ROPopupView.

RODynamicEdge

데이터 요소와 그 관계의 시각화를 반복해야 하는 경우 마우스가 요소를 가리키면 나가는 edge가 표시된다. 요소로 들어가거나 떠날 때 콜백의 올바른 조합을 시도하는 대신 RODynamicEdge 상호작용을 사용할 경우 작업을 상당히 줄여준다.

아래 예제는 마우스를 일부 요소 위에서 맴돌 때 몇 행이 표시되도록 한다.

```
| rawView el1 el2 el3 |
rawView := ROView new.
rawView add: (el1 := ROBox element size: 20).
rawView add: (el2 := ROBox element size: 20).
rawView add: (el3 := ROBox element size: 20).
ROCircleLayout on: (Array with: el1 with: el2 with: el3).

el1 @ RODraggable.
el2 @ RODraggable.
el3 @ RODraggable.

el1 @ (RODynamicEdge toAll: (Array with: el2 with: el3) using: (ROLine arrowed color: Color red)).

rawView open
ROAnimation
```

애니메이션 또한 Roassal에서 상호작용에 해당한다 (예: ROAnimation은 ROInteraction의 서브클래스다). 일부 애니메이션은 요소들이 일정한 속도(ROLinearMove), 일정한 가속(ROMotionMove), 또는 수학 함수에 따라(ROFunctionMove) 선형적으로 전환되도록 허용한다. ROZoomInMove와 ROZoomOutMove 클래스가 제공하는 애니메이션은 뷰를 포커스인, 포커스 아웃시킨다. 모든 애니메이션은 ROAnimation의 서브클래스이다.

각 애니메이션에는 완료해야 할 다수의 사이클이 있는데, 각 사이클은 doStep 메시지를 전송하여 실행된다. ROAnimation은 after: 메시지를 이용하여 애니메이션이 완료된 후 실행되어야 하는 블록을 설정하도록 해준다. 애니메이션이 완료된 후 실행되어야 하는 액션은 애니메이션이 트리거되기 전에 설정되어야 하며, 이를 어길 경우 실행되지 않음을 주목하라.

```
view := ROView new.

element := ROElement new.
element size: 10.
element + (ROEllipse color: Color green).

view add: element.
element translateBy: 30@20.

ROFunctionMove new
    nbCycles: 360;
    blockY: [ :x | (x * 3.1415 / 180) sin * 80 + 50 ];
    on: element.
view open.
```

image:DeepintoPharo Image 11-27.jpg(nothing)

그림 11.27은 ROLinearMove를 나타낸다. 아래 코드는 요소가 ROFunctionMove를 이용해 sinus 곡선을 따르도록 해준다.

뷰의 카메라 이해하기

뷰의 카메라는 실제로 뷰를 바라보는 관점을 나타낸다.

translateBy: 또는 translateTo: 메시지가 뷰로 전송되면 뷰 자체가 아니라 사실상 그 카메라가 이동한다. 카메라의 위치는 position 인스턴스 변수에 의해 주어진다. 카메라의 위치는 수동으로 translateBy: 또는 translateTo: 메시지를 카메라로 전송하여 설정되지만 매개변수의 값으로 부정 값(negated value)을 사용한다. 이는 뷰를 10 픽셀만큼 가로 및 세로로 이동해야 하는 경우 아래와 같이 실행할 수 있음을 의미한다.

```
view translateBy: 10@10
```

혹은 뷰의 카메라를 수동으로 전환하는 방법도 있다.

```
view camera translateBy: (-10)@(-10)
```

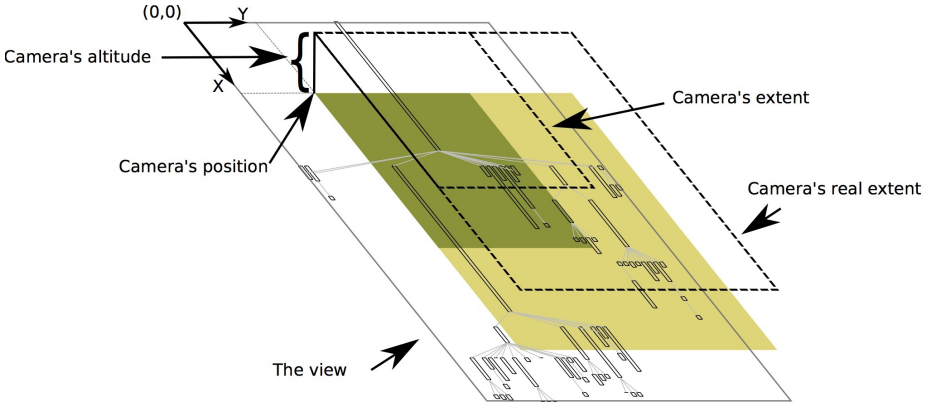
카메라에는 우리가 실제로 바라보는 extent(범위), 그리고 더 먼 범위를 나타내는 real extent(실제 범위)가 있다. 실제 뷰의 카메라의 범위는 캔버스에 뷰가 그려지는 방식에 영향을 미친다. 뷰를 렌더링할 때는 각 포인트, 각도 혹은 그려야 하는 다른 모양이 카메라의 범위에 따라 구성된다. 이는 카메라의 시야에 비례한 가상 포인트에서 각 절대 위치를 변형함으로써 이루어진다. 예를 들어 뷰에서 이를 확대하면 extent 상의 내용이 "확장"되어 real extent를 채우고, 객체의 크기는 더 커진다. 카메라의 extent와 real extent는 각각 extent: 와 realExtent: 접근자를 이용해 수정된다. 카메라는 시각화의 창 크기를 보관하기도 한다.

카메라는 extent를 이용해 계산되는 뷰의 높이(altitude)를 갖는다. extent 값이 작을수록 카메라는 낮게 위치하고, extent 값이 크면 카메라는 높게 위치한다. 카메라의 높이는 숫자를 매개변수로 이용하여 altitude: 메시지를 전송함으로써 설정된다. 카메라는 회전이 불가하며 전환만 가능하다. 이는 카메라가 항상 뷰를 수직으로 바라봄을 의미한다.

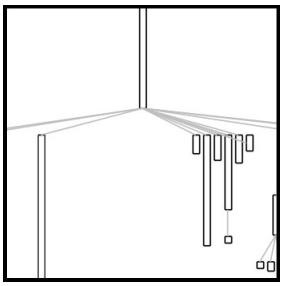
그림 11.28은 방금 언급한 내용을 표시하는데, 뷰와 연관된 정보를 모두 나타낸다. 시각화의 시각적 부분이 카메라의 extent에 의해 주어진다는 것도 확인할 수 있겠다.

ROZoomMove 상호작용은 카메라의 범위에 영향을 미친다. 이러한 상호작용은 카메라의 위치를 수정하고 원하는 직사각형에 들어맞도록 확장시킨다. 예를 들어, 뷰의 특정 요소를 포커스 인 하기 위해 확대하면 ROZoomMove는 요소의 경계에 들어맞도록 카메라를 전환하고 확장한다. 이러한 움직임은 카메라의 altitude를 변경하여 시뮬레이션이 가능하다.

카메라를 이용해 탐색을 위한 미니맵(minimap) 빌드하기. Roassal이 제공하는 상호작용과 애니메이션 모델은 복잡한 행위를 지원한다. 아래 코드를 고려해보자.



(a) Camera Diagram



(b) Camera extent, showing what it is actually seen

그림 11.28: 뷰의 카메라의 구성 요소

```
| view eltos |
view := ROView new.
view @ RODraggable .
view on: ROMouseRightClick do: [ :event |
    ROZoomInMove new on: view ].

view on: ROMouseLeftClick do: [ :event |
    ROZoomOutMove new on: view ].

eltos := ROElement spritesOn: (1 to: 400).
eltos do: [:el | el + ROLabel ].
view addAll: eltos.
ROGridLayout new on: view elements.

"Mini map opens by pressing m"
view @ ROMiniMap.
view open.
```

이 코드는 400개의 라벨이 붙은 요소가 있는 뷰를 연다. 요소들은 그리드 레이아웃을 이용해 정렬된다. 왼쪽 마우스 버튼을 클릭하면 뷰가 확대된다. 오른쪽 마우스를 클릭하면 축소된다.

m 키를 누르면 미니맵이 열릴 것이다. 해당 기능은 ROMiniMap 상호작용을 이용해 활성화된다.

ROMiniMap은 시각화의 완전한 시야를 제공하는 새 창을 연다. 뿐만 아니라 본래 뷰의 카메라를 이용함으로써 탐색을 수월하게 해준다.

미니맵은 시각화의 작은 버전, 그리고 메인 뷰의 창에서 현재 시각적 부분을 나타내는 lupa (확대경)로 구성된다.

본문 예제로 돌아와, 상호작용은 뷰로 @ROMiniMap 메시지를 전송하여 추가하고 "m"을 누르면 뷰가 열린다 (그림 11.29 참고).

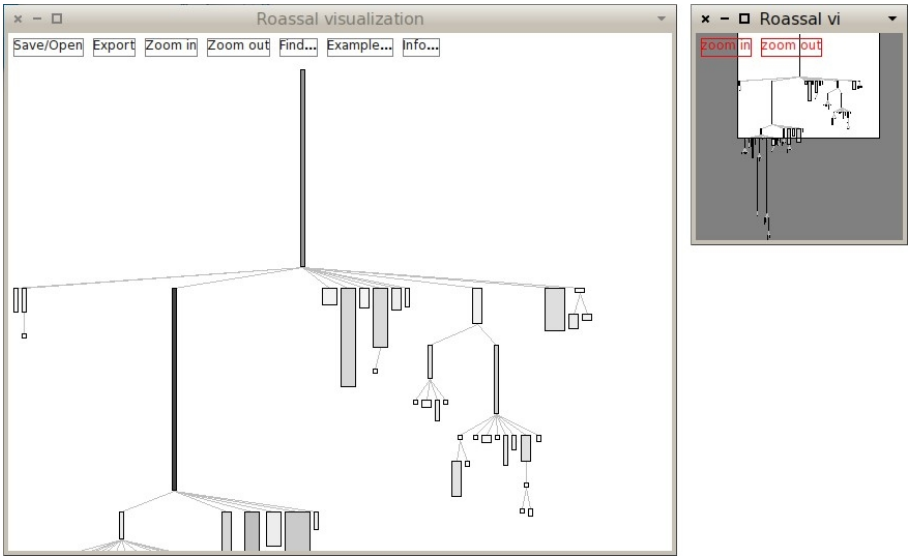


그림 11.29: ROMiniMap을 Collection 계층구조 예제에 적용한 모습.

뷰의 더 작은 버전은 특정 모양, 즉 ROViewDisplayer의 서브클래스인 ROMiniMapDisplayer를 이용해 표시된다. ROViewDisplayer는 요소에 뷰를 표시하는 모양이다 (기본적으로 팝업 뷰에 사용됨). 두 가지의 차이점은 ROMiniMapDisplayer는 고유의 카메라를 사용한다는 점으로, 뷰의 카메라와는 다른 extent를 가진다. 따라서 동일한 뷰지만 여러 크기로 볼 수 있도록 해준다.

Lupa 크기는 창의 시각적 부분을 표현하고 그 위치는 뷰의 카메라 위치에 연관된다. 뷰가 포인트로 전환되면 lupa는 그 위치를 변경하여 따르는데, 카메라 위치를 나타내는 포인트는 ROMiniMapDisplayer 카메라 extent 상의 포인트로 전환된다. 뷰를 확대하거나 축소하면 카메라의 extent가 변경되어 lupa의 크기가 증가하거나 감소한다.

Pharo를 넘어서

Roassal은 다른 스몰토크 dialect로 쉽게 이식되도록 설계되었다. 현재는 VisualWorks, Amber, VA Smalltalk로 이식되고 있다.

그림 11.30이 나타내는 바와 같이 Roassal은 3가지 구성 요소로 이루어진다.

- Roassal Core. 메인 클래스를 정의하는 패키지의 집합으로, ROView, ROElement, ROShape, ROCamera가 이에 해당한다. 이는 모든 테스트를 포함하기도 한다.
- Mondrian DSL. Roassal-Builder와 Roassal-Builder-Tests 패키지로 구성된다.
- 플랫폼 의존적인 패키지. Roassal이 이식되는 스몰토크 dialect 전용으로 존재한다.

플랫폼 의존적인 패키지에는 몇 가지 클래스가 구현되어야 한다. 주요 클래스로는 뷰를 렌더링할 수 있는 native canvas 클래스, 캔버스를 포함하도록 객체를 리턴하고 모든 외부 이벤트를 위임할 수 있는 위젯 팩토리 클래스를 들 수 있겠다. 첫 번째는 ROAbstractCanvas이며, 두 번째는 RONativeWidgetFactory의 서브클래스여야 한다.

ROPlatform 클래스는 의존적 패키지와 코어 패키지 간 연계(bridge)를 어떻게 구현해야하는지 정의한다. 해당 클래스는 이름에 따라 클래스를 보관하는 canvasClass와 widgetFactory와 같은 인스턴스 변수를 정의한다. 각 플랫폼 의존적 패키지는 자신의 플랫폼 클래스를 ROPlatform의 서브클래스로서 구현하고, 구현된 모든 플랫폼 의존적 클래스를 참조해야 한다. 내부적으로 이러한 클래스 중 하나라도 필요할 때마다 코어 패키지는 필요한 클래스를 리턴하기 위해 ROPlatform의 현재 인스턴스에 의존한다.

그림 11.30: Roassal 구조.

요약

Roassal은 객체의 어떤 그래프든 시각화할 수 있도록 해준다. 본 장에서는 Roassal의 주요 기능을 살펴보았다.

- 그래픽 요소를 생성하고 원하는 모양대로 형성한다.
- edge를 생성하여 그래픽 요소들 간 관계를 나타낸다.
- 요소의 컬렉션을 자동으로 배열하기 위해 레이아웃을 적용한다.

- 콜백과 정의된 상호작용을 설정함으로써 요소들이 이벤트에 반응하도록 만든다.
- 뷰의 카메라와 상호작용을 통해 시각화 포인트를 이동시킨다.

`http://objectprofile.com` 에서는 Rossal에 관련된 스크린샷, 온라인 예제, 스크린캐스트를 찾을 수 있다.

감사의 말. 우선 Roassal을 개발해준 Chris Thorgrimsson과 ESUG에 감사의 말을 전한다.

본문을 검토해준 Nicolas Rosselot Urrejola 와 Stephane Ducasse에게 매우 감사드린다. 또 Roassal의 디자인에 관한 여러 논의를 제공해주신 Pietriga와 Tudor Girba에게도 감사를 표하는 바이다.

제 12 장

Mondrian을 이용해 시각화 스크립팅하기

많은 양의 데이터에 의미를 부여하기란 적절한 툴이 없이는 힘들다. 텍스트 출력(textual output)은 그 표현의 유연성(expressiveness)과 상호작용의 지원에 제한이 있는 것으로 알려져 있다.

Mondrian은 시각화의 스크립팅에 있어 도메인 특정 언어에 해당한다. 그 구현은 Roassal 위에서 실행된다 (제 11장 참고). 이는 객체와 그 관계와 관련해 정의된 임의의 데이터를 시각화하고 그와 상호작용한다. Mondrian은 소프트웨어 평가 활동에 주로 사용된다. Mondrian은 소프트웨어 소스 코드의 시각화에 뛰어난 기능을 보인다. 이번 장에서는 Mondrian의 원리를 소개하고 재빠르게 데이터를 구성하기 위한 그 표현식 명령을 설명하고자 한다. 더 상세한 정보는 해당 주제를 본격적으로 다룬 Moose book[1]의 관련 장을 참고하길 바란다. 본문을 읽고 나면 상호작용적이고 시각적인 표현을 생성할 수 있을 것이다.

설치 및 첫 시각화

Mondrian은 Roassal의 일부이다. 설치 절차는 Roassal을 다룬 장을 확인한다. Pharo의 Moose 배포판[2]을 이용할 경우 이미 Roassal이 설치되어 있을 것이다.

첫 시각화

첫 번째 시각화는 워크스페이스로 들어가 아래 코드를 실행하면 얻을 수 있다. 워크스페이스에 아래 코드를 실행하면 Collection 클래스 계층구조가 보일 것이다.

[1] http://themoosebook.org/book/internals/mondrian
[2] http://www.moostechnology.org/

```
| view |
view := ROMondrianViewBuilder new.
view shape rectangle
    width: [ :cls | cls numberOfVariables*5 ];
    height: #numberOfMethods;
    linearFillColor: #numberOfLinesOfCode within: Collection withAllSubclasses.

view interaction action: #browse.

view nodes: ROShape withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.
view open
```

시각화는 아래와 같이 읽힐 것이다.

- 각 클래스는 도표에 상자로 표현된다.
- 상속은 상자 사이의 edge로 표현된다. 슈퍼클래스는 그 서브클래스들 위에 위치한다.
- 각 클래스의 width(너비)는 인스턴스 변수의 양을 나타낸다.
- 클래스의 height(높이)는 클래스에 정의된 메서드의 양을 나타낸다. 클래스가 길수록 클래스가 정의한 메서드가 많음을 의미한다.
- 클래스 shading(음영)은 클래스가 포함하는 코드 행의 개수를 나타낸다. 검정색으로 칠해진 클래스는 거의 모든 코드 행을 포함한다. 흰색 클래스는 소량의 코드 행을 포함한다.

시각화에 수반되는 메커니즘은 모두 후에 상세히 살펴볼 것이다.

Mondrian 시작하기

ROMondrianViewBuilder는 Mondrian 도메인 특정 언어(DSL)를 모델화한다[3]. ROMondrianViewBuilder는 내부적으로 ROView의 인스턴스를 포함하는데, 이는 raw view라고 불린다. 그 접근자는 raw이다. ROMondrianViewBuilder를 이용한 스크립팅은 모두 스크립트가 설정한 상호작용과 모양으로 된 ROElements를 생성하여 raw view로 추가되는 결과를 야기한다. 빌더를 이용해 시각화를 시작하려면 아래 코드를 이용하면 된다.

```
view := ROMondrianViewBuilder new.
view open.
```

Mondrian 빌더는 ROView의 인스턴스를 이용해 초기화할 수도 있다. 하지만 빌더는 기본적으로 고유의 raw view를 생성할 것이기 때문에 이는 바람직하지 않음을 이해하는 것이 중요하겠다. 빌더로 작업 시에는 Mondrian DSL을 이용해 ROMondrianViewBuilder의 인스턴스로 메시지를 전송하거나 raw view를 직접 작업하는 것이 가능하다.

```
rawView := ROView new.
view := ROMondrianViewBuilder view: rawView.
view open.
```

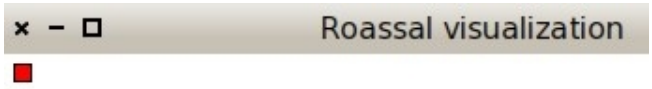
[3] http://www.moosetechnology.org/tools/mondrian

추후 내부적으로 ROElement로서 전환(translate)되는 노드를 시각화에 추가하기 위해서는 나타내고자 하는 객체와 함께 node: 선택자를 이용한다. 기본적으로 각 요소마다 작은 사각형이 그려진다.

```
view := ROMondrianViewBuilder new.
view node: 1.
view open.
```

모양을 정의하기 위해선 노드(들) 정의 앞에 shape 메시지를 이용한 후 원하는 모양과 특성을 이용한다. 이는 노드에 대한 모양을 국부적으로 정의할 것이다.

```
view := ROMondrianViewBuilder new.
view shape rectangle
    size: 10;
    color: Color red.
view node: 1.
view open.
```



객체의 컬렉션과 함께 nodes: 메시지를 이용하면 여러 개의 노드를 생성할 수 있다.

```
view := ROMondrianViewBuilder new.
view shape rectangle
    size: 10;
    color: Color red.
view nodes: (1 to: 5).
view open.
```

노드(들이)가 중첩된 노드들을 가진 경우 그들을 추가하기 위해선 node:forIt: 이나 nodes:forEach: 메시지를 사용하라. 두 번째 매개변수는 블록으로서, 아래 코드와 같이 중첩된 노드들을 추가할 것이다.

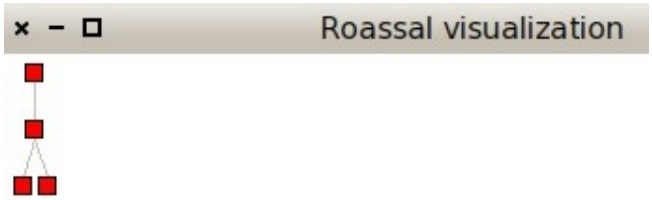
```
vview := ROMondrianViewBuilder new.
view shape rectangle
    size: 10;
    color: Color red.
view
    nodes: (1 to: 5)
    forEach: [ :each |
        view shape rectangle
            size: 5;
            color: Color yellow.
        view nodes: (1 to: 2) ].
view open.
```

노드의 모델 간 연관의 컬렉션과 함께 edgesFromAssociations: 메시지를 이용하면 edges를 생성할 수 있다.

```
view := ROMondrianViewBuilder new.
view shape rectangle
    color: Color red.
view nodes: (1 to: 4).
view
    edgesFromAssociations: (Array with: 1-> 2 with: 2 -> 3 with: 2 -> 4).
view open.
```

본 장이 시작될 때 소개한 Collection 계층구조의 예제와 마찬가지로 적절한 레이아웃이 필요하다. 기본적으로 빌더는 가로 직선 레이아웃을 적용하며, 우리에겐 트리 레이아웃이 필요하다. 따라서 treeLayout을 이용해 적용한다.

```
vview := ROMondrianViewBuilder new.
view shape rectangle
    size: 10;
    color: Color red.
view nodes: (1 to: 4).
view edgesFromAssociations: (Array with: 1-> 2 with: 2 -> 3 with: 2 -> 4).
view treeLayout.
view open.
```



Collection 계층구조 예제

Mondrian DSL은 Collection 계층구조 시각화에 대해 이번 장에 걸쳐 구성된 스크립팅보다 단순한 스크립팅을 허용한다. 각 요소와 edge를 어떻게 생성할 것인지 설정하면 수동으로 생성할 필요가 없어진다. 앞서 소개한 버전을 아래 코드로 대체할 수 있겠다.

```
view := ROMondrianViewBuilder new.
view shape rectangle
    width: [ :cls | cls instVarNames size ];
    height: [ :cls | cls methods size ].
view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.
view open.
```

Mondrian을 작업하는 방법에는 두 가지가 있는데, easel을 이용하는 방법과 view renderer를 이용하는 방법이다. easel은 사용자가 스크립트를 이용해 시각화를 상호작용적으로, 그리고 점진적으로 빌드할 수 있는 툴이다. easel은 특히 prototyping에 유용하다. MOViewRenderer는 시각화를 비상호작용적인 방식으로 계획에 따라 빌드될 수 있도록 해준다. 자신의 애플리케이션에 시각화를 내장할 경우 해당 클래스를 사용하길 원할 것이다.

우리는 먼저 가장 간단한 방식, 즉 easel을 이용해 Mondrian을 사용해볼 것이다. easel을 열기 위해선 World 메뉴("Mondrian Easel" 엔트리를 포함할 것이다)를 이용하거나 아래 표현식을 실행한다.

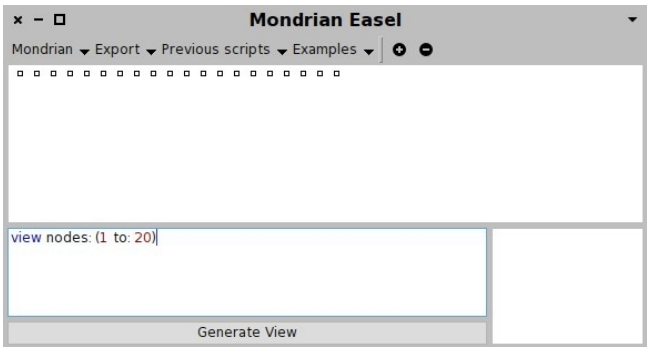
```
ROEaselMorphic open.
```

방금 열린 easel에서 두 개의 패널을 볼 수 있을 것인데, 상단의 패널은 시각화 패널이고, 나머지는 스크립트 패널이다. 스크립트 패널에서 아래 코드를 입력하고 generate 버튼을 누른다.

```
view nodes: (1 to: 20).
```

패인에 20개의 작은 상자가 상단 좌측 모서리부터 줄을 지어 있음을 볼 수 있다. 당신은 방금 1부터 20까지 숫자 집합을 렌더링한 것이다. 각 상자는 하나의 숫자를 나타낸다. 당신이 실행할 수 있는 상호작용의 양은 현재 꽤 제한된다. 한 숫자를 드래그 앤 드롭하면 그 값을 나타내는 툴팁(tooltip)을 얻을 것이다. 상호작용을 정의하는 방법은 곧 살펴볼 것이다. 당장은 Mondrian의 기본 그리기 기능을 살펴보자.

이전에 그린 노드들 사이에 edges를 추가할 수 있겠다. 두 번째 행을 추가하라.




```
view nodes: (1 to: 20).
view edgesFrom: [:v | v\(\ast{}\)2].
```


각 숫자는 그 double과 연결된다. 모든 double이 눈에 보이는 것은 아니다. 가령 20의 double은 40인데, 이는 시각화에 해당하지 않기 때문에 이런 경우 edge가 그려지지 않는다.

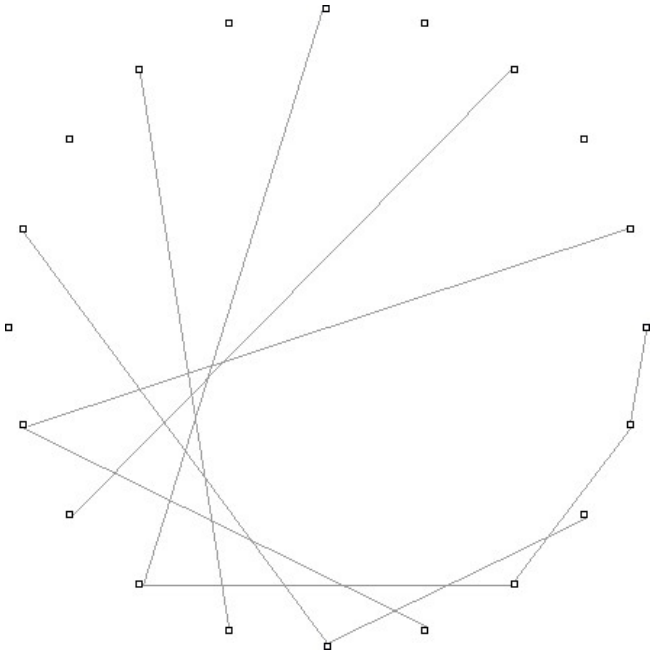
edgesFrom: 메시지는 edge의 정의가 가능한 경우 각 노드 당 하나의 edge를 정의한다. 시각화에 각 노드가 추가되면 edge는 해당 노드와 제공된 블록에서 검색된 노드 사이로 정의된다.

Mondrian은 노드를 정렬하기 위한 다수의 레이아웃을 포함한다. 여기선 원형 레이아웃을 사용해보도록 하겠다.

```
view nodes: (1 to: 20).
view edgesFrom: [:v | v\(\ast{}\)2].
view circleLayout.
```

실행하면 다음같은 결과를 얻을 수 있다:

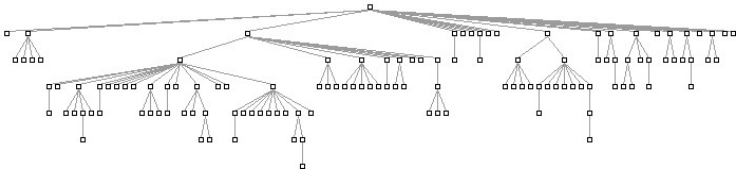
다음 절에서는 소프트웨어 코드를 시각화할 것이다. 소스 코드의 시각화는 보통 패턴을 발견하는 데에 사용되며, 코드의 질을 평가 시 유용하다.



컬렉션 프레임워크 시각화하기

이제 Pharo 클래스를 시각화할 것이다. 이 시점부터는 Pharo의 반영적 기능을 집중적으로 이용하여 Collection 클래스 계층구조를 들여다볼 것인데, 이는 설득력 강한 예제가 된다. Collection 프레임워크에 포함된 클래스의 계층구조를 시각화 해보자.

```
view nodes: Collection withAllSubclasses.
view edgesFrom: \#superclass.
view treeLayout.
```



트리 레이아웃을 이용해 클래스 계층구조를 시각화하였다. 스몰토크는 단일 상속 지향적이기 때문에 이 레이아웃이 적절하다. Collection은 Pharo 컬렉션 프레임워크 라이브러리의 루트 클래스가 아니다. withAllSubclasses 메시지는 Collection과 그 서브클래스의 리스트를 리턴한다.

클래스는 상속 링크를 따라 수직으로 정렬된다. 슈퍼클래스는 그 서브클래스의 위에 위치한다.

노드 모양 변경하기

Mondrian은 객체에 대한 그래프를 시각화한다. 도메인의 각 객체는 그래프 요소, 노드, 또는 edge에 연관되어 있다. 그래프 요소는 그들의 그래프 표현에 대해 알지 못한다. 그래프 면은 shape에 의해 주어진다.

지금까지는 노드와 edge를 표현하기 위해 기본 모양만 사용해왔다. 노드의 기본 모양은 5 픽셀 너비의 사각형이며, edge의 기본 모양은 회색의 얇은 직선이다.

수많은 치수(dimension)가 shape의 외관을 정의하는데, 직사각형의 너비와 높이, 선(line dash)의 크기, 테두리와 안쪽 색상을 예로 들 수 있겠다. 우리는 시각화하는 클래스의 내적 구조에 관해 더 많은 정보를 제공하기 위해 시각화의 노드 모양을 변경할 것이다. 아래를 고려해 보자.

```
view shape rectangle
width: [:each | each instVarNames size\(\ast{}\)3];
height: \#numberOfMethods.
view nodes: Collection withAllSubclasses.
view edgesFrom: \#superclass.
view treeLayout.
```

이 결과는 그림 12.1에 실려 있으며, 각 클래스는 상자로 표현된다. Collection 클래스, 즉 계층구조의 루트는 가장 위에 위치한 상자다. 클래스의 너비는 클래스가 가진 인스턴스 변수의 양을 알려준다. 해당 수치에 3을 곱하면 좀 더 대조적인 결과가 야기된다. 높이는 메서드의 개수를 알려준다. 다른 클래스에 비해 메서드가 많은 클래스들, Collection, SequentiableCollection, String, CompiledMethod를 즉시 알아챌 수 있을 것이다. 다른 클래스에 비해 변수의 개수가 많은 클래스로 RunArray, SparseLargeTable을 들 수 있다.

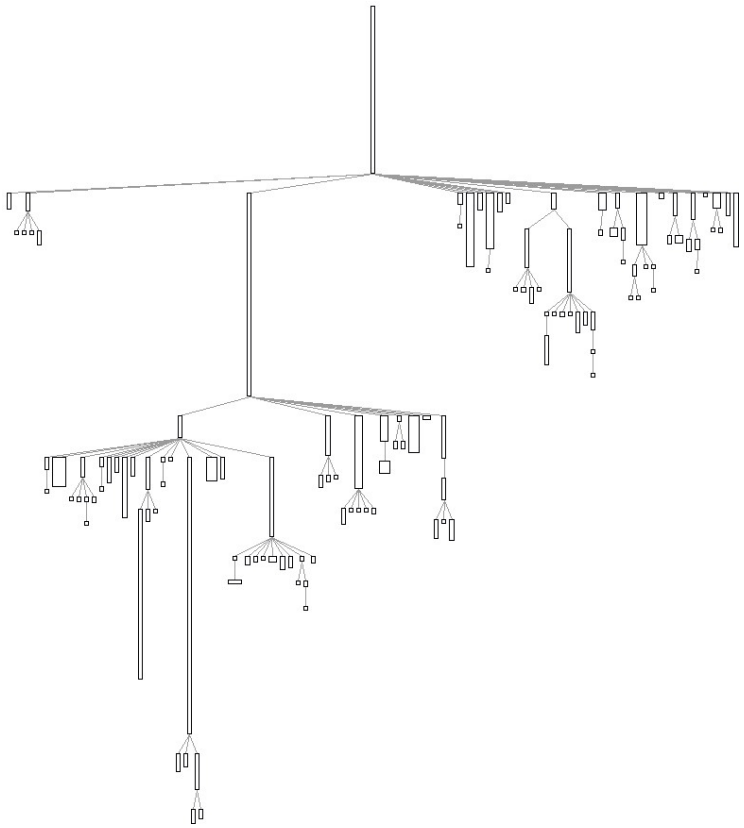


그림 12.1: Collection 클래스 계층구조의 시스템 복잡성.

다수의 edge

edgesFrom: 메시지는 기껏해야 하나의 노드 당 하나의 edge만 그리는 데 사용된다. 그 변형체로 edges:from:toAll: 이란 메시지가 있는데, 이는 주어진 노드에서 시작해 여러 개의 edge 정의를 지원한다. 클래스들 간 의존성을 고려해보자.

```
view shape rectangle
    size: [:cls | cls referencedClasses size ];
    withText.
view nodes: ArrayedCollection withAllSubclasses.
view shape arrowedLine.
view
    edges: ArrayedCollection withAllSubclasses from: #yourself toAll: #referencedClasses.
view circleLayout.
```

위의 스크립트에서 얻은 시각화를 그림 12.2에 제시하겠다.

String과 CompiledMethod가 눈에 띈다. 두 클래스는 다른 클래스에 대한 다수의 참조를 포함한다. text: 는 shape가 텍스트를 포함하도록 만드는 것도 볼 수 있다.

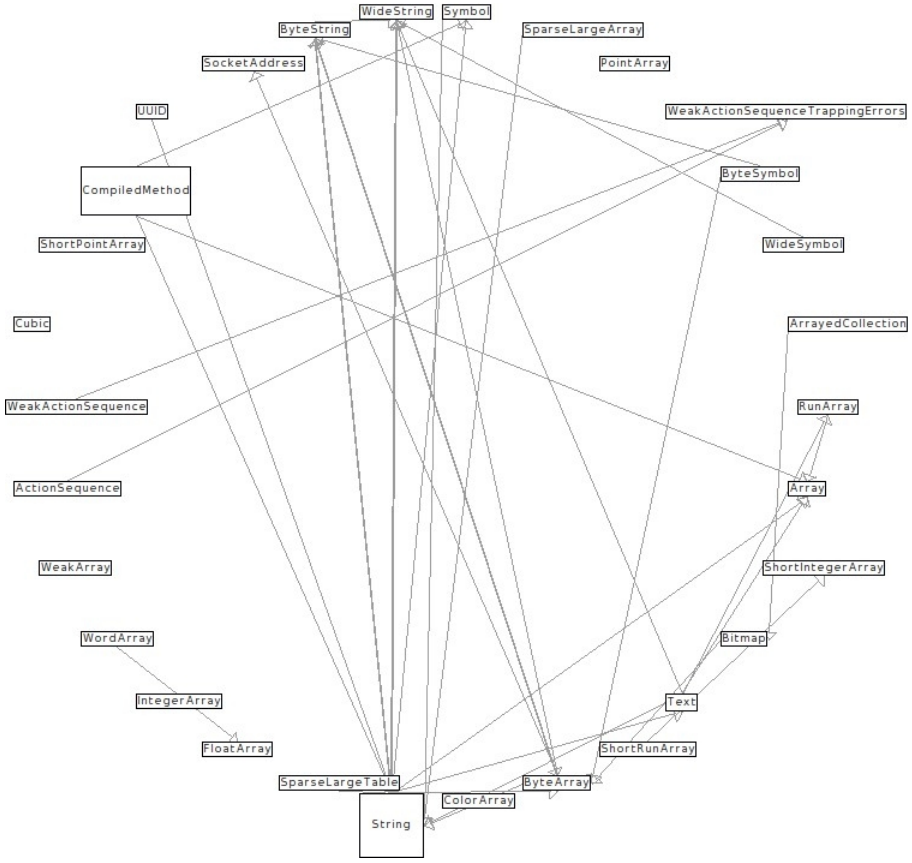


그림 12.2: 클래스 간 직접 참조.

Mondrian은 요소를 쉽게 생성하도록 수많은 유틸리티 메서드를 제공한다. 아래 표현식을 고려해보자.

```
view edgesFrom: \#superclass
```

edgesFrom: 는 edges:from:to: 와 같다.

```
view edges: Collection withAllSubclasses from: \#superclass to: \#yourself.
```

그리고 그 자체는 아래와 동일하다.

```
view
    edges: Collection withAllSubclasses
    from: [ :each | each superclass ]
    to: [ :each | each yourself ].
```

색이 칠해진 모양

모양은 여러 다양한 방식으로 색상을 입힐 수 있다. 노드 모양은 fillColor:, textColor:, borderColorder: 메시지를 이해한다. 직선 모양은 color: 을 이해한다. 이제 Collection 계층구조의 시각화에 색을 칠해보자.

```
view shape rectangle
    size: 10;
    borderColor: [ :cls | ('*Array*' match: cls name)
        ifTrue: [ Color blue ]
        ifFalse: [ Color black ] ];
   fillColor: [ :cls | cls hasAbstractMethods ifTrue: [ Color lightGray ] ifFalse: [ Color white]
        ].
view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.
```

생성된 시각화는 그림 12.3에 주어진다. 이는 "Array"로 명명되지 않은 추상적 클래스들과, 추상적 메서드가 없는 추상적인 클래스를 식별하도록 도와준다.

height:와 width: 와 유사하게 색상을 정의하는 메시지들도 기호, 블록, 또는 상수 값을 인자로서 취한다. 인자는 그래픽 요소가 표현하는 도메인 객체에 대해 평가된다 (이중 디스패치가 moValue: 메시지를 인자로 전송한다). ifTrue:ifFalse: 는 그다지 실용적이지 못하다. 특정 조건에서 색상을 쉽게 선택하기 위한 용도로 유틸리티 메서드들이 제공된다. 모양은 아래와 같이 간단하게 정의할 수 있다.

```
view shape rectangle
size: 10;
```

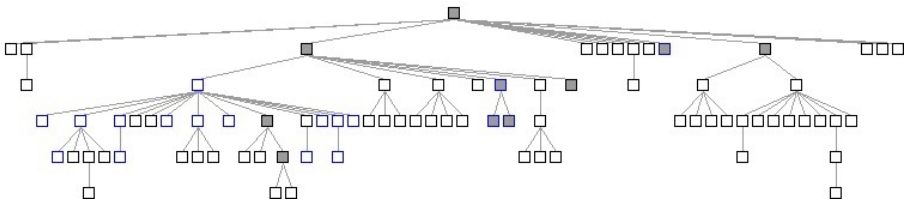


그림 12.3: 추상 클래스는 회색으로, 이름이 "Abstract"라는 단어로 된 클래스는 파란색으로 표시된다.

```
    if: [ :cls | ('*Array*' match: cls name) ] borderColor: Color blue;
    if: [ :cls | cls hasAbstractMethods ] fillColor: Color lightGray;
...
```

hasAbstractMethods 메서드는 Pharo에서 Behavior 와 Metaclass 에서 정의된다. 클래스로 hasAbstractMethods를 전송하면 부울 값을 리턴하면서 클래스가 추상적인지 여부를 알려준다. 스몰토크에서 추상적 클래스란 최소한 하나의 추상적 메서드를 (예: self subclassResponsibility를 포함하는) 정의하거나 상속하는 클래스임을 기억할 것이다.

색상에 관한 추가 정보

색상은 프로퍼티를 지정하는 데에 꽤 유용하다 (예: 클래스가 추상적일 경우 회색). 이는 연속적인 분포를 표현할 때 사용되기도 한다. 예를 들어, 채도(color intensity)는 측정(metric)의 결과를 나타낼 수 있다. 앞에서 노드 채도가 코드 행 수에 관해 알려주는 스크립트를 생각해보자.

```
view interaction action: #browse.
view shape rectangle
    width: [ :each | each instVarNames size*3 ];
    height: [ :each | each methods size ];
    linearFillColor: #numberOfLinesOfCode within: Collection withAllSubclasses.
view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.
```

그림 12.4는 결과 그림을 보여준다. linearFillColor:within: 메시지는 첫 번째 인자로서 숫자형 값을 리턴하는 블록 함수를 취한다. 두 번째 인자는 각 노드의 명암도(intensity)를 변경하는 데에 사용되는 요소의 그룹이다. 블록 함수는 그룹의 각 요소에 적용된다. 음영은 흰색에서 (코드의 0행) 검정색으로 (코드의 최대 숫자 행) 측정된다. 최대 명암도는 Collection의 모든 서브클래스에 대한 최대 #numberOfLInesOfCode에 의해 주어진다. linearFillColor:within:의 변형체로 linearXXXFillColor:within: 이 있는데, 여기서 XXX는 Blue, Green, Red, Yellow 중에 하나가 해당한다.

여기서 얻는 시각화[4]는 메서드의 개수, 인스턴스 변수의 개수, 코드의 행 수에 대한 각 클래스의 관계를 넣는다. 클래스들 간 크기 차이는 관리 활동이 필요함을 제시할지도 모른다.

[4] 시각화는 '시스템 복잡성'으로 명명되는데 이에 관해 더 알고 싶다면 'Polymetric Views — A Lightweight Visual Approach to Reverse Engineering'(Transactions on Software Engineering, 2003)을 참고한다.

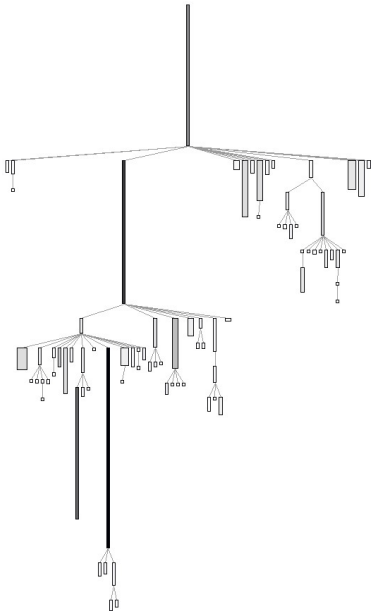


그림 12.4: 시스템 복잡성 시각화: 노드는 클래스이고, 높이는 메서드의 행 수, 너비는 변수의 개수, 색상은 코드 행의 개수를 말한다.

색상은 identifyFillColorOf: 를 이용해 객체 정체성으로 할당될 수 있다. 인자는 블록이거나 기호로서, 도메인 객체에 대해 평가된다. 색상은 인자의 결과에 연관된다.

팝업 뷰

추상 클래스 예제로 돌아가보자. 아래 스크립트는 추상 클래스, 그리고 그들이 정의하는 추상 메서드 개수를 나타낸다.

```
view shape rectangle
    size: [ :cls | (cls methods select: #isAbstract ) size*5 ] ;
    if: #hasAbstractMethods fillColor: Color lightRed;
    if: [:cls | cls methods anySatisfy: #isAbstract ] fillColor: Color red.
view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.
```

그림 12.5는 추상 클래스의 정의 또는 상속에 따라 추상적인 클래스를 나타낸다. 클래스 크기는 정의된 추상 메서드의 개수를 나타낸다.

팝업 메시지는 추상 메서드를 열거하도록 개선할 수 있다. 클래스 위에 마우스를 놓으면 그 이름 뿐만 아니라 클래스 내의 추상 메서드 리스트까지 제공한다. 아래 코드 조각을 앞에 추가해야 한다.

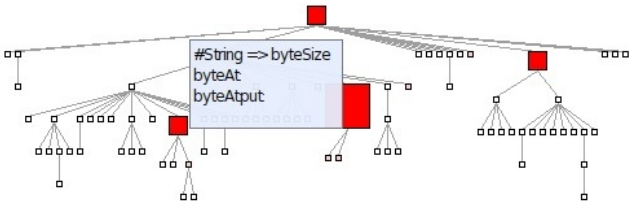


그림 12.5: 상자는 클래스, 링크는 상속 관계다. 추상 메서드의 양은 클래스 크기에 따라 표현된다. 빨간색 클래스는 추상 메서드를 정의하고, 분홍색 클래스는 추상 클래스에서만 상속된다.

```
view interaction popupText: [ :aClass |
    | stream |
    stream := WriteStream on: String new.
    (aClass methods select: #isAbstract thenCollect: #selector)
        do: [:sel | stream nextPutAll: sel; nextPut: $ ; cr].
    aClass name printString, ' => ', stream contents ].
...
```

지금까지 그래픽 표현을 설명하기 위해 모양을 가진 요소를 살펴보았다. 요소는 이벤트 핸들러를 포함하는 상호작용도 포함한다. popupText: 메시지는 블록을 인자로 취한다. 블록은 도메인 객체를 인자로 하여 평가된다. 블록은 팝업 텍스트 내용을 리턴해야 하는데, 본문의 예제에서는 단순히 메서드의 리스트에 해당한다.

텍스트 내용에 더해 Mondrian은 뷰의 팝업을 허용하기도 한다. 이 요점을 설명하기 위해 앞의 예제를 확장해보겠다. 마우스가 노드로 들어가면 새 뷰가 정의되고 노드 옆에 표시된다.

```
view interaction popupView: [ :element :secondView |
    secondView node: element forIt: [
        secondView shape rectangle
            if: #isAbstract fillColor: Color red;
            size: 10.
        secondView nodes: (element methods sortedAs: #isAbstract).
        secondView gridLayout gapSize: 2
    ]].

view shape rectangle
    size: [ :cls | (cls methods select: #isAbstract ) size*5 ] ;
    if: #hasAbstractMethods fillColor: Color lightRed;
    if: [:cls | cls methods anySatisfy: #isAbstract ] fillColor: Color red.
view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.
```

popupView: 의 인자는 2 인자 블록이다. 블록의 첫 번째 매개변수는 마우스 아래 위치한 노드가 표현하는 요소다. 두 번째 인자는 열리게 될 새 뷰가 된다.

예제에서 우리는 메서드를 표현하는 노드를 정렬하기 위해 sortedAs: 를 이용했다. 해당 메서드는 Collection에서 정의되고 Mondrian에 속한다. sortedAs: 의 사용 예제를 보고 싶다면 해당하는 단위 시험(unit test)을 훑어보도록 한다.

마지막 예제에서는 하위뷰를 정의하기 위해 팝업 뷰에서 node:forIt: 을 사용한다.

하위뷰

노드는 그 자체 내의 뷰이다. 이는 그래프가 어떤 노드로든 내장될 수 있도록 해준다. 내장된 뷰는 캡슐화하는 노드에 의해 바인딩된다. 내장은 nodes:forEach: 와 node:forIt: 키워드를 통해 실현된다.

아래 예제는 서로 호출이 가능한 메서드를 연결함으로써 메서드들 간 의존성을 계산한다. 메서드 m1이 선택자 #m2로의 참조를 포함할 경우 m1은 메서드 m2로 연결된다. 이는 간단하지만 메서드들 간 의존성을 효율적으로 바라보는 방법이다. 아래를 고려해보자.

```
view nodes: ROShape withAllSubclasses forEach: [:cls |
    view nodes: cls methods.
    view edges: cls methods from: #yourself toAll: [ :cm | cls methods select: [ :rcm | cm
        messages anySatisfy: [:s | rcm selector == s ] ] ].
    view treeLayout
].
view interaction action: #browse.
view edgesFrom: #superclass.
view treeLayout.
```

하위뷰는 고유의 레이아웃을 포함한다. 하위뷰에서 정의된 상호작용과 모양은 중첩(nesting) 노드에서 접근할 수 없다 (그림 12.6).

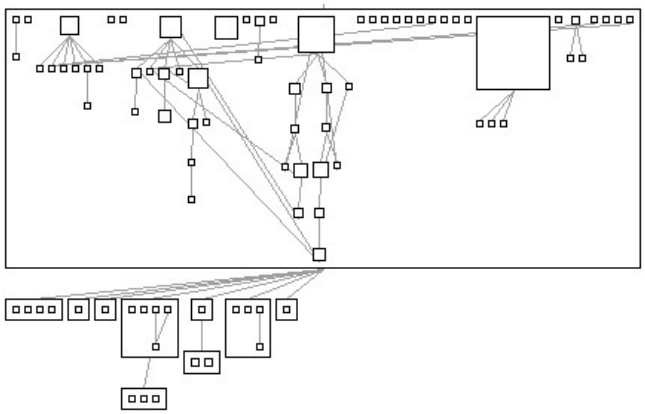


그림 12.6: 큰 상자는 클래스들이다. 그 내부의 상자들은 메서드에 해당한다. edges는 둘 사이에 가능한 호출을 표시한다.

이벤트 전달하기

앞 절에서 주어진 시각화의 메서드들은 간단한 드래그 앤 드롭을 통해 이동할 수 있다. 하지만 메서드의 위치는 고정되어 있고 클래스만 드래그 앤 드롭이 가능하길 원하는 경우도 있는데, 이럴 때는 상호작용으로 forward 메시지가 전송되어야 한다. 아래를 살펴보자.

```
view nodes: ROShape withAllSubclasses forEach: [:cls |
    view interaction forward.
    view shape rectangle
        size: #linesOfCode.
    view nodes: cls methods.
    view edges: cls methods from: #yourself toAll: [ :cm | cls methods select: [ :rcm | cm
        messages anySatisfy: [:s | rcm selector == s ] ] ].
    view treeLayout
].
view interaction action: #browse.
view edgesFrom: #superclass.
view treeLayout.
```

메서드를 이동하면 클래스를 대신 이동시킬 것이다. 때로는 하나 이상의 요소를 드래그 앤 드롭하는 것이 편리하기도 하다. 대부분의 운영체제에서 Mondrian은 Ctrl 키나 Cmd 키를 이용해 다중 선택을 제공한다. 기본 행위는 모든 노드를 대상으로 이용할 수 있다. 다중 선택은 노드 그룹의 이동을 가능하게 한다.

이벤트

각 마우스 이동, 클릭, 키보드 키누름은 특정 이벤트에 들어맞는다. Mondrian은 풍부한 이벤트 계층구조를 제공하며, 계층구조의 루트는 MOEvent이다. 특정 액션을 이벤트로 연관시키고 싶다면 객체 상호작용에서 핸들러를 정의해야 한다. 아래 예제의 경우, 클래스를 클릭하면 코드 브라우저가 열린다.

```
view shape rectangle
    width: [ :each | each instVarNames size*5 ];
    height: [ :each | each methods size ];
    if: #hasAbstractMethods fillColor: Color lightRed;
    if: [:cls | cls methods anySatisfy: #isAbstract ] fillColor: Color red.

view interaction on: ROMouseClick do: [ :event | event model browse ].

view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.
```

블록 핸들러는 하나의 인자, 생성된 이벤트를 수락한다. 이벤트를 트리거한 객체는 이벤트 객체로 modelElement를 전송하여 얻는다.

상호작용

Mondrian은 컨텍스트의 다수의 상호작용 메커니즘을 제공한다. 상호작용 객체는 그러한 용도의 키워드를 많이 포함한다. highlightWhenOver: 메시지는 블록을 인자로서 취하는데, 해당 블록은 마우스가 노드로 진입 시 강조해야 하는 노드의 리스트를 리턴한다. 아래 예제를 살펴보자.

```
view interaction
    highlightWhenOver: [:v | {v - 1 . v + 1. v + 4 . v - 4}].
view shape rectangle
    width: 40;
    height: 30;
    withText.
view nodes: (1 to: 16).
view gridLayout gapSize: 2.
```

노드 5로 진입하면 노드 4, 6, 1, 9가 강조된다. 이러한 메커니즘은 연결 edge의 오버로딩을 피하는 데 꽤 효율적이다. 관심 노드에 대한 정보만 표시되기 때문이다.

아래 예제에서는 좀 더 강력한 highlightWhenOver: 의 적용 예제를 제시한다. 클래스의 계층구조는 좌측에 표시된다. 우측에는 unit test의 계층구조가 표시된다. 마우스 포인터를 unit test로 갖다 대면 해당 unit test 메서드 중 하나가 참조하는 클래스들을 강조한다. (길지만) 아래 스크립트를 고려해보라.

```
"System complexity of the collection classes"
view shape rectangle
    width: [ :each | each instVarNames size*5 ];
    height: [ :each | each methods size ];
    linearFillColor: #numberOfLinesOfCode within: Collection withAllSubclasses.
view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.

"Unit tests of the package CollectionsTest"
view shape rectangle withoutBorder.
view node: 'compound' forIt: [
    view shape label.
    view node: 'Collection tests'.

view node: 'Collection tests' forIt: [
    | testClasses |
    testClasses := (PackageInfo named: 'CollectionsTests') classes reject: #isTrait.
    view shape rectangle
        width: [ :cls | (cls methods inject: 0 into: [ :sumLiterals :mtd | sumLiterals + mtd
            allLiterals size]) / 100 ];
        height: [ :cls | cls numberOfLinesOfCode / 50 ].
    view interaction
        highlightWhenOver: [ :cls | ((cls methods inject: #()
            into: [:sum :el | sum , el allLiterals ]) select: [:v | v isKindOf: Association ]
        thenCollect: #value) asSet ].
    view nodes: testClasses.
    view edgesFrom: #superclass.
    view treeLayout ].

    view verticalLineLayout alignLeft
].
```

위 스크립트는 두 가지 부분을 포함한다. 첫 번째는 컬렉션 프레임워크에서 아주 흔하게 발견되는 시스템 복잡성이고, 두 번째는 CollectionsTests에 포함된 테스트를 제공한다. 클래스 너비는 클래스에 포함된 리터럴 개수를, 높이는 코드의 행 수를 나타낸다. 컬렉션 테스트는 코드의 재사용을 위한 특성을 많이 사용하기 때문에 이러한 측정(metrics)은 축소되어야 한다. 마우스를 test unit 위로 갖다 대면 해당 클래스 내에서 참조된 컬렉션 프레임워크의 모든 클래스들이 강조된다.

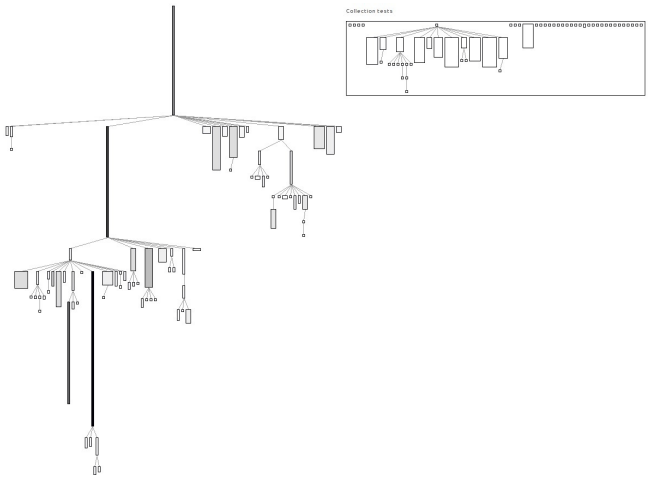


그림 12.7: 상호작용적 시스템 복잡성.

요약

Mondrian은 객체의 어떤 그래프든 시각화하도록 해준다. 본 장에서는 아래와 같은 Mondrian의 주요 특징들을 살펴보았다.

- 노드를 정의할 때는 nodes: 를, edges를 정의할 때는 edgesFrom:, edges:from:to:, edges:from:toAll: 을 이용하는 방법이 가장 일반적이다.
- 레이아웃의 전체 범위가 제공된다. 가장 일반적인 레이아웃은 circleLayout, treeLayout, gridLayout을 뷰로 전송하면 접근할 수 있다.
- Shape은 요소의 그래픽적 측면을 정의한다. 요소의 높이와 너비는 보통 각각 height: 과 width: 를 이용해 설정된다.
- 모양은 borderColor: 와 fillColor: 로 칠한다.
- 정보는 popupText: 와 popupView: 를 이용해 팝업할 수 있다.
- 하위뷰는 nodes:forEach: 와 node:forIt: 을 이용해 정의된다.
- 하위 노드의 이벤트는 forward: 와 forward: 를 이용해 그 부모 노드로 전달된다.
- 강조(highlighting)는 highlightWhenOver: 를 통해 이용 가능한데, 이는 1 인자 블록을 취하며 강조해야 할 노드의 리스트를 리턴한다.

본 장은 Mondrian 도메인 특정 언어를 집중적으로 다루었다. Mondrian은 Tudor Girba와 Michael Meyer이 2005년에 개발한 오래된 시각화 프레임워크다. Mondrian은 2008년부터 2009년까지 Alexandre Bergel에 의해 관리되었다.

감사의 말. 본 장의 초판을 검토해준 Nicolas Rosselot Urrejola에게 감사드린다.

노드 색상은 노드의 높이와 너비로서 중요한 정보 지원이다. 색상은 특정 상태를 표현하도록 선택하기 쉬워야 한다.

if:fillColor: 키워드는 특정 상태에 대해 색상을 할당하도록 해준다. 앞의 예제를 확장시켜 추상 클래스를 빨간색으로 칠한다고 가정하자.

```
view shape rectangle
    width: [ :each | each instVarNames size*3 ];
    height: [ :each | each methods size ];
    if: #hasAbstractMethods fillColor: Color red.
view nodes: Collection withAllSubclasses.

view edges: Collection withAllSubclasses from: #yourself toAll: #subclasses.

view treeLayout.
```

if:fillColor: 메시지는 조건적으로 색상을 설정하도록 shape로 전송될 수 있다.

```
view shape rectangle
    width: [ :each | each instVarNames size*3];
    height: [ :each | each methods size ];
    if: #hasAbstractMethods fillColor: Color red.
view nodes: Collection withAllSubclasses.

view edgesFrom: #superclass.

view treeLayout.
```

빨간색 노드는 모두 추상적 클래스를 표현한다. 노드 위에서 마우스(moose라고 되어 있는데 오타 같아서 mouse로 번역해드립니다만 확인 바랍니다.)를 왔다 갔다 하면 이름을 표시하는 텍스트 툴팁에 나타난다.

상호작용을 정의하기 위한 확장된 기능들도 존재한다. 이는 추후 살펴볼 것이다. 당장은 노드에서 직접 시스템 브라우저를 열고 싶다면 아래 상호작용을 정의한다.

```
view interaction action: #browse.
view shape rectangle
    width: [ :each | each instVarNames size*3];
    height: [ :each | each methods size ];
    if: #hasAbstractMethods fillColor: Color red.
view nodes: Collection withAllSubclasses.

view edgesFrom: #superclass.

view treeLayout.
```

서브클래스를 갖고 있지 않은 빨간색 노드가 몇 개 눈에 띌 것이다. 추상 클래스는 서브클래스를 가져야 하므로 이는 설계 흐름을 나타낸다. 인스턴스화 되어선 안 되는 (추상적이므로) 클래스가 서브클래스를 갖지 않는다는 것은 말이 안 된다.

자신의 클래스에서도 이와 같은 분석을 실현할 수 있다.

```
view interaction action: #browse.
view shape rectangle
    width: [ :each | each instVarNames size*3];
    height: [ :each | each methods size ];
    if: #hasAbstractMethods fillColor: Color red.
view nodes: (PackageInfo named: 'Mondrian') classes.

view edgesFrom: #superclass.

view treeLayout.
```

shape은 하나 이상의 상태를 포함할 수도 있다. 추상 메서드를 정의하는 클래스로부터 추상 클래스를 구별하자.

```
view shape rectangle
    width: [ :each | each instVarNames size*3];
    height: [ :each | each methods size ];
    if: #hasAbstractMethods fillColor: Color lightRed;
    if: [:cls | cls methods anySatisfy: #isAbstract ] fillColor: Color red.
view nodes: Collection withAllSubclasses.

view edgesFrom: #superclass.

view treeLayout.
```

모든 빨간색 노드는 여전히 추상적 클래스이다. 연한 빨간색은 추상 메서드를 정의하지 않는 클래스를 나타내고, 진한 빨간색은 최소 하나의 추상 메서드를 정의하는 클래스를 의미한다.

제 IV 부

Language

제 13 장

예외 처리하기

***Clément Bera** 참여 (bera.clement@gmail.com)*

모든 애플리케이션은 예외적인 상황을 처리해야 한다. 산술적 오류가 발생하기도 하고 (0으로 나눌 경우), 예기치 못한 상황이 발생하기도 하며 (파일을 찾을 수 없는 경우), 자원이 소진될 수도 있다 (네트워크가 다운되거나 디스크가 꽉 차는 경우 등). 예전에는 실패 연산이 특별한 오류 코드를 리턴하도록 만들어 이러한 문제를 해결했는데, 이는 클라이언트 코드가 각 연산의 리턴 값을 확인하고 오류를 처리하기 위해 특별한 액션을 취해야 함을 의미한다. 이러한 경우 불안정한 코드로 이어진다.

몇 가지 예제를 도움 삼아 이러한 가능성들을 모두 살펴보고, 예외와 예외 핸들러의 내재적 기법을 깊숙이 들여다보고자 한다.

서론

스몰토크를 포함해 현대 프로그래밍 언어들은 예외적 상황을 시그널링하고 처리하는 방식을 매우 간소화하는 예외 처리 전용의 메커니즘을 제공한다. 1996년에 ANSI 스몰토크 표준이 개발되기 이전에는 여러 개의 예외 처리 메커니즘이 존재했으며 대개 서로 호환이 되지 않았다. Pharo의 예외 처리는 ANSI 표준을 따르고, 몇 가지 embellishments를 이용하는데, 본 장에서는 사용자 관점으로부터 이를 제시하겠다.

예외 처리의 기본 개념은 클라이언트 코드가 오류 코드의 검사로 주요(main) 논리 흐름을 채우지 않고 대신 예외를 "잡아내도록" 예외 핸들러를 명시한다는 데에 있다. 무언가 잘못되면 오류 코드를 리턴하는 대신 예외적 상황을 감지한 메서드가 예외를 시그널링함으로써 주요 실행 흐름을 방해하는 것이다. 이는 두 가지 일을 해내는데, 예외가 발생하는 컨텍스트에 관한 필수 정보를 포착하고, 클라이언트가 작성한 예외 핸들러로 제어를 전달하여 어떻게 할 것인지 결정하도록 한다. "컨텍스트에 관한 필수 정보"는 Exception 객체에 저장되고, 발생할 수 있는 다양한 예외적 상황을 다루도록 Exception의 다양한 클래스가 명시된다.

Pharo의 예외 처리 메커니즘은 특별히 표현적이고 유연하여 광범위한 가능성을 가진다. 예외 핸들러는 무언가 잘못되더라도 특정 액션이 일어나도록 하거나 무언가 잘못되었을 때만 액션을 취하도록 보장하는 데에 사용할 수 있다. 스몰토크에서 여느 것과 마찬가지로 예외 또한 객체이며 다양한 메시지에 반응한다. 예외가 핸들러에 의해 포착되면 다수의 응답이 가능하다. 핸들러가 실행해야 할 대안적 액션을 명시할 수도 있고, 방해된 연산을 재개해줄 것을 예외 객체로 요청할 수도 있고, 연산을 재시도할 수 도 있으며, 다른 핸들러로 예외를 전달하기도 하고, 혹은 완전히 다른 예제를 재발생 시킬 수도 있다.

실행 보장하기

ensure: 메시지를 블록으로 전송하면 블록이 실패하더라도 (예 : 예외를 발생시킨다) 인자 블록이 실행되도록 확보할 수 있다.

```
anyBlock ensure: ensuredBlock "ensuredBlock will run even if anyBlock fails"
```

사용자가 찍은 스크린샷에서 이미지 파일을 생성하는 아래 예제를 고려해보자.

```
| writer |
writer := GIFReadWriter on: (FileStream newFileNamed: 'Pharo.gif').
[ writer nextPutImage: (Form fromUser) ]
    ensure: [ writer close ]
```

위의 코드는 파일로 작성 중이거나 Form fromUser에 오류가 발생하더라도 writer 파일 핸들이 닫히도록 보장한다.

좀 더 상세히 설명하자면 이렇다. GIFReadWriter 클래스의 nextPutImage: 메서드는 폼 (예 : 비트맵 이미지를 나타내는 Form 클래스의 인스턴스)을 GIF 이미지로 변환한다. 해당 메서드는 파일에서 열린 스트림으로 작성한다. nextPutImage: 메시지는 그것이 작성하는 스트림을 닫지 않으므로 작성하는 동안 문제가 발생하더라도 스트림이 닫히도록 보장해야 한다. 이는 작성을 수행하는 블록으로 ensure: 메시지를 전송하면 가능하다. nextPutImage: 가 실패할 경우 제어는 ensure: 로 전달된 블록으로 흐른다. 따라서 어떤 경우든 우리는 writer가 확실히 닫히도록 확보할 수 있다.

Cursor: 클래스를 살펴보면 ensure: 의 또 다른 사용 예를 볼 수 있다.

```
Cursor>>showWhile: aBlock
    "While evaluating the argument, aBlock,
    make the receiver be the cursor shape."
    | oldcursor |
    oldcursor := Sensor currentCursor.
    self show.
    ^aBlock ensure: [ oldcursor show ]
```

aBlock이 예외를 시그널링하든 말든 인자 [oldcursor show]는 평가된다. ensure: 의 결과는 인자의 값이 아니라 수신자의 값임을 주목하라.

```
[1] ensure: [0] → 1 "not 0"
```

non-local returns 처리하기

ifCurtailed: 메시지는 주로 "정리(cleaning)" 액션에 사용된다. 이는 ensure: 와 비슷하지만, 수신자가 이례적으로 종료되더라도 인자 블록이 평가되도록 보장하는 ensure: 와 달리 ifCurtailed: 는 수신자가 실패하거나 리턴 할 때에만 인자 블록이 평가되도록 보장한다.

아래 예제에서 ifCurtailed: 의 수신자는 early return을 실행하기 때문에 다음 문은 절대 도달할 수 없다. 스몰토크에서는 이를 non-local return이라 부른다. 그럼에도 불구하고 인자 블록은 실행될 것이다.

```
[\textasciicircum{} 10] ifCurtailed: [Transcript show: 'We see this'].
Transcript show: 'But not this'.
```

아래 예제에서는 수신자가 이례적으로 종료될 때에만 ifCurtailed: 에 대한 인자가 평가됨을 분명히 확인할 수 있다.

```
[Error signal] ifCurtailed: [Transcript show: 'Abandoned'; cr].
Transcript show: 'Proceeded'; cr.
```

 transcript 를 열고 워크스페이스에 위의 코드를 평가하라. 디버거 창이 열리면 먼저 Proceed 를 선택하고 Abandon 을 선택하라. 수신자가 이례적으로 종료될 때에만 ifCurtailed: 로 인자가 평가됨을 명심하라. Debug 를 선택하면 어떤 일이 발생하는가?

ifCurtailed: 이 사용되는 예를 들 것인데, Transcript show: 의 텍스트는 상황을 설명한다.

```
[\textasciicircum{} 10] ifCurtailed: [Transcript show: 'This is displayed'; cr]

[10] ifCurtailed: [Transcript show: 'This is not displayed'; cr]

[1 / 0] ifCurtailed: [Transcript show: 'This is displayed after selecting Abandon in the debugger'; cr]
```

Pharo에서 ifCurtailed: 와 ensure: 는 마커 프리미티브를 이용해 구현되지만 원칙적으로 ifCurtailed: 는 다음과 같이 ensure: 를 이용해 구현할 수도 있다.

```
ifCurtailed: curtailBlock
    | result curtailed |
    curtailed := true.
    [ result := self value.
        curtailed := false ] ensure: [ curtailed ifTrue: [ curtailBlock value ] ].
    ^ result
```

이와 비슷하게 ensure: 는 다음과 같이 ifCurtailed: 를 이용해 구현할 수도 있다.

```
ensure: ensureBlock
    | result |
    result := self ifCurtailed: ensureBlock.
    "If we reach this point, then the receiver has not been curtailed,
    so ensureBlock still needs to be evaluated"
    ensureBlock value.
    ^ result
```

ensure: 과 ifCurtailed: 모두 그 중요한 "cleanup" 코드가 실행되도록 확보하는 데 매우 유용하지만 그 둘 만으로 모든 예외적 상황을 처리하기엔 역부족이다. 이제 예외 처리를 위한 좀 더 일반적인 메커니즘을 살펴보자.

예외 핸들러

일반적인 메커니즘은 on:do: 메시지에 의해 제공되는데, 그 모습은 아래와 같다.

```
aBlock on: exceptionClass do: handlerAction
```

aBlock은 이상한 상황을 감지하고 예외를 시그널링하는 코드로서, 보호 블록(protected block)이라 불린다. handlerAction은 예외가 시그널링될 경우 평가되는 블록이며, 예외 핸들러라고 불린다. exceptionClass는 예외의 클래스를 정의하고, handlerAction는 예외를 처리하도록 요청 받을 것이다.

on:do: 메시지는 수신자의 (보호 블록) 값을 리턴하고 오류가 발생 시 아래 표현식과 같이 handlerAction 블록의 값을 리턴한다.

```
[1+2] on: ZeroDivide do: [:exception | 33]
→ 3

[1/0] on: ZeroDivide do: [:exception | 33]
→ 33

[1+2. 1+ 'kjhjkhjk'] on: ZeroDivide do: [:exception | 33]
→ raise another Error
```

이 메커니즘의 미는 어떤 가능한 오류와도 상관 없이 보호 블록을 손쉬운 방식으로 작성할 수 있다는 데에 있다. 하나의 예외 핸들러는 잘못될 가능성이 있는 상황을 처리할 책임을 지닌다.

아래 예제에서 우리는 한 파일 상의 내용을 다른 파일로 복사하길 원한다. 파일과 관련해 여러 가지 일이 잘못될 수 있지만 예외 처리를 이용하면 간단한 행의 메서드를 작성하여 전체 transaction에 대해 하나의 예외 핸들러를 정의할 수 있다.

```
| source destination fromStream toStream |
source := 'log.txt'.
destination := 'log-backup.txt'.
[ fromStream := FileStream oldFileNamed: (FileSystem workingDirectory / source).
    [ toStream := FileStream newFileNamed: (FileSystem workingDirectory / destination).
        [ toStream nextPutAll: fromStream contents ]
            ensure: [ toStream close ] ]
        ensure: [ fromStream close ] ]
    on: FileStreamException
    do: [ :ex | UIManager default inform: 'Copy failed -- ', ex description ].
```

FileStreams와 관련해 예외가 발생하면 예외 객체를 그 인자로 가진 핸들러 블록(do: 다음에 오는 블록)이 실행된다. 핸들러 코드가 사용자에게 복사가 실패했음을 알리고, 오류에 관한 세부 내용을 제공하는 업무를 예외 객체 ex에게 위임한다. 두 번의 중첩된 ensure: 의 사용은 예외의 발생 여부와 상관없이 두 개의 파일 스트림이 닫히도록 보장한다.

on:do: 메시지의 수신자인 블록이 예외 핸들러의 범위를 정의한다는 사실을 이해하는 것이 중요하다. 해당 핸들러는 수신자가 (예: 보호 블록) 완료되지 않았을 경우에만 사용될 것이다.

우선 완료되면 예외 핸들러는 사용되지 않을 것이다. 뿐만 아니라 핸들러는 on:do: 에 대한 첫 번째 인자로서 명시된 예외 유형과 독자적으로 연관된다. 따라서 앞의 예제에서는 FileStreamException(또는 좀 더 구체적인 변형체)만 처리될 수 있다.

버그 해결. 아래 코드를 연구하고 무엇이 잘못되었는지 확인하라.

```
| source destination fromStream toStream |
source := 'log.txt'.
destination := 'log-backup.txt'.
    [ fromStream := FileStream oldFileNamed: (FileSystem workingDirectory / source).
    toStream := FileStream newFileNamed: (FileSystem workingDirectory / destination).
    toStream nextPutAll: fromStream contents ]
        on: FileStreamException
        do: [ :ex | UIManager default inform: 'Copy failed -- ', ex description ].
    fromStream ifNotNil: [fromStream close].
    toStream ifNotNil: [toStream close].
```

FileStreamException 를 제외한 예외가 발생할 경우 파일이 적절하게 닫히지 않는다.

오류 코드 — 제발 삼가하라!

예외를 사용하는 방법 외에 예기치 않은 결과를 생성하지 못하는 메서드를 처리하는 방법 중 바람직하지 못한 방법을 들자면, 가능한 리턴 값으로서 명시적 오류 코드를 소개하는 것을 들 수 있겠다. 사실상 C와 같은 언어에서 코드는 그러한 오류 코드에 대한 검사(check)로 인해 지저분해지고, 메인 애플리케이션 로직을 모호하게 만들기도 한다. 오류 코드는 그 진화에도 불구하고 취약하다고 볼 수 있는데, 새 오류 코드가 추가되면 모든 클라이언트는 새 코드를 고려하도록 적응해야 하기 때문이다. 오류 코드 대신 예외를 사용할 경우 프로그래머는 각 리턴 값을 명시적으로 검사하는 업무로부터 벗어나고, 프로그램 로직은 깔끔한 모습 그대로 유지된다. 뿐만 아니라 예외는 클래스에 해당하기 때문에 새로운 예외적 상황이 발견되면 서브클래싱될 수도 있다. 따라서 기존 클라이언트는 새 클라이언트보다 덜 특정적인 예외를 제공하긴 하지만 여전히 작동할 것이다.

스몰토크는 예외 처리 지원을 제공하지 않았기 때문에 앞 절에서 살펴본 작은 예제는 오류 코드를 이용해 아래와 같이 작성할 수 있겠다.

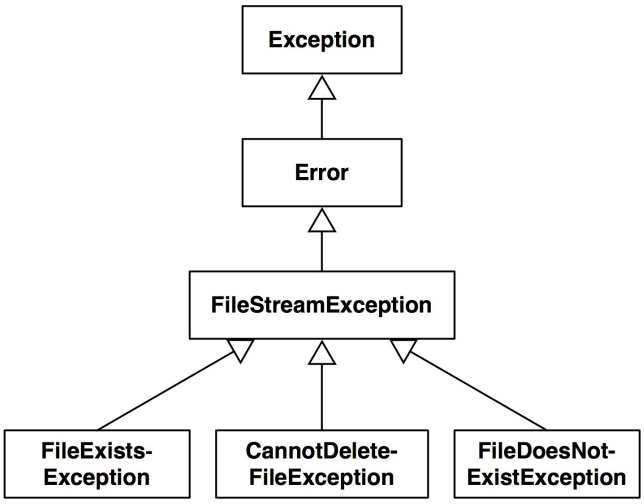


그림 13.1: Pharo 예외 계층구조의 일부분.

```
"Pseudo-code -- luckily Smalltalk does not work like this. Without the
benefit of exception handling we must check error codes for each operation."
source := 'log.txt'.
destination := 'log-backup.txt'.
success := 1. "define two constants, our error codes"
failure := 0.
fromStream := FileStream oldFileNamed: (FileSystem workingDirectory / source).
fromStream ifNil: [
    UIManager default inform: 'Copy failed -- could not open', source.
    ^ failure "terminate this block with error code" ].
toStream := FileStream newFileNamed: (FileSystem workingDirectory / destination).
toStream ifNil: [
    fromStream close.
    UIManager default inform: 'Copy failed -- could not open', destination.
    ^ failure ].
contents := fromStream contents.
contents ifNil: [
    fromStream close.
    toStream close.
    UIManager default inform: 'Copy failed -- source file has no contents'.
    ^ failure ].
result := toStream nextPutAll: contents.
result ifFalse: [
    fromStream close.
    toStream close.
    UIManager default inform: 'Copy failed -- could not write to ', destination.
    ^ failure ].
fromStream close.
toStream close.
^ success.
```

엉망진창이다! 예외 처리가 없이는 다음 연산으로 넘어가기 전에 각 연산의 결과를 명시적으로 확인해야 한다. 무언가 잘못되는 지점마다 오류 코드를 확인해야 할 뿐만 아니라 그 시점까지 실행된 모든 연산을 정리(cleanup)하고 나머지 코드 부분을 취소할 준비도 되어 있어야

한다.

어떤 예외를 처리할 것인지 명시하기

스몰토크에서는 예외도 물론 객체다. Pharo에서 예외는 예외 클래스의 인스턴스로서, 예외 클래스의 계층구조에 해당한다. 예를 들어, FileDoesNotExistException, FileExistsException, CannotDeleteFileException 예외들은 FileStreamException 의 특별한 유형으로, 그림 13.1에서와 같이 FileStreamException의 서브클래스로서 표현된다. "특수화(specialization)"는 예외 핸들러를 다소 일반적인 예외적 상황과 연관하도록 해준다. 따라서 우리가 원하는 세분성의 수준에 따라 여러 표현식을 작성할 수 있다.

```
[{\dots}] on: Error do: [{\dots}] or
[{\dots}] on: FileStreamException do: [{\dots}] or
[{\dots}] on: FileDoesNotExistException do: [{\dots}]
```

FileStreamException 클래스는 그것이 설명하는 구체적인 이상한 상황을 특징짓기 위해 Exception 클래스로 정보를 추가한다. 특히 FileStreamException은 fileName 인스턴스 변수를 정의하는데, 이것은 예외를 시그널링한 파일명을 포함한다. 예외 클래스 계층구조의 루트는 Object의 직계 서브클래스인 Exception이다.

예외 처리에는 두 개의 메시지가 수반되는데, 앞에서 살펴봤듯이 on:do: 는 예외 핸들러를 설정하기 위해 블록으로 전송되고, signal은 예외가 발생하였음을 시그널링하기 위해 Exception의 서브클래스로 전송된다.

예외의 집합 포착하기

지금까지는 예외의 단일 클래스를 포착하는 데에만 on:do: 를 사용해왔다. 시그널링된 예외가 명시된 예외 클래스의 하위 인스턴스인 경우에만 핸들러가 호출될 것이다. 하지만 다수의 exception 클래스를 포착하길 원하는 상황을 상상해보자. 이는 간단한데, 아래 예제와 같이 콤마로 구분된 클래스 리스트를 명시하면 된다.

```
result := [ Warning signal . 1/0 ]
    on: Warning, ZeroDivide
    do: [:ex | ex resume: 1 ].
result → 1
```

어떻게 작동하는지가 궁금하다면 Exception class» 의 구현을 살펴보라.

```
Exception class>> anotherException
    "Create an exception set."
    ^ExceptionSet new add: self; add: anotherException; yourself
```

나머지 마법은 ExceptionSet 클래스에서 발생하는데, 그 구현은 놀라울 정도로 간단하다.

```
Object subclass: #ExceptionSet
    instanceVariableNames: 'exceptions'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Exceptions-Kernel'

ExceptionSet>>initialize
    super initialize.
    exceptions := OrderedCollection new

ExceptionSet>>, anException
    self add: anException.
    ^self

ExceptionSet>>add: anException
    exceptions add: anException

ExceptionSet>>handles: anException
    exceptions do: [:ex | (ex handles: anException) ifTrue: [^true]].
    ^false
```

handles: 메시지는 단일 예외에서 정의되고, 수신자가 예외를 처리하는지 여부를 리턴한다.

예외 시그널링하기

예외를 시그널링하기 위해서는[1] 예외 클래스의 인스턴스를 생성하여 그곳으로 텍스트 설명과 함께 signal: 메시지 signal 또를 전송하기만 하면 된다. Exception 클래스는 예외를 생성하여 시그널링하는 편의(convenience) 메서드 signal을 제공한다. ZeroDivide 예외를 간편하게 시그널링하는 방법으로 두 가지가 있다.

```
ZeroDivide new signal.
ZeroDivide signal. "class-side convenience method does the same as above"
```

예외를 시그널링하기 위해 예외를 생성하는 일을 예외 클래스가 책임지지 않고 직접 생성해야 하는 이유가 궁금할지도 모르겠다. 인스턴스의 생성이 중요한 이유는 예외가 시그널링된 컨텍스트에 관한 정보를 캡슐화하기 때문이다. 따라서 각각이 다른 예외의 컨텍스트를 설명하는 예외 인스턴스를 많이 가질 수 있다.

예외가 시그널링되면 예외 처리 메커니즘은 실행 스택에서 시그널링된 예외의 클래스와 연관된 예외 핸들러를 검색한다. 핸들러를 마주치면 (예: 스택 상에 on:do: 메시지가 있는 경우), implementation은 exceptionClass가 시그널링된 예외의 슈퍼클래스인지를 확인하고, 예외를 유일한 인자로 하여 handlerAction를 실행한다. 핸들러가 예외 객체를 사용할 수 있는 방법을 간략하게 살펴볼 것이다.

예외를 시그널링할 때는 아래 코드에서 볼 수 있듯이 방금 마주친 상황에 특정적인 정보를 제공하는 것이 가능하다. 예를 들어, 열어야 할 파일이 존재하지 않는다면 존재하지 않는 파일명이 예외 객체에 기록될 수 있다.

[1] 예외를 "발생" 시키거나 "던지는" 것과 같은 의미다. 필수 메시지는 signal 이라 불리므로 그 용어는 이번 장에서만 사용하겠다.

```
StandardFileStream class>>oldFileNamed: fileName
    ”Open an existing file with the given name for reading and writing. If the name has no
       directory part, then default directory will be assumed. If the file does not exist, an
       exception will be signaled. If the file exists, its prior contents may be modified or
        replaced, but the file will not be truncated on close.”
    | fullName |
    fullName := self fullName: fileName.
    ^(self isAFileNamed: fullName)
        ifTrue: [self new open: fullName forWrite: true]
        ifFalse: [”File does not exist...”
            (FileDoesNotExistException new fileName: fullName) signal]
```

예외 핸들러는 이상한 상황에서 회복하기 위해 이 정보를 이용할 수도 있다. 예외 핸들러 [:ex]…]에서 ex는 FileDoesNotExistException의 인스턴스이거나 그 서브클래스 중 하나의 인스턴스가 될 것이다. 이는 fileName을 전송하여 missing 파일의 파일명을 질의하는 예외이다.

```
| result |
result := [(StandardFileStream oldFileNamed: ’error42.log’) contentsOfEntireFile]
    on: FileDoesNotExistException
    do: [:ex | ex fileName , ’ not available’].
Transcript show: result; cr
```

모든 예외는 예외적 상황을 명확하고 포괄적인 방식으로 보고하기 위해 개발 툴이 사용하는 기본 설명을 갖고 있다. 설명을 이용 가능하게 만들기 위해 모든 예외 객체들은 description 메시지에 응답한다. 그리고 messageText: aDescription 메시지를 전송하거나, signal: aDescription 예외를 시그널링하면 기본 설명을 변경할 수 있다.

또 다른 시그널링 예제는 스몰토크의 반영적 기능의 중심인 doesNotUnderstand: 메커니즘에서 발생한다. 객체가 이해하지 못하는 메시지가 객체로 전송될 때마다 가상 머신은 (결국) 문제가 되는 메시지를 표현하는 인자와 함께 doesNotUnderstand: 메시지를 전송할 것이다. Object 클래스에 정의된 doesNotUnderstand: 의 기본 구현은 MessageNotUnderstood 예외를 시그널링하여 실행 중 그 시점에서 디버거가 열리도록 한다.

doesNotUnderstand: 메서드는 예외 특정적인 정보, 즉 이해하지 못한 메시지나 수신자 등의 정보를 예외로 저장하여 디버거에서 이용 가능하게 만드는 방법을 묘사한다.

```
Object>>doesNotUnderstand: aMessage
    ”Handle the fact that there was an attempt to send the given message to the receiver
        but the receiver does not understand this message (typically sent from the machine
        when a message is sent to the receiver and no method is defined for that selector).
        ”

    MessageNotUnderstood new
        message: aMessage;
        receiver: self;
        signal.
    ^ aMessage sentTo: self.
```

이것으로 예외가 어떻게 사용되는지에 대한 설명은 끝났다. 나머지 부분은 예외가 어떻게 구현되는지를 논하고, 자신의 예외를 정의할 경우에만 관련된 세부 내용을 추가하겠다.

핸들러 찾기

이제 예외가 시그널링되면 어떻게 예외 핸들러를 검색하고 실행 스택으로부터 인출하는(fetch) 지를 살펴볼 것이다. 하지만 이를 알아보기 전에 프로그램의 제어 흐름이 가상 머신에서 내부적으로 어떻게 표현되는지를 이해할 필요가 있겠다.

프로그램의 실행 시 각 포인트마다 프로그램의 실행 스택은 활성 컨텍스트(activation context)의 리스트로서 표현된다. 각 활성 컨텍스트는 메서드 호출을 나타내고, 그 실행에 필요한 모든 정보, 즉 수신자, 인자, 로컬 변수를 포함한다. 뿐만 아니라 생성을 트리거하는 컨텍스트에 대한 참조도 포함하는데, 가령 해당하는 컨텍스트를 생성한 메시지를 전송한 메서드 실행에 연관된 활성 컨텍스트를 예로 들 수 있겠다. Pharo에서는 MethodContext(슈퍼클래스가 ContextPart에 해당하는)가 이러한 정보를 모델링한다. 활성 컨텍스트들 간 참조는 그들을 사슬로 연결하는데, 이러한 활성 컨텍스트의 사슬이 바로 스몰토크의 실행 스택이다.

존재하지 않는 파일로부터 doIt 을 통해 FileStream 열기를 시도한다고 가정해보자. FileDoesNotExistException 이 시그널링되고, 실행 스택은 그림 13.2와 같이 doIt, oldFileNamed, signal에 대한 MethodContexts를 포함할 것이다.

스몰토크에서는 모든 것이 객체이므로 메서드 컨텍스트도 객체일 것으로 예상할 것이다. 하지만 일부 스몰토크 구현은 끊임없는 객체의 생성을 피하기 위해 가상 머신의 native C 실행 스택을 사용한다. 최신 Pharo 가상 머신은 사실상 항상 모든(full) 스몰토크 객체를 사용하는데, 속도를 위해 각 메시지 전송마다 메서드 컨텍스트 객체를 새로 생성하는 대신 기존의 메서드 컨텍스트 객체를 재활용한다.

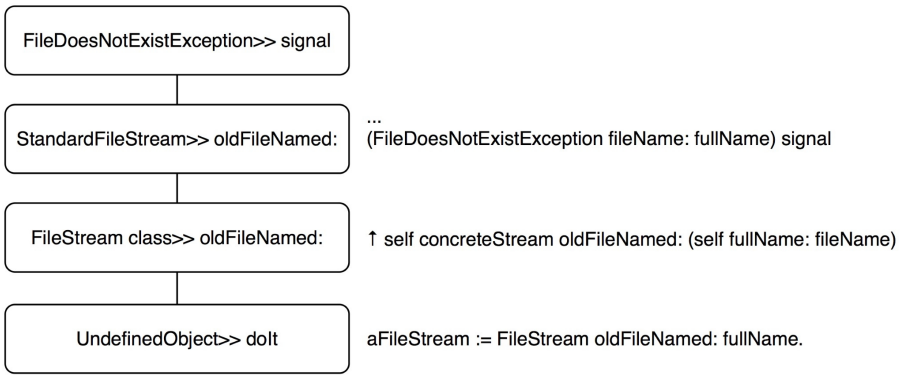


그림 13.2: Pharo 실행 스택.

aBlock on: ExceptionClass do: actionHandler 를 전송할 때는 aBlock 라는 보호 블록의 활성 컨텍스트에 대해 주어진 예외 클래스(ExceptionClass)와 예외 핸들러(actionHandler)를 연관시키려는 의도를 갖고 있다. 이러한 정보는 적절한 클래스의 예외가 시그널링될 때마다 actionHandler 를 식별하고 실행하는 데에 사용되는데, actionHandler 는 꼭대기부터 (가장 최근의 메시지 전송) 시작해 on:do: 메시지를 전송한 컨텍스트까지 내려오면서 순회하여 찾을 수 있다.

스택에서 어떤 예외 핸들러도 발견되지 않을 경우 ContextPart»handleSignal: 또는 Un-

definedObject»handleSignal: 에 의해 defaultAction 메시지가 전송된다. 후자는 스택의 맨 아래에 연관되며, 아래와 같이 정의된다.

```
UndefinedObject>>handleSignal: exception
    "When no more handler (on:do:) context is left in the sender chain, this gets called.
        Return from signal with default action."
    ^ exception resumeUnchecked: exception defaultAction
```

handleSignal: 메시지는 Exception»signal 에 의해 전송된다.

예외 E가 시그널링되면 시스템은 아래와 같이 스택을 하향으로 검색함으로써 해당하는 예외 핸들러를 식별하고 인출한다.

1. 핸들러에 대한 현재 활성 컨텍스트를 살펴보고 핸들러가 canHandleSignal: E 인지 테스트하라.
2. 스택이 비어 있지 않은데 어떤 핸들러도 발견되지 않은 경우 스택의 하단에서 단계 1로 돌아가라.
3. 어떤 핸들러도 발견되지 않고 스택도 비어 있는 경우 E로 defaultAction을 전송하라. Error 클래스 내의 기본 구현은 디버거를 여는 것이다.
4. 핸들러가 발견되는 경우 핸들러로 value: E 를 전송하라.

중첩 예외. 예외 핸들러는 그들 고유의 범위 밖에 있다. 즉, 예외가 예외 핸들러 내부로부터 시그널링될 경우를 중첩 예외라고 부르는데, 이런 일이 발생하면 중첩 예외를 포착하도록 별도의 핸들러를 설정해야 한다.

on:do: 메시지가 다른 메시지의 수신자에 해당하는 예제를 들어보되, 이 다른 메시지는 on:do: 메시지의 핸들러에 의해 시그널링되는 오류를 포착할 것이다 (on:do: 의 결과는 보호 블록이거나 핸들러 액션 블록 값임을 기억하라).

```
result := [[ Error signal: 'error 1' ]
    on: Exception
    do: [ Error signal: 'error 2' ]]
        on: Exception
        do: [:ex | ex description ].
result → 'Error: error 2'
```

두 번째 핸들러가 없다면 중첩 예외는 포착되지 않을 것이며, 디버거가 호출될 것이다.

위를 방법을 첫 번째 메시지 내에 두 번째 핸들러를 명시하는 대안책이 있다.

```
result := [ Error signal: 'error 1' ]
    on: Exception
    do: [[ Error signal: 'error 2' ]
        on: Exception
        do: [:ex | ex description ]].
result → 'Error: error 2'
```

미묘한 점이니 시도해보고 연구하길 바란다.

예외 처리하기

예외가 시그널링되면 핸들러는 예외를 처리하는 것과 관련해 여러 선택권을 갖는데, 무엇보다도 아래를 실행할 수 있다.

(i) 단순히 대안적 결과를 명시함으로써 보호 블록의 실행을 *abandon*(포기)한다. 이는 프로토콜의 일부에 해당하지만 return 과 비슷하기 때문에 사용되지 않는다.

(ii) 예외 객체로 return: aValue 를 전송함으로써 보호 블록에 대한 대안적 결과를 *return*(반환)한다.

(iii) *retry* 를 전송함으로써 보호 블록을 재시도하거나 retryUsing: 을 전송하여 다른 블록을 시도한다.

(iv) 실패 지점에서 resume 또는 resume: 을 전송함으로서 보호 블록을 *resume*(재개)한다.

(v) pass 를 전송함으로써 잡은 예외를 enclosing 핸들러로 *pass*(전달)한다.

(vi) 예외로 resignalAs: 를 전송함으로써 다른 예외를 *resignal*(재시그널링)한다.

앞의 세 가지 가능성을 간략하게 살펴본 후 나머지를 자세히 살펴보겠다.

보호 블록 포기하기

첫 번째 가능성은 아래와 같이 보호 블록의 실행을 단념하는 방법이다.

```
answer := [ |result|
    result := 6*7.
    Error signal.
    result "This part is never evaluated" ]
        on: Error
        do: [ :ex | 3 + 4 ].
answer → 7
```

핸들러는 오류가 시그널링되는 지점부터 인계 받고, 원본 블록 내에서 그 다음에 따라오는 코드는 평가되지 않는다.

return: 를 이용해 값 리턴하기

블록은 블록의 보호 여부와 상관없이 블록에서 마지막 문의 값을 리턴한다. 하지만 핸들러 블록에 의해 결과가 리턴되어야 하는 상황이 종종 있다. 예외로 전송된 return: aValue 메시지는 보호 블록의 값으로서 aValue 를 리턴하는 효과를 갖는다.

```
result := [Error signal]
    on: Error
    do: [ :ex | ex return: 3 + 4 ].
result → 7
```

ANSI 표준은 값을 리턴할 때 do: [:ex | 100] 를 사용할 때와 do: [:ex | ex return: 100]를 사용할 때 차이점에 대해 불분명하다. Pharo에선 두 표현식이 동일하긴 하지만 목적을 좀 더 알려주는 return: 을 사용할 것을 권한다.

return: 의 변형체로 return이라는 메시지가 있는데, 이는 nil을 리턴한다.

어떤 경우든 제어(control)는 보호 블록으로 리턴하진 않겠지만 enclosing 컨텍스트까지는 전달될 것임을 기억하라.

```
6\(\ast{}\)([Error signal] on: Error do: [:ex | ex return: 3 + 4]) → 42
```

retry와 retryUsing: 을 이용해 계산 재시도하기

때로는 예외를 야기한 상황을 변경하여 보호 블록을 재시도하길 원하는 경우가 있다. 이는 예외 객체로 retry 또는 retryUsing: 을 전송하면 가능하다. 예외를 야기한 조건은 보호 블록을 재시도하기 전에 변경하는 것이 중요한데, 이를 어길 경우 무한 루프가 발생할 것이다.

```
[Error signal] on: Error do: [:ex | ex retry] "will loop endlessly"
```

조금 더 나은 예제를 소개하겠다. 보호 블록은 theMeaningOfLife가 적절히 초기화되어 있는 수정된(modified) 환경에서 재평가된다.

```
result := [ theMeaningOfLife*7 ] "error -- theMeaningOfLife is nil"
    on: Error
    do: [:ex | theMeaningOfLife := 6. ex retry ].
result → 42
```

retryUsing: aNewBlock 메시지는 보호 블록을 aNewBlock으로 대체할 수 있게 해준다. 이러한 새 블록은 원본 블록과 동일한 핸들러를 이용해 실행 및 보호된다.

```
x := 0.
result := [ x/x ] "fails for x=0"
    on: Error
    do: [:ex |
        x := x + 1.
        ex retryUsing: [1/((x-1)*(x-2))] "fails for x=1 and x=2"
    ].
result → (1/2) "succeeds when x=3"
```

아래 코드는 무한 루프를 야기하는 반면,

```
[1 / 0] on: ArithmeticError do: [:ex | ex retryUsing: [ 1 / 0]]
```

아래 코드는 Error 를 시그널링할 것이다.

```
[1 / 0] on: ArithmeticError do: [:ex | ex retryUsing: [ Error signal]]
```

또 다른 예제로 앞에서 살펴본 파일 처리 코드, 즉 파일이 발견되지 않을 경우 Transcript 상에 메시지를 출력하는 예제를 상기시켜보자. 대신 아래와 같이 파일의 입력을 요청할 수 있다.

```
[(StandardFileStream oldFileNamed: 'error42.log') contentsOfEntireFile]
    on: FileDoesNotExistException
    do: [:ex | ex retryUsing: [FileList modalFileSelector contentsOfEntireFile] ]
```

실행 재개하기

예외를 시그널링하는 메서드 isResumable은 시그널링 직후에 재개가 가능하다. 따라서 예외 핸들러는 일부 액션을 실행한 후 실행 흐름을 재개할 수 있다. 이러한 행위는 핸들러 내 예외로 resume: 을 전송함으로써 얻는다. 인자는 예외를 시그널링한 표현식 대신에 사용될 값이다. 아래 예제에서는 시그널링한 다음 Tests-Exceptions 범주에서 정의되는 MyResumableTestError를 포착할 것이다.

```
result := [ | log |
    log := OrderedCollection new.
    log addLast: 1.
    log addLast: MyResumableTestError signal.
    log addLast: 2.
    log addLast: MyResumableTestError signal.
    log addLast: 3.
    log ]
        on: MyResumableTestError
        do: [ :ex | ex resume: 0 ].
result → an OrderedCollection(1 0 2 0 3)
```

여기서 우리는 MyResumableTestError signal의 값이 resume: 메시지에 대한 인자 값임을 분명히 확인할 수 있다.

resume 메시지는 resume: nil 과 같다.

예외를 재개할 경우 이점은 아래와 같이 로딩되는 기능을 통해 설명된다. 패키지를 설치 시 경고가 시그널링되기도 한다. 경고는 위험한 오류로 간주되어선 안 되기 때문에 경고를 단순히 무시하고 설치를 계속하면 된다. PackageInstaller 클래스는 존재하지 않지만 가능한 구현을 요약하자면 다음과 같다.

```
PackageInstaller>>installQuietly: packageNameCollection
    ....
    [ self install ] on: Warning do: [ :ex | ex resume ].
```

재개가 유용한 또 다른 상황으로, 사용자에게 어떤 일을 하도록 요청할 때를 들 수 있겠다. 예를 들어, 아래 메서드를 이용해 ResumableLoader 클래스를 정의한다고 가정해보자.

```
ResumableLoader>>readOptionsFrom: aStream
    | option |
    [aStream atEnd]
        whileFalse: [option := self parseOption: aStream.
            "nil if invalid"
            option isNil
                ifTrue: [InvalidOption signal]
                ifFalse: [self addOption: option]].
```

유효하지 않은 옵션을 마주치면 InvalidOption 예외를 시그널링한다. readOptionsFrom: 을 전송하는 컨텍스트는 적절한 핸들러를 준비할 수 있다.

```
ResumableLoader>>readConfiguration
| stream |
stream := self optionStream.
[self readOptionsFrom: stream]
    on: InvalidOption
    do: [:ex | (UIManager default confirm: 'Invalid option line. Continue loading?')
        ifTrue: [ex resume]
        ifFalse: [ex return]].
stream close
```

스트림을 확실히 닫히도록 확보하기 위해서는 ensure: 호출을 이용해 stream close를 보호해야 한다.

사용자 입력에 따라 readConfiguration 내 핸들러는 nil(nil을 리턴)을 리턴하거나, 예외를 resume(재개)하여 readOptionsFrom: 내에서 리턴해야 할 signal 메시지 전송과 계속해야 할 옵션 스트림의 파싱을 야기한다.

InvalidOption은 재개 가능해야 함을 주목하고, 그것만으로도 Exception의 하위클래스로서 정의하기엔 충분하다.

사용 방법은 resume: 의 전송자를 살펴보도록 한다.

예외 전달하기

예외의 전달과 같은 예외 처리의 다른 방법들을 설명하기 위해 perform: 메서드의 일반화(generalization)를 구현하는 방법을 살펴보도록 하겠다. 객체로 perform: aSymbol 을 전송하면 이는 aSymbol 이라는 메시지를 해당 객체로 전송되도록 야기할 것이다.

```
5 perform: \#factorial → 120 "same as: 5 factorial"
```

해당 메서드의 여러 변형체가 존재하는데, 아래를 예로 들 수 있겠다.

```
1 perform: \#+ withArguments: \#(2) → 3 "same as: 1 + 2"
```

perform: 과 닮은 메서드들은 동적으로 인터페이스로 접근하기에 매우 유용한데, 전송되는 메시지들을 런타임 시 결정할 수 있기 때문이다. 누락된 메시지가 하나 있는데 이는 주어진 수신자로 단항 메시지의 cascade를 전송하는 메시지다. 단순하면서 간단한 구현을 소개하겠다.

```
Object>>performAll: selectorCollection
    selectorCollection do: [:each | self perform: each] "aborts on first error"
```

해당 메서드는 아래와 같이 사용 가능하다.

```
Morph new performAll: \#( \#activate \#beTransparent \#beUnsticky)
```

하지만 문제가 하나 있다. 컬렉션 내에 객체가 이해하지 못하는 선택자가 존재할 수 있다는 것이다 (예 : #activate). 그러한 선택자는 무시하고 나머지 메시지의 전송을 계속하겠다. 아래 구현이 적당해 보인다.

```
Object>>performAll: selectorCollection
    selectorCollection do: [:each |
        [self perform: each]
            on: MessageNotUnderstood
            do: [:ex | ex return]] "also ignores internal errors"
```

좀 더 자세히 살펴보니 또 다른 문제가 있다. 이 핸들러는 본래 수신자가 이해하지 못하는 메시지를 포착하고 무시할 뿐만 아니라 이해된 메시지에 대해, 메서드 내에서 전송되었으나 이해되지 않은 메시지들도 포착하고 무시할 것이란 점이다. 이는 그러한 메서드에 프로그래밍 오류를 숨길 것인데, 우리 의도와는 다르다. 현재 선택자의 실행을 시도하여 야기되었는지 확인하기 위해 예외를 분석하는 핸들러가 필요하다. 올바른 구현은 다음과 같다.

메서드 13.1: Object»performAll:

```
Object>>performAll: selectorCollection
    selectorCollection do: [:each |
        [self perform: each]
            on: MessageNotUnderstood
            do: [:ex | (ex receiver == self and: [ex message selector == each])
                ifTrue: [ex return]
                ifFalse: [ex pass]]] "pass internal errors on"
```

MessageNotUnderstood 오류가 만일 우리가 실행 중인 메시지 리스트에 속하지 않는 경우, 이는 그러한 오류를 주위 컨텍스트로 전달하는 효과를 일으킨다. pass 메시지는 실행 스택에서 다음으로 적용 가능한 핸들러로 예외를 전달할 것이다.

스택에 다음 핸들러가 존재하지 않을 경우 defaultAction 메시지가 예외 인스턴스로 전송된다. pass 액션은 조금도 전송자 사슬을 수정하지 않지만 제어(control)가 전달되는 핸들러는 전송자 사슬을 수정할지도 모른다. 이번 절에서 논한 다른 메시지들과 마찬가지로 pass 또한 특별한 메시지인데, 절대로 전송자에게 리턴하지 않기 때문이다.

이번 절의 목표는 예외의 강점을 설명하는 데에 있다. 예외를 이용해 거의 모든 일을 할 수 있는 반면 그로 야기되는 코드를 이해하기가 항상 쉬운 것은 아니란 사실이 명백해진다. 예외를 사용하지 않고 더 단순한 방식으로 동일한 효과를 얻는 방법도 있는데, performAll: 을 더 잘 구현하는 방법은 291 페이지의 메서드 13.2를 참고하라.

예외 재전송하기

performAll: 예제에서 수신자가 이해하지 못한 선택자를 더 이상 무시하지 말고 그러한 선택자의 발생을 오류로 고려하길 원한다고 가정해보자. 하지만 그러한 선택자는 일반적인 MessageNotUnderstood가 아니라 애플리케이션 특정적 예외, 즉 InvalidAction으로서 시그널링되길 원한다. 다시 말하자면, 우리는 시그널링된 예외를 다른 예외로서 "재시그널링"할 수 있는 능력을 원한다고 볼 수 있겠다.

언뜻 보면 핸들러 블록에서 새 예외를 시그널링하는 것으로 단순해 보일지도 모른다. performAll: 의 구현에서 핸들러 블록은 다음과 같을 것이다.

```
[:ex | (ex receiver == self and: [ex message selector == each])
    ifTrue: [InvalidAction signal] "signals from the wrong context"
    ifFalse: [ex pass]]
```

하지만 좀 더 가까이 살펴보면 미묘한 문제가 존재한다. 본래 의도는 MessageNotUnderstand의 발생을 InvalidAction으로 대체하는 것이었다. 이러한 대체는 프로그램 내에서 원래의 MessageNotUnderstood 예외와 같은 장소에서 InvalidAction이 시그널링되는 효과를 보여야 한다. 그런데 위의 해결책은 InvalidAction을 다른 장소에서 시그널링한다. 위치의 차이는 적용 가능한 핸들러의 차이로 이어질 것이다.

이러한 문제를 해결하기 위해 예외를 재시그널링하는 것은 시스템이 처리하는 특수 액션이다. 이러한 목적으로 시스템은 resignalAs: 메시지를 제공한다. performAll: 예제에서 핸들러 블록의 올바른 구현은 다음과 같을 것이다.

```
[:ex | (ex receiver == self and: [ex message selector == each])
    ifTrue: [ex resignalAs: InvalidAction] "resignals from original context"
    ifFalse: [ex pass]]
```

outer와 pass 비교하기

ANSI 프로토콜은 outer 행위를 명시하기도 한다. outer 메서드는 pass와 매우 유사하다. 예외로 outer를 전송해도 enclosing 핸들러 액션을 평가한다. 유일한 차이는 outer 핸들러는 예외를 재개한 후 애초에 예외가 시그널링된 장소가 아니라 outer가 전송된 지점으로 제어가 리턴될 것이란 점이다.

```
passResume := [[ Warning signal . 1 ] "resume to here"
    on: Warning
    do: [ :ex | ex pass . 2 ]]
        on: Warning
        do: [ :ex | ex resume ].
passResume → 1 "resumes to original signal point"

outerResume := [[ Warning signal . 1 ]
    on: Warning
    do: [ :ex | ex outer . 2 ]] "resume to here"
        on: Warning
        do: [ :ex | ex resume ].
outerResume → 2 "resumes to where outer was sent"
```

예외와 ensure:/ifCurtailed: 상호작용

예외가 어떻게 작용하는지 보았으니 예외와 ensure: 또는 ifCurtailed: 구문 간 상호 작용을 제시하겠다. ensure: 또는 ifCurtailed: 블록은 예외 핸들러가 실행되고 나서야 실행된다. ensure: 인자는 항상 실행되는 반면 ifCurtailed: 인자는 그 수신자 실행이 스택의 언와인딩(unwinding)을 야기한 경우에만 실행된다.

아래 예제가 그러한 행위를 보여준다. 이는 should show first error 다음에 then should show curtailed 를 출력한 후 4를 리턴한다.

```
[[ 1/0 ]
    ifCurtailed: [ Transcript show: 'then should show curtailed'; cr. 6 ]]
    on: Error do: [ :e |
        Transcript show: 'should show first error'; cr.
        e return: 4 ].
```

먼저 [1/0] 는 제로 오류(0으로 나눗셈)를 발생시킨다. 이러한 오류는 예외 핸들러에 의해 처리된다. 이는 첫 번째 메시지를 출력한다. 이후 값 4를 리턴하고, 수신자가 오류를 야기하였기 때문에 ifCurtailed: 메시지의 인자가 평가되어 두 번째 메시지를 출력한다. ifCurtailed: 는 오류 핸들러나 ifCurtailed: 인자에 의해 표현된 리턴 값을 변경하지 않음을 주목하라.

아래 표현식은, 스택이 언와인딩 되지 않았을 경우 표현식 값이 단순히 리턴되거나 어떤 핸들러도 실행되지 않음을 보여준다. 1이 리턴된다.

```
[[ 1 ]
        ifCurtailed: [ Transcript show: 'curtailed'; cr. 6 "does not display it" ]]
    on: Error do: [ :e |
        Transcript show: 'error'; cr. "does not display it"
        e return: 4 ].
```

ifCurtailed: 은 스택 이상 행위에 대해 반응하는 watchdog다. 예를 들어, 이전 표현식의 수신자 내에 리턴 문을 추가할 경우 ifCurtailed: 메시지의 인자가 발생할 것이다. 사실상 리턴 문은 메서드에서 정의되지 않았으므로 유효하지 않다.

```
[[ ^ 1 ]
        ifCurtailed: [ Transcript show: 'only shows curtailed'; cr. ]]
    on: Error do: [ :e |
        Transcript show: 'error 2'; cr. "does not display it"
        e return: 4 ].
```

아래 예제는 오류가 발생하지 않더라도 ensure: 가 체계적으로 실행됨을 보여준다. 여기서는 should show ensure 메시지가 표시되고 값으로 1이 리턴된다.

```
[[ 1 ]
        ensure: [ Transcript show: 'should show ensure'; cr. 6 ]]
    on: Error do: [ :e |
        Transcript show: 'error'; cr. "does not display it"
        e return: 4 ].
```

아래 표현식은 앞에서와 같이 오류가 발생하면 ensure: 인자 이전에 오류와 연관된 핸들러가 실행됨을 보여준다. 여기서 표현식은 should show error first를 출력하고 나서 then should show ensure를 출력한 후 4를 리턴한다.

```
[[ 1/0 ]
        ensure: [ Transcript show: 'then should show ensure'; cr. 6 ]]
    on: Error do: [ :e |
        Transcript show: 'should show error first'; cr.
        e return: 4 ].
```

마지막으로 아래 표현식은 오류에서 가장 가까운 오류부터 가장 먼 오류까지 오류가 하나씩 실행된 후에 ensure: 인자가 실행됨을 보여준다. 여기서는 error1, error2, then should show ensure 순으로 표시된다.

```
[[[ 1/0 ] ensure: [ Transcript show: 'then should show ensure'; cr. 6 ]]
    on: Error do: [ :e|
        Transcript show: 'error 1'; cr.
        e pass ]] on: Error do: [ :e |
            Transcript show: 'error 2'; cr. e return: 4 ].
```

예제: Deprecation

Deprecation은 재개 가능한 예외를 이용하여 빌드된 메커니즘의 사례 연구를 제공한다. deprecation은 소프트웨어 재공학 패턴으로서, "오래되어 사라진(deprecated)" 메서드를 표시하도록 해주는데, 즉 향후 배포판에서는 사라질 수 있으며 새 코드에서 사용해서는 안 된다는 의미다. Pharo에서는 아래와 같이 오래되어 사라진 메서드를 표시할 수 있다.

```
Utilities class>>convertCRtoLF: fileName
   "Convert the given file to LF line endings. Put the result in a file with the extention '.lf'"

    self deprecated: 'Use ''FileStream convertCRtoLF: fileName'' instead.'
        on: '10 July 2009' in: #Pharo1.0 .
    FileStream convertCRtoLF: fileName
```

convertCRtoLF: 메시지가 전송되었을 때 raiseWarning 설정이 true로 된 경우 팝업 창과 함께 알림이 표시되면서 프로그래머가 애플리케이션 실행을 재개할 수 있게 되는데, 이를 그림 13.3에 소개하고 있다 (Settings는 제 5장에서 상세히 설명한 바 있다). 물론 해당 메서드는 오래되어 사라졌기 때문에 현재 Pharo 배포판에선 찾아볼 수 없다. deprecated:on:in: 의 또 다른 전송자를 살펴보라.

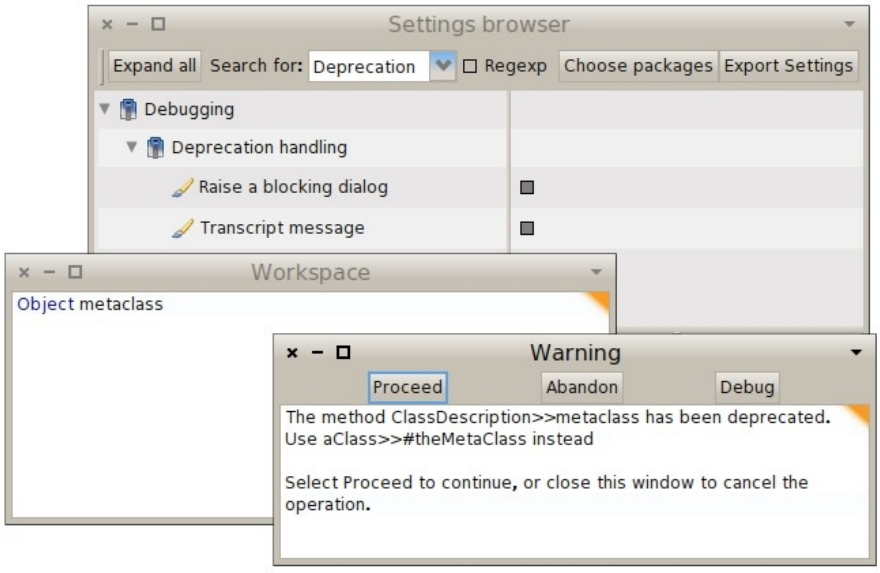


그림 13.3: 오래되어 사라진 메시지 전송하기.

deprecation은 몇 가지 단계를 통해 Pharo에서 구현된다. 첫째, Deprecation을 Warning의 하위클래스로서 정의한다. deprecation에 관한 정보를 포함하기 위한 인스턴스 변수를 몇 개 가져야 하는데 Pharo에서는 methodReference, explanationString, deprecationDate, versionString이 해당한다. 따라서 이러한 변수들에 대한 인스턴스 측 initialization 메서드를 정의하고, 그에 해당하는 메시지를 전송하는 클래스 측 인스턴스 생성 메서드도 정의할 필요가 있다.

새로운 예외 클래스를 정의할 때는 isResumable, description, defaultAction 의 오버라이드를 고려해야 한다. 이런 경우 첫 두 메서드에 상속된 구현이 괜찮다.

- isResumable 은 Exception으로부터 상속되고, true를 응답한다.
- description은 Exception으로부터 상속되고, 적절한 텍스트 설명을 응답한다.

하지만 defaultAction의 구현은 필수적으로 오버라이드 해야 하는데, 일부 설정에 따라 달라지길 원하기 때문이다. Pharo의 구현을 살펴보자.

```
Deprecation>>defaultAction
    Log ifNotNil: [:log| log add: self].
    self showWarning ifTrue:
        [Transcript nextPutAll: self messageText; cr; flush].
    self raiseWarning ifTrue:
        [super defaultAction]
```

첫 번째 개인설정은 단순히 Transcript 상에 경고 메시지가 나타나도록 야기할 뿐이다. 두 번째 개인설정은 예외의 시그널링을 요청하는데, 이는 super를 전송하는 defaultAction을 이용해 이루어진다.

Object에서 몇 가지 편의 메서드도 구현할 필요가 있으며, 일례를 들자면 다음과 같다.

```
Object>>deprecated: anExplanationString on: date in: version
    (Deprecation
        method: thisContext sender method
        explanation: anExplanationString
        on: date
        in: version) signal
```

예제: Halt 구현

Pharo By Example 의 Debugger 장에서 논한 바와 같이 스몰토크 메서드 내에서는 메시지 전송 self halt를 코드로 삽입하는 것이 중단점을 설정하는 가장 일반적인 방법이다. Object에서 구현되는 halt 메서드는 중단점의 위치에서 디버거를 열기 위해 예외를 사용하는데, 이는 아래와 같이 정의된다.

```
Object>>halt
    "This is the typical message to use for inserting breakpoints during
    debugging. It behaves like halt:, but does not call on halt: in order to
    avoid putting this message on the stack. Halt is especially useful when
    the breakpoint message is an arbitrary one."
    Halt signal
```

Halt는 Exception의 직계 하위클래스다. Halt 예외는 재개 가능한데, 즉 Halt가 시그널링된 후 실행을 계속하는 것이 가능하다는 의미다.

Halt는 예외가 포착되지 않을 경우 (예: 실행 스택 어디에서도 Halt에 대한 예외 핸들러를 찾을 수 없는 경우) 실행할 액션을 명시하는 defaultAction 메서드를 오버라이드한다.

```
Halt>>defaultAction
    "No one has handled this error, but now give them a chance to decide
    how to debug it. If no one handles this then open debugger
    (see UnhandledError-defaultAction)"
    UnhandledError signalForException: self
```

위의 코드는 핸들러가 존재하지 않음을 전달하는 새로운 예외, UnhandledError를 시그널링한다. UnhandledError의 defaultAction은 디버거를 여는 것이다.

```
UnhandledError>>defaultAction
    "The current computation is terminated. The cause of the error should be logged or
        reported to the user. If the program is operating in an interactive debugging
        environment the computation should be suspended and the debugger activated."
    ^ UIManager default unhandledErrorDefaultAction: self exception

MorphicUIManager>>unhandledErrorDefaultAction: anException
    ^ Smalltalk tools debugError: anException.
```

여러 개의 메시지 다음에 디버거가 열린다.

```
Process>>debug: context title: title full: bool
    ^ Smalltalk tools debugger
        openOn: self
        context: context
        label: title
        contents: nil
        fullView: bool.
```

특정 예외

Pharo에서 Exception 클래스는 그림 13.4과 같이 10개의 직계 하위클래스를 갖는다. 그림에서 가장 먼저 눈에 띄는 점은 Exception 계층구조가 약간 엉망이라는 점이며, Pharo가 향상되면서 세부 내용이 변경될 것으로 예상할 수 있다.

두 번째로 눈에 들어오는 내용은 두 개의 하위 계층구조, Error와 Notification이 존재한다는 사실이다. 오류들은 프로그램이 이상한 상황에 빠졌음을 알려준다. 반대로 Notifications는 이벤트가 발생하였다고 말하지만 그것이 비정상적이라는 가정은 없다. 따라서 Notification이 처리되지 않으면 프로그램은 계속해서 실행될 것이다. Notification의 중요한 서브클래스로 Warning이 있는데, 경고는 시스템의 다른 부분들이나 사용자에게 비정상적이지만 위험하지 않은 행위를 통지하는 데에 사용된다.

재개 가능성이란 프로퍼티는 대개 계층구조에서 예외의 위치와 직교를 이룬다. 일반적으로 Errors는 재개 가능하지 않지만 그 서브클래스들 중 10개는 재개 가능하다. 예를 들어, MessageNotUnderstood는 Error의 서브클래스지만 재개 가능하다. TestFailures는 재개 가능하지 않지만, 이름에서 알 수 있듯이 ResumableTestFailures는 재개 가능하다.

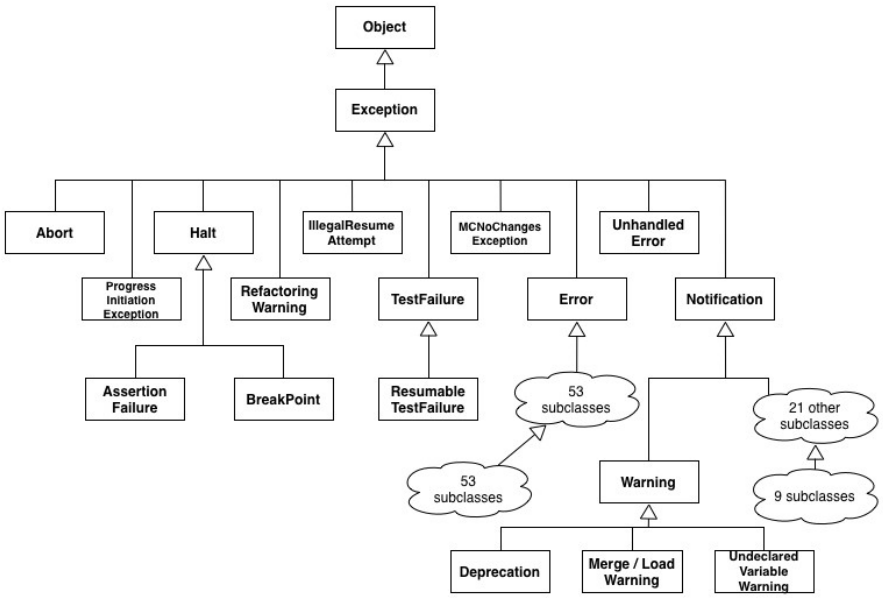


그림 13.4: Pharo 예외 계층구조의 일부분.

재개 가능성은 private Exception 메서드인 isResumable에 의해 제어되는데, 아래에 예를 소개하겠다.

```
Exception new isResumable → true
Error new isResumable → false
Notification new isResumable → true
Halt new isResumable → true
MessageNotUnderstood new isResumable → true
```

알다시피 대략 모든 예외의 2/3은 재개 가능하다.

```
Exception allSubclasses size → 160
(Exception allSubclasses select: [:each | each new isResumable]) size → 79
```

예외의 새 서브클래스를 선언할 경우 그 프로토콜에서 isResumable 메서드를 검색하여 자신의 예외 구문에 적절한 것으로 오버라이드해야 한다.

일부 상황에서는 예외를 재개하는 의미가 없을 것이다. 그러한 경우, 당신은 재개 가능하지 않은 서브클래스를 시그널링해야 하는데, 기존에 존재하는 서브클래스를 사용해도 되고 새로 생성해도 좋다. 그 외의 상황에서는 필요가 없는 핸들러 없이 예외를 재개하는 방법은 언제든 괜찮을 것이다. 사실 이는 알림(notification)을 특징 짓는 또 다른 방법을 제공하는데, Notification은 시스템 상태를 먼저 수정하지 않고 안전하게 재개할 수 있는 재개 가능 Exception이다. 시스템의 상태가 먼저 어떤 방법으로든 수정될 경우에만 예외를 재개하는 것이 안전한 경우도 자주 있을 것이다. 따라서 재개 가능 예외를 시그널링할 경우에는 예외를 재개하기 전에 예외 핸들러가 그러한 일을 실행할 것임을 분명히 확신해야 한다.

새로운 예외를 정의할 때. 기존 예외를 재사용하지 않고 새로운 예외를 정의하는 것이 가치가 있는 경우가 언제인지 결정하기란 까다로운 문제다. 몇 가지 경험적 지식을 알려주겠다. 첫째, 예외적 상황에 대해 적절한 해결책이 있다면 평가해야 한다. 둘째, 예외적 상황이 처리되지 않을 경우 구체적인 기본 행위가 필요하다. 셋째, 예외 사례를 다루기 위해 더 많은 정보를 보관할 필요가 있다.

예외를 사용하면 안 되는 경우

Pharo가 예외 처리를 갖고 있다고 해서 그 사용이 언제나 적절하다는 결론을 내려선 안 된다. 이번 장의 서론 내용을 상기시켜보면, 예외 처리란 예외적 상황을 위한 것임을 언급한 바 있다. 따라서 예외를 사용하기 위한 첫 번째 규칙은 일반적인 실행에서 타당하게 발생할 것으로 예상되는 상황에서는 예외를 사용하지 않는 것이다.

물론 라이브러리를 작성한다면 라이브러리가 사용되는 컨텍스트에 따라 그 기준이 좌우될 것이다. 구체적으로 설명하기 위해 Dictionary를 예로 살펴볼 것인데, aDictionary at: aKey 는 aKey 가 존재하지 않을 경우 Error를 시그널링할 것이다. 하지만 이 오류에 대해 핸들러를 작성해선 안 된다! 자신의 애플리케이션의 로직에서 dictionary 내에 키가 존재하지 않을 가능성이 있다면 at: aKey ifAbsent: [remedial action]를 대신 사용해야 한다. 사실 Dictionary»at: 는 Dictionary»at:ifAbsent: 를 이용해 구현된다. aCollection detect: aPredicateBlock 도 유사한데, 이를 이용해도 충족시키지 못할 가능성이 있을 경우 aCollection detect: aPredicateBlock ifNone: [remedial action]를 사용해야 한다.

예외를 시그널링하는 메서드를 작성할 때는 remedial block을 추가 인자로서 취하는 대안적 메서드도 제공해야 하는지의 여부를 고려하고, 일반 액션을 완료할 수 없는지를 평가해야 한다. 이러한 기법은 closures를 지원하는 프로그래밍 언어라면 어디서든 사용 가능하지만, 스몰토크에서는 그 모든 제어 구조에 대해 closures를 사용하므로 특히 사용하기가 자연스럽다.

예외 처리를 피하는 또 다른 방법은 예외를 시그널링할 수 있는 메시지를 전송하기 전에 예외의 전제조건(precondition)을 테스트하는 방법이 있다. 예를 들어, 메서드 13.1에서 우리는 perform: 을 이용해 객체로 메시지를 전송하고, 그에 뒤따를 수 있는 MessageNotUnderstood 오류를 처리하였다. perform: 을 실행하기 전에 메시지가 이해되었는지 확인하는 더 간단한 방법으로 다음을 소개하겠다.

메서드 13.2: Object»performAll: revisited

```
performAll: selectorCollection
    selectorCollection
        do: [:each | (self respondsTo: each)
            ifTrue: [self perform: each]]
```

메서드 13.2에 대한 주요 반론으로 효율성을 들 수 있다. respondsTo: 의 구현은 s 가 이해될 것인지 알아내기 위해 대상의 메서드 dictionary에서 s를 검색해야 한다. 응답이 yes일 경우 perform: 은 다시 검색할 것이다. 뿐만 아니라 첫 번째 구현은 스몰토크에서 구현되지만 가상 머신에선 구현되지 않는다. 코드가 만일 성능 결정적인 루프에 있을 경우 문제가 될지도

모른다. 하지만 메시지의 컬렉션이 사용자 상호작용에서 온다면 performAll: 의 속도는 문제가 되지 않을 것이다.

예외 구현

지금까지는 예외가 가상 머신 수준에서 어떻게 구현되는지 깊이 설명하지 않고 예외의 사용만 설명해왔다. 예외를 사용하려면 어떻게 구현되어야 하는지 모르고 있으므로 본 저서를 처음 읽는 독자라면 이번 절을 건너뛰어도 좋다. 하지만 가상 머신 수준에서 예외가 어떻게 구현되는지 궁금하고 알고 싶다면 꼭 읽기를 바란다. 메커니즘은 꽤 단순하여서 그것이 어떻게 작동하는지 알만한 가치가 있다. 가상 머신 수준에서 예외가 어떻게 구현되는지를 살펴보고 그 정보를 보관하기 위해 스택 실행 요소(컨텍스트)를 사용해보자.

핸들러 보관하기. 먼저 예외 클래스와 그에 연관된 핸들러가 어떻게 저장되는지, 이러한 정보를 런타임 시 어떻게 발견하는지 이해할 필요가 있다. BlockClosure 클래스에 정의된 on:do: 라는 중심(central) 메서드의 정의를 살펴보자.

```
BlockClosure>>on: exception do: handlerAction
    ”Evaluate the receiver in the scope of an exception handler.”
    | handlerActive |
    <primitive: 199>
    handlerActive := true.
    ^self value
```

이 코드는 두 가지를 알려주는데, 첫째, 해당 메서드는 프리미티브로서 구현되어서 메서드가 호출되면 가상 머신의 원시 연산이 실행된다. VM 프리미티브는 보통 리턴하지 않으며, 프리미티브를 연속적으로 실행할 경우 <primitive: n> 명령어를 포함한 메서드를 종료시키고 프리미티브의 결과를 응답한다. 따라서 프리미티브를 따르는 스몰토크 코드는 두 가지 목적을 달성하는데, 이는 프리미티브가 하는 일을 기록하고, 프리미티브가 실패 시 실행되는 역할을 하는 것이다. on:do: 는 단순히 임시 변수 handlerActive를 true로 설정한 후 수신자(물론 블록에 해당)를 평가하는 일만 한다.

이는 놀랍도록 간단하고 다소 헷갈리기도 한다. on:do: 메서드의 인자는 어디 보관될까? 이에 대한 답은 인스턴스들이 실행 스택 요소를 표현하는 MethodContext 클래스의 정의를 살펴보면 얻을 수 있겠다. 제 14장에서 설명한 바와 같이 컨텍스트(다른 언어에선 활성 레코드 또는 스택 프레임이라고도 불리는)는 특정 실행 지점을 나타낸다 (프로그램 카운터를 유지하고, 실행해야 할 다음 명령어, 이전 컨텍스트, 인자, 수신자를 가리킨다).

```
ContextPart variableSubclass: #MethodContext
    instanceVariableNames: ’method closureOrNil receiver’
    classVariableNames: ’’
    poolDictionaries: ’’
    category: ’Kernel-Methods’
```

여기서는 예외 클래스나 핸들러를 보관할 인스턴스 변수도 없을 뿐더러 슈퍼클래스에서는 그들을 보관할 공간조차 없다. 하지만 MethodContext는 variableSubclass로서 정의됨을 주목하라. 이는 명명된 인스턴스 변수 뿐만 아니라 해당 클래스의 인스턴스들 또한 몇몇 색인된 슬롯을 갖고 있음을 의미한다. 사실상 모든 MethodContext는 그것이 나타내는 호출의 메서드에서 각 인자마다 색인된 슬롯을 갖고 있다. 메서드의 임시 변수마다 추가 색인 슬롯도 있다.

해당하는 경우, on:do: 메시지의 인자는 스택 실행 인스턴스의 색인 변수에 보관된다. 이를 검증하기 위해 아래의 코드 조각을 평가하라.

```
| exception handler |
    [ thisContext explore.
    self halt.
    exception := thisContext sender at: 1.
    handler := thisContext sender at: 2.
    1 / 0]
        on: Error
        do: [:ex | 666].
^ {exception. handler} explore
```

보호 블록에서 우리는 thisContext sender를 이용해 보호 블록 실행을 표현하는 스택 요소를 질의한다. 이러한 실행은 on:do: 메시지 실행에 의해 트리거된다. 마지막 행은 예외 클래스와 예외 핸들러를 포함하는 2-요소 배열을 검색한다.

halt를 사용하여 이상한 결과를 얻고 보호 블록 내부를 검사한다면 가상 머신에 의해 재활용되는 컨텍스트에 주의를 기울여라. thisContext에서 explorer를 열 경우 컨텍스트 전송자가 사실상 on:do: 메서드의 실행임을 보여줄 것이다.

아래 코드를 이용해 explorer를 얻을 수 있을 뿐만 아니라 예외 클래스와 핸들러는 메서드 컨텍스트 객체의 첫 번째와 두 번째 가변 인스턴스 변수에 보관된다는 사실을 볼 수 있다 (메서드 컨텍스트란 실행 스택 요소를 나타낸다).

```
[thisContext sender explore] on: Error do: [:ex|].
```

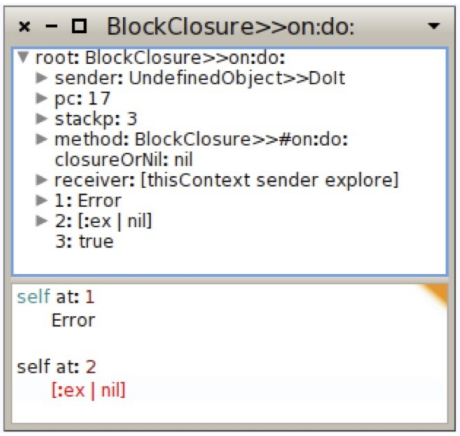


그림 13.5: 예외 클래스와 핸들러를 찾기 위해 메서드 컨텍스트 탐색하기.

on:do: 실행은 예외 클래스와 그 핸들러를 메서드 컨텍스트(실행 스택 프레임) 상에 보관함을 볼 수 있다. 이는 on:do: 에 한정된 것은 아니지만 여느 메시지 실행이든 인자를 스택 프레임에 보관한다는 사실을 주목하라.

핸들러 찾기. 정보가 어디에 보관되는지 알았으니 런타임 시 정보를 어떻게 찾는지 살펴보자.

primitive 199(on:do: 에 의해 사용되는)는 쓰기가 복잡하다고 생각할지 모르겠다. 하지만 primitive 199는 항상 실패하기 때문에 너무 평범하다! 프리미티브는 항상 실패할 것이고, on:do: 에 대한 스몰토크 body가 항상 실행된다. 하지만 <primitive: 199> 라는 표기법은 실행 컨텍스트를 유일한 방법으로 표시한다.

프리미티브의 소스 코드는 VMMaker SqueakSource 패키지의 Interpreter»primitiveMarkHandlerMethod 에서 찾을 수 있다.

```
primitiveMarkHandlerMethod
    "Primitive. Mark the method for exception handling. The primitive must fail after
    marking the context so that the regular code is run."
    self inline: false.
    ^self primitiveFail
```

on:do: 메서드가 언제 실행되는지 알았으니 스택 프레임을 구성하는 MethodContext가 태그되고 핸들러와 예외 클래스가 그 곳에 보관된다.

이제 예외가 스택 윗방향으로 시그널링되면 signal 메서드는 적절한 핸들러를 찾기 위해 스택을 검색하는데, 이러한 과정은 모두 아래 코드로부터 발생한다.

```
Exception>>signal
    "Ask ContextHandlers in the sender chain to handle this signal.
    The default is to execute and return my defaultAction."

    signalContext := thisContext contextTag.
    ^ thisContext nextHandlerContext handleSignal: self

ContextPart>>nextHandlerContext

    ^ self sender findNextHandlerContextStarting
```

findNextHandlerContextStarting 메서드는 프리미티브로서 구현되고 (number 197), 그 body는 그것이 하는 일을 설명한다. 스택 프레임이 마치 on:do: 메서드의 실행으로 인해 생성된 컨텍스트인 것처럼 보인다 (프리미티브 번호가 199번인 것처럼 보인다). 해당 컨텍스트를 응답하는 경우는 아래와 같다.

```
ContextPart>>findNextHandlerContextStarting
    "Return the next handler marked context, returning nil if there
    is none. Search starts with self and proceeds up to nil."
    | ctx |
    <primitive: 197>
    ctx := self.
    [ ctx isHandlerContext ifTrue: [^ctx].
        (ctx := ctx sender) == nil ] whileFalse.
    ^nil

MethodContext>>isHandlerContext
    "is this context for method that is marked?"
    ^method primitive = 199
```

findNextHandlerContextStarting 가 제공하는 메서드 컨텍스트는 모든 예외 처리 정보를 포함하므로 예외 클래스가 현재 예외를 처리하기에 적절한지 확인할 수 있다. 만일 적합하다면 연관된 핸들러를 실행할 수 있고, 그렇지 않을 경우 검색(look-up)이 계속된다. 이 모든 것은 handleSignal: 메서드에서 구현된다.

```
ContextPart>>handleSignal: exception
    ”Sent to handler (on:do:) contexts only. If my exception class (first arg) handles
        exception then execute my handle block (second arg), otherwise forward this
        message to the next handler context. If none left, execute exception’s defaultAction
        (see nil>>handleSignal:).”

    | val |
    (((self tempAt: 1) handles: exception) and: [self tempAt: 3]) ifFalse: [
        ^ self nextHandlerContext handleSignal: exception].

    exception privHandlerContext: self contextTag.
    self tempAt: 3 put: false. ”disable self while executing handle block”
    val := [(self tempAt: 2) valueWithPossibleArgs: {exception}]
        ensure: [self tempAt: 3 put: true].
    self return: val. ”return from self if not otherwise directed in handle block”
```

해당 메서드가 tempAt: 를 이용해 예외 클래스로 접근하여 그것의 예외 처리 여부를 묻는 방식을 주목하라. tempAt: 3은 어떤가? 이것은 on:do: 메서드의 handlerActive 임시 변수다. handlerActive가 true인지 테스트한 후 false로 설정하면 스스로 시그널링하는 예외를 핸들러가 처리하지 않도록 보장한다. handleSignal 의 마지막 액션으로서 전송되는 return: 메시지는 self 위에서 스택 프레임을 제거함으로써 실행 스택의 ”언와인딩”을 책임진다.

요약하자면, signal 메서드는 약간의 가상 머신의 도움을 받아 적절한 예외 클래스를 이용해 on:do: 메시지에 해당하는 컨텍스트를 찾는다. 실행 스택은 여느 객체와 동일하게 조작할 수 있는 Context 객체들로 구성되기 때문에 스택은 언제든 단축될 수 있다. 이는 스몰토크의 유연성을 보여주는 최고의 예이다.

Ensure: 의 구현

이제 ensure: 메서드의 구현을 살펴볼 것을 권한다.

먼저 unwind 블록이 어떻게 보관되는지와 이러한 정보를 런타임 시 어떻게 찾는지를 이해할 필요가 있겠다. BlockClosure에 정의된 중심 메서드 ensure: 의 정의를 살펴보자.

```
ensure: aBlock
    ”Evaluate a termination block after evaluating the receiver, regardless of
        whether the receiver’s evaluation completes. N.B. This method is*not*
        implemented as a primitive. Primitive 198 always fails. The VM uses prim
        198 in a context’s method as the mark for an ensure:/ifCurtailed: activation.”

    | complete returnValue |
    <primitive: 198>
    returnValue := self valueNoContextSwitch.
    complete ifNil: [
        complete := true.
        aBlock value ].
    ^ returnValue
```

<primitive: 198 > 는 앞 절에서 살펴본 <primitive: 199 > 와 같은 방식으로 작동한다. 이는 항상 실패하지만 이것이 존재할 경우 실행하는 컨텍스트를 유일한 방식으로 표시한다. 게다가 unwind 블록은 예외 클래스 및 그에 연관된 핸들러와 동일한 방식으로 보관된다. 좀 더 분명히 말하자면 ensure: 메서드 실행의 컨텍스트에 (스택 프레임) 보관되는데, thisContext sender tempAt: 1을 통해 블록으로부터 접근이 가능하다.

블록이 실패하지 않고 non-local 리턴을 갖지 않은 경우, ensure: 메시지 구현은 매우 이해하기 쉽다. 메시지는 블록을 평가하고, returnValue 변수 내에 결과를 저장하며, 인자 블록을 평가하고, 마지막으로 이전에 저장된 블록의 결과를 리턴한다. complete 변수는 인자 블록이 두 번 실행되는 것을 피하기 위해 존재할 뿐이다.

실패 블록 확보하기. ensure: 메시지는 블록이 실패하더라도 인자 블록을 실행할 것이다. 아래 예제에서 ensureWithOnDo 메시지는 2를 리턴하고 1을 실행한다. 그 다음 절에서 우리는 블록이 어디에서 무엇을 실제로 리턴하는지를 비롯해 블록이 어떤 순서로 실행되는지를 주의 깊게 살펴볼 것이다.

```
Bexp>>ensureWithOnDo
    ^[ [ Error signal ] ensure: [ 1 ].
        ^3 ] on: Error do: [ 2 ]
```

구현을 살펴보기 전에 간략한 예를 들어보자. 실패 Block을 확보하는 4개의 블록과 1개의 메서드를 정의한다.

```
Bexp>>mainBlock
    ^[ self traceCr: 'mainBlock start'.
    self failingBlock ensure: self ensureBlock.
    self traceCr: 'mainBlock end' ]

Bexp>>failingBlock
    ^[ self traceCr: 'failingBlock start'.
    Error signal.
    self traceCr: 'failingBlock end' ]

Bexp>>ensureBlock
    ^[ self traceCr: 'ensureBlock value'.
    #EnsureBlockValue ]

Bexp>>exceptionHandlerBlock
    ^[ self traceCr: 'exceptionHandlerBlock value'.
    #ExceptionHandlerBlockValue ]

Bexp>>start
    | res |
    self traceCr: 'start start'.
    res := self mainBlock on: Error do: self exceptionHandlerBlock.
    self traceCr: 'start end'.
    self traceCr: 'The result is : ', res, '.'.
    ^ res
```

Bexp new start 를 실행하면 아래가 출력된다 (호출 흐름을 강조하기 위해 들여쓰기를 적용하였다).

```
start start
    mainBlock start
        failingBlock start
            exceptionHandlerBlock value
            ensureBlock value
start end
The result is: ExceptionHandlerBlockValue.
```

세 가지 사항이 중요하다. 첫째, 실패하는 블록과 메인 블록은 signal 메시지 때문에 완전히

실행되지 않는다. 둘째, 실행 블록이 ensure 블록 이전에 실행된다. 마지막으로, start 메서드는 예외 핸들러 블록의 결과를 리턴할 것이다.

이것이 어떻게 작용하는지 이해하기 위해 예외 구현의 끝 부분을 살펴봐야겠다. handleSignal 메서드에 관한 이전 설명을 끝마친다.

```
ContextPart>>handleSignal: exception
    ”Sent to handler (on:do:) contexts only. If my exception class (first arg) handles
        exception then  execute my handle block (second arg), otherwise forward this
       message to the next handler context. If none left, execute exception’s defaultAction
        (see nil>>handleSignal:).”

    | val |
    (((self tempAt: 1) handles: exception) and: [self tempAt: 3]) ifFalse: [
        ^ self nextHandlerContext handleSignal: exception].

    exception privHandlerContext: self contextTag.
    self tempAt: 3 put: false. ”disable self while executing handle block”
    val := [(self tempAt: 2) valueWithPossibleArgs: {exception}]
        ensure: [self tempAt: 3 put: true].
    self return: val. ”return from self if not otherwise directed in handle block”
```

우리가 제시한 예제라면 Pharo는 실패하는 블록을 실행한 후 다음 <primitive: 199 >로 표시된 핸들러 컨텍스트를 검색할 것이다. 정규적 예외(regular exception)로서 Pharo는 예외 핸들러 컨텍스트를 찾고 exceptionHandlerBlock을 실행한다. handleSignal 메서드는 return: 메서드를 이용해 완료되는데, 아래를 통해 살펴보자.

```
ContextPart>>return: value
    ”Unwind thisContext to self and return value to self’s sender. Execute any unwind
        blocks while unwinding. ASSUMES self is a sender of thisContext”

    sender ifNil: [self cannotReturn: value to: sender].
    sender resume: value
```

return: 메시지는 컨텍스트가 전송자를 갖고 있는지 확인할 것이며, 전송자가 없을 경우 Cannotreturn Exception을 전송할 것이다. 이후 해당 컨텍스트의 전송자는 resume: 메시지를 호출할 것이다.

```
ContextPart>>resume: value
    ”Unwind thisContext to self and resume with value as result of last send. Execute
        unwind blocks when unwinding. ASSUMES self is a sender of thisContext”

    | ctxt unwindBlock |
    self isDead ifTrue: [self cannotReturn: value to: self].
    ctxt := thisContext.
    [ ctxt := ctxt findNextUnwindContextUpTo: self.
        ctxt isNil
    ] whileFalse: [
        (ctxt tempAt: 2) ifNil:[
            ctxt tempAt: 2 put: true.
            unwindBlock := ctxt tempAt: 1.
            thisContext terminateTo: ctxt.
            unwindBlock value].
    ].
    thisContext terminateTo: self.
    ^ value
```

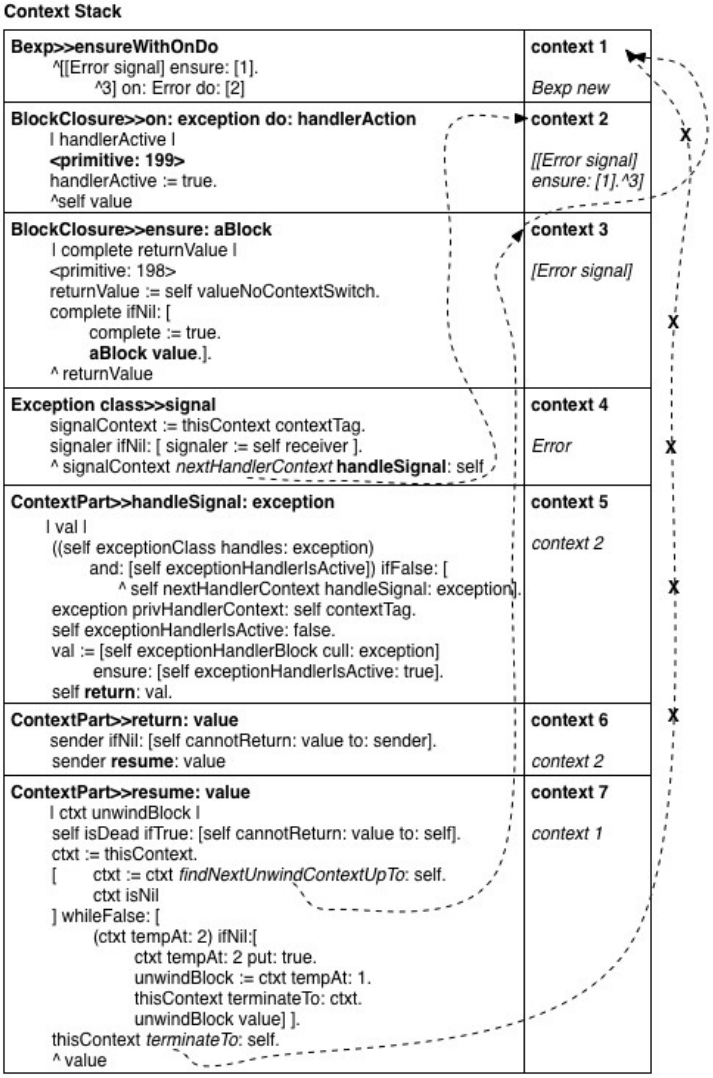


그림 13.6: 컨텍스트 스택.

이는 ensure: 의 인자 블록이 실행되는 메서드이다. 해당 메서드는 resume: 메서드의 컨텍스트와 on:do: 컨텍스트(본문 예제에서는 start의 컨텍스트)의 전송자인 self사이의 모든 unwind 컨텍스트를 검색한다. 메서드가 언와인딩된 컨텍스트를 찾으면 unwound 블록이 실행된다. 마지막으로 terminateTo: 메시지를 트리거한다.

```
ContextPart>>terminateTo: previousContext
    ”Terminate all the Contexts between me and previousContext, if previousContext is on
        my Context stack. Make previousContext my sender.”

    | currentContext sendingContext |
    <primitive: 196>
    (self hasSender: previousContext) ifTrue: [
        currentContext := sender.
        [currentContext == previousContext] whileFalse: [
            sendingContext := currentContext sender.
            currentContext terminate.
            currentContext := sendingContext]].
    sender := previousContext
```

기본적으로 해당 메서드는 on:do: 컨텍스트(본문 예제에서는 start의 컨텍스트)의 전송자인 self와 thisContext 사이의 모든 컨텍스트를 종료한다. 뿐만 아니라 thisContext의 전송자가 on:do: 컨텍스트(본문 예제에서는 start의 컨텍스트)의 전송자인 self가 될 것이다. 이것은 프리미티브로서 구현되지만 어떻게 작동하는지는 아래 코드에서 설명한다.

앞서 정의된 ensureWithOnDo 메서드의 실행을 보여주는 그림 13.6에서 무슨 일이 발생하는지 간략하게 살펴보자.

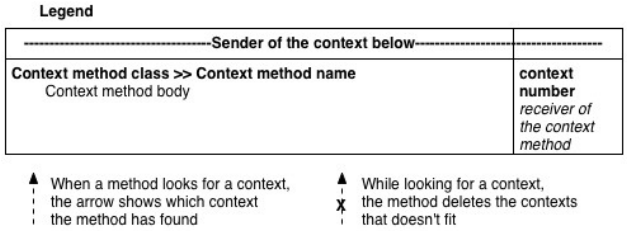


그림 13.7: 그림 범례.

- 왼쪽 코멘트 : 메서드가 컨텍스트를 검색할 때 화살표는 메서드가 발견된 컨텍스트를 표시한다.
- 오른쪽 코멘트 : 컨텍스트를 검색하는 동안 메서드는 일치하지 않는 컨텍스트를 삭제한다.

non-local 리턴 확보하기. non-local 리턴에 대한 ensure: 의 구현도 존재한다. 기본적으로는 언와인딩된 컨텍스트의 검색은 resume:through: 내 값을 리턴할 때 트리거되는 resume: 메시지에서와 동일한 유형이다.

```
ContextPart>>resume: value through: firstUnwindCtxt
    "Unwind thisContext to self and resume with value as result of last send.
        Execute any unwind blocks while unwinding. ASSUMES self is a sender of
        thisContext."

    | ctxt unwindBlock |
    self isDead ifTrue: [self cannotReturn: value to: self].
    ctxt := firstUnwindCtxt.
        [ctxt isNil] whileFalse:
            [(ctxt tempAt: 2) ifNil:
                [ctxt tempAt: 2 put: true.
                    unwindBlock := ctxt tempAt: 1.
                    thisContext terminateTo: ctxt.
                    unwindBlock value].
        ctxt := ctxt findNextUnwindContextUpTo: self].
    thisContext terminateTo: self.
    ^value
```

요약

본 장에서는 시그널링해야 하는 예외를 사용하는 방법과 코드에서 발생하는 비정상적 상황을 처리하는 방법을 살펴보았다.

- 예외를 제어-흐름 메커니즘으로 사용하지 말라. 비정상적 상황과 알림용으로 남겨두어라. 예외를 시그널링하기 위한 방법으로, 블록을 인자로서 취하는 메서드를 제공하는 방도를 고려해보라.
- protectedBlock이 비정상적으로 종료되더라도 actionBlock이 실행되도록 확보하기 위해서는 protecBlock ensure: actionBlock 을 사용하라.
- protectedBlock이 비정상적으로 종료될 때에만 actionBlock이 실행되도록 확보하기 위해서는 protecBlock ifCurtailed: actionBlock 을 사용하라.
- 예외는 객체다. 예외 클래스들은 계층구조를 형성하고, 그 계층구조 루트에는 Exception 클래스가 위치한다.
- ExceptionClass(또는 그 서브클래스들 중 하나)의 인스턴스에 해당하는 예외를 포착하기 위해서는 protectedBlock on: ExceptionClass do: handlerBlock 를 사용하라. handlerBlock 은 예외 인스턴스를 유일한 인자로서 취해야 한다.
- 예외는 signal 이나 signal: 메시지 중 하나를 전송함으로써 시그널링된다. signal: 은 그 인자로 설명적 문자열을 취한다. 예외의 설명은 예외로 description을 전송하면 얻을 수 있다.
- 자신의 코드에 메시지 전송 self halt를 삽입하면 중단점(breakpoint)을 설정할 수 있다. 이는 재개 가능한 Halt 예외를 시그널링하며, 기본적으로 이는 중단점이 발생하는 지점에서 디버거를 열 것이다.

- 예외가 시그널링되면 런타임 시스템이 특정 예외 클래스에 대한 핸들러를 실행 스택에서 검색할 것이다. 어떤 핸들러도 발견되지 않을 경우 해당 예외에 대한 defaultAction이 실행될 것이다 (예 : 대부분의 경우 디버거가 열릴 것이다).
- 예외 핸들러는 시그널링된 예외로 return: 을 전송하여 보호 블록을 종료할 수 있다. 보호 블록의 값은 return: 으로 제공되는 인자가 될 것이다.
- 예외 핸들러는 시그널링된 예외로 retry를 제공하여 보호 블록을 재시도할 수도 있다. 핸들러는 계속 유효하다.
- 예외 핸들러는 시그널링된 예외로 새 블록을 인자로 하여 retryUsing: 를 전송하여 새 블록을 명시할 수 있다. 이 또한 핸들러는 계속 유효하다.
- Notifications는 핸들러가 특정 액션을 취하지 않고도 안전하게 재개할 수 있는 프로퍼티를 가진 Exception의 서브클래스이다.

감사의 말. 새로운 내용을 제공해준 Vassili Bykov에게 감사드린다. ensure: 와 ifCurtailed: 의 스몰토크 구현을 제공해준 Paolo Bonzini에게도 감사의 마음을 전한다. 또 코멘트와 여러 의견을 제공해준 Hernan Wilkinson, Lukas Renggli, Christopher Oliver, Camillo Bruni, Hernan Wilkinson, Carlos Ferro께도 감사의 말을 전하고 싶다.

제 14 장

블록: 세부적인 분석

***Clément Bera** 참여 (bera.clement@gmail.com)*

어휘적 유효 범위(lexically-scoped)만 가진 블록 클로저(block closures), 짧게 말해 블록이라는 것은 강력하고 필수적인 스몰토크 기능이다. 블록이 없이는 작고 간편한 구문을 갖기가 힘들 것이다. 스몰토크에서 블록의 사용은 조건문과 루프를 라이브러리 메시지로서 얻어 언어 구문으로 하드코딩하지 않는 데에 핵심이 된다. 이것이 바로 블록이 스몰토크의 구문을 전달하는 메시지와 절대적으로 잘 작용한다고 말할 수 있는 이유다.

뿐만 아니라 블록은 가독성, 재사용 가능성, 코드의 효율성을 향상시키는 데에도 효과적이다. 하지만 스몰토크의 양호한 동적 런타임 구문은 잘 문서화되지 않는다. 예를 들어, 리턴 문이 있을 때 블록은 escaping 메커니즘과 같이 행동하지만 극단적으로 사용하면 보기 싫은 코드를 야기하므로 이해하는 것이 중요하다.

이번 장에서는 블록 생성 시 변수의 capture와 정의 환경에 대한 중심 개념을 학습할 것이다. 블록이 프로그램 흐름을 어떻게 변경시키는지도 학습할 것이다. 마지막으로 블록을 이해하기 위해 우리는 프로그램이 어떻게 실행하는지, 또 주어진 실행 상태를 나타내는 컨텍스트, 소위 활성 레코드라는 것을 어떻게 표현하는지 설명하고자 한다. 블록 실행 도중에 컨텍스트가 어떻게 사용되는지 보일 것이다. 본 장은 예외(제 13장)에 관한 장을 보충한다. Pharo by Example 저서에서는 블록을 작성하고 사용하는 방법을 제시하였다. 반대로 이번 장은 블록의 좀 더 깊은 측면과 그들의 런타임 행위에 초점을 두겠다.

기본

블록이란 뭘까? 블록은 생성 시 그 환경을 포착(또는 뒤덮는)하는 람다(lambda) 표현식이다. 이것이 정확히 어떤 의미인지는 후에 살펴보겠다. 블록은 익명 함수로서 인식할 수도 있겠다. 블록은 코드 조각으로서, 그 평가는 freeze되어 메시지 사용에 효과가 나타난다. 블록은 사각 괄호로 정의된다.

아래 코드를 실행하고 결과를 출력하면 3이 아니라 하나의 블록을 얻을 것이다. 사실 당신은 블록 값을 요청한 것이 아니라 블록 자체를 요청한 것이므로 블록을 얻게 된다.

```
[1 + 2] → [1 + 2]
```

블록으로 value 메시지를 전송하면 블록이 평가된다. 좀 더 정밀한 블록은 value (의무적으로 취해야 하는 인자가 없는 경우), value: (블록이 하나의 인자를 필요로 하는 경우), value:value (2개의 인자), value:value:value: (3개의 인자), valueWithArguments:anArray (4개 이상의 인자)를 사용해 평가된다. 이러한 메시지들은 블록 평가에 오래 사용된 기본적인 API이다. 이는 Pharo by Example 서적에서 제시되어 있다.

```
[1 + 2] value → 3

[:x | x + 2] value: 5 → 7
```

몇 가지 간단한 확장(extension)

value 메시지 외에도 Pharo는 cull: 과 같은 유형의 간편한 메시지를 비롯하여 필요 이상의 값이 있을 경우에도 블록의 평가를 지원한다. cull: 은 만일 수신자가 제공되는 것 이상의 인자를 필요로 하는 경우 오류를 발생시킬 것이다. ValueWithPossibleArgs: 메시지는 cull: 과 비슷하지만 블록을 인자로서 전달하기 위해 매개변수의 배열을 취한다. 블록이 제공된 것 이상의 인자를 필요로 할 경우 valueWithPossibleArgs: 는 그것을 nil로 채울 것이다.

```
[ 1 + 2 ] cull: 5 → 3
[ 1 + 2 ] cull: 5 cull: 6 → 3
[ :x | 2 + x ] cull: 5 → 7
[ :x | 2 + x ] cull: 5 cull: 3 → 7
[ :x :y | 1 + x + y ] cull: 5 cull: 2 → 8
[ :x :y | 1 + x + y ] cull: 5    error because the block needs 2 arguments.
[ :x :y | 1 + x + y ] valueWithPossibleArgs: #(5)
     error because 'y' is nil and '+' does not accept nil as a parameter.
```

그 외 메시지들. 평가의 프로파일링에 유용한 메시지도 몇 가지가 있다 (상세한 정보는 제 17장을 참고).

bench. 5초 이내에 수신자 블록을 평가할 수 있는 횟수를 리턴한다.

durationToRun. 수신자 블록을 평가하는 데에 들인 시간을 (Duration의 인스턴스) 응답한다.

timeToRun. 해당 블록을 평가하는 데에 들인 시간을 밀리초로 응답한다.

일부 메시지는 오류 처리와 관련된다 (제 13장에서 설명한 바와 같이).

ensure: terminationBlock. 수신자의 평가가 완료되었는지와 상관없이 수신자를 평가한 후 종료(termination) 블록을 평가한다.

ifCurtailed: onErrorBlock. 수신자를 평가하며, 평가가 완료되지 않은 경우 오류 블록을 평가한다. 수신자의 평가가 정상적으로 완료되면 오류 블록은 평가되지 않는다.

on: exception do: catchBlock. 수신자를 평가한다. exception 예외가 발생하면 catch 블록을 평가한다.

on: exception fork: catchBlock. 수신자를 평가한다. exception 예외가 발생하면 오류를 처리하게 될 새로운 프로세스를 fork한다. 원래 프로세스는 마치 수신자 평가가 끝나고 nil을 응답할 때와 마찬가지로 계속해서 실행될 것인데, 가령 [self error: 'some error'] on: Error fork: [:ex | 123] 와 같은 표현식은 원본 프로세스에게 nil을 응답할 것이다. 컨텍스트 스택, 즉 해당 메시지를 수신자로 전송한 컨텍스트부터 시작해 스택의 최상위까지가 forked 프로세스로 전송될 것이며, catch 블록은 최상단에 위치할 것이다. 마지막으로 catch 블록은 forked 프로세스에서 평가될 것이다.

일부 메시지들은 프로세스 스케줄링과 연관된다. 본문에서는 가장 중요한 메시지만 열거하겠다. 본 장은 Pharo에서 동시적 프로그래밍에 관한 내용이 아니므로 깊이 다루지 않을 것이다.

fork. 수신자를 평가하는 Process를 생성하고 schedule한다.

forkAt: aPriority. 주어진 우선순위에서 수신자를 평가하는 Process를 생성하고 schedule한다. 새로 생성된 프로세스를 응답한다.

newProcess. 수신자를 평가하는 Process를 응답한다. 프로세스는 schedule되지 않는다.

변수와 블록

블록은 고유의 임시 변수를 가질 수 있다. 그러한 변수들은 각 블록 평가 도중에 초기화되며 블록에 국부적이다. 그러한 변수가 어떻게 유지되는지는 후에 살펴볼 것이다. 지금은 블록이 다른 (non-local) 변수들을 참조할 때 어떤 일이 일어나는지를 분명히 하는 것이 문제다. 블록은 그것이 사용하는 외부 변수를 둘러쌀 것이다. 즉, 블록이 사용하는 변수를 어휘적으로 포함하지 않는 환경에서 블록을 후에 실행하더라도 블록은 여전히 실행 도중에 변수로 접근성을 가질 것을 의미한다. 지역 변수를 구현하고 컨텍스트를 이용해 보관하는 방법은 후에 제시하겠다.

Pharo에서 private 변수들은 (self, 인스턴스 변수, 메서드 임시 변수와 인자) 어휘적 유효 범위만 가진다 (즉, 메서드 내 표현식은 가령 클래스의 인스턴스 변수로 접근할 수 있으나 다른 클래스 내의 동일한 표현식은 동일한 변수로 접근할 수 없다). 런타임 시 이러한 변수들은 블록이 평가되는 컨텍스트가 아니라 변수들을 포함하는 블록이 정의된 컨텍스트에서 바인딩된다 (그들과 연관된 값을 얻는다). 즉, 블록은 다른 곳에서 평가될 경우 블록이 생성될 당시 그 범위 내에서 (블록에게 표시) 변수로 접근할 수 있다. 관습상 블록이 정의되는 컨텍스트는 블록 홈 컨텍스트 (block home context) 라고 명명된다.

블록 홈 컨텍스트란 특정 실행 포인트를 나타내므로 (애초에 블록을 생성한 프로그램 실행이므로) 블록 홈 컨텍스트라는 개념은 프로그램 실행을 표현하는 객체에 의해 표현되는데, 스몰토크에서는 컨텍스트에 해당하겠다. 본질적으로 컨텍스트 (다른 언어로 스택 프레임 또는 활성 레코드라 불리는) 는 현재 평가 단계가 실행되는 컨텍스트, 다음으로 실행될 바이트 코드, 임시 변수의 값과 같이 현재 평가 단계에 관한 정보를 나타낸다. 컨텍스트는 스몰토크 실행 스택 요소를 표현하는 활성 레코드이다. 이는 중요한 내용으로, 해당 개념은 후에 다시 살펴볼 것이다.

> 블록은 어떤 컨텍스트 (실행 지점을 표현하는 객체) 내부에서 생성된다.

몇 가지 작은 실험

변수가 블록에서 어떻게 바인딩되는지 이해하기 위해 실험을 약간 해보도록 하자. Bexp(BlockExperiment)로 명명된 클래스를 정의한다.

```
Object subclass: #Bexp
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'BlockExperiment'
```

실험 1: 변수 검색. 변수는 블록 정의 컨텍스트에서 검색된다. 우리는 두 개의 메서드를 정의하는데, 하나는 변수 t를 정의하여 42로 설정하는 메서드, 나머지 하나는 다른 곳에서 정의된 블록을 실행하는 새 변수를 정의한다.

```
Bexp>>setVariableAndDefineBlock
    | t |
    t := 42.
    self evaluateBlock: [ t traceCr ]

Bexp>>evaluateBlock: aBlock
    | t |
    t := nil.
    aBlock value

Bexp new setVariableAndDefineBlock
    → 42
```

Bexp new setVariableAndDefineBlock 표현식을 실행하면 Transcript에 (메시지 raceCr) 42를 출력한다. 블록이 메서드 실행 도중에 평가된다 하더라도 evaluateBlock: 메서드에 정의된 것보다는 setVariableAndDefineBlock 메서드에 정의된 임시 변수 t 의 값이 사용된다. 변수 t는 블록 생성 컨텍스트에서 검색된다 (컨텍스트는 블록 평가 메서드 evaluateBlock: 의 컨텍스트가 아니라 setVariableAndDefineBlock 메서드의 실행 도중에 생성).

이제 세부적으로 살펴보자. 그림 14.1은 Bexp new setVariableAndDefineBlock 표현식의 실행을 보여준다.

- setVariableAndDefineBlock 메서드의 실행 중 t 변수가 정의되고 42가 할당된다. 이후 블록이 생성되며, 이 블록은 임시 변수(단계 1)를 보유하는 메서드 활성 컨텍스트를 참조한다.
- EvaluteBlock: 메서드는 블록에 있는 것과 같은 이름으로 된 고유의 지역 변수 t를 정의한다. 하지만 이는 블록이 평가될 당시 사용되는 변수는 아니다. EvaluateBlock: 메서드를 실행하는 동안 블록이 평가되고 (단계 2), t traceCr 표현식이 실행되는 동안 non-local 변수 t가 블록의 홈 컨텍스트에서 검색되는데, 이는 동시에 실행되는 메서드의 컨텍스트가 아니라 블록을 생성한 메서드 컨텍스트에 해당하겠다.

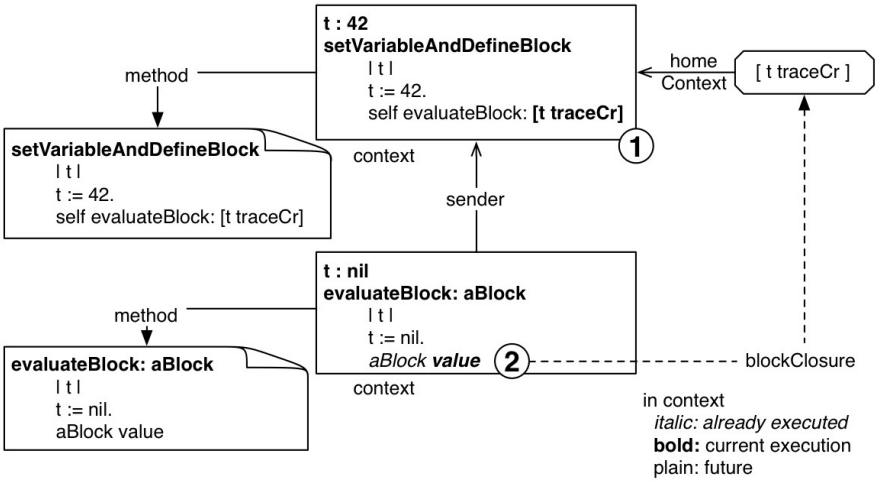


그림 14.1: 블록이 평가된 곳이 아니라 블록이 생성된 메서드 활성 컨텍스트에서 non-local 변수가 검색된다.

> Non-local 변수들은 블록을 실행하는 컨텍스트가 아니라 블록의 홈 컨텍스트(예: 블록을 생성한 메서드 컨텍스트)에서 검색된다.

실험 2: 변수 값 변경하기. 우리의 실험을 계속해보자. SetVariableAndDefineBlock2 메서드는 블록의 평가 도중에 non-local 변수 값을 변경할 수 있음을 보여준다. Bexp new setVariableAndDefineBlock2 를 실행하면 33을 출력하는데, 33은 변수 t의 마지막 값이기 때문이다.

```
Bexp>>setVariableAndDefineBlock2
    | t |
    t := 42.
    self evaluateBlock: [ t := 33. t traceCr ]

Bexp new setVariableAndDefineBlock2
    → 33
```

실험 3: 공유 non-local 변수로 접근하기. 두 개의 블록이 non-local 변수를 공유할 수 있고, 서로 다른 순간에 해당 변수의 값을 수정할 수 있다. 이를 확인하려면 아래와 같이 새 메서드 setVariableAndDefineBlock3을 정의해보자.

```
Bexp>>setVariableAndDefineBlock3
    | t |
    t := 42.
    self evaluateBlock: [ t traceCr. t := 33. t traceCr ].
    self evaluateBlock: [ t traceCr. t := 66. t traceCr ].
    self evaluateBlock: [ t traceCr ]

Bexp new setVariableAndDefineBlock3
    → 42
    → 33
    → 33
    → 66
    → 66
```

Bexp new setVariableAndDefineBlock3 는 42, 33, 33, 66, 66를 출력할 것이다. 여기서 두 개의 블록 [t := 33. t traceCr] 와 [t := 66. t traceCr] 는 동일한 변수 t로 접근 및 수정할 수 있다. EvaluateBlock: 메서드가 첫 번째로 실행되는 동안 그 현재 값인 42가 출력되고, 이후에 값이 변경 및 출력된다. 두 번째 호출에도 비슷한 상황이 발생한다. 이 예제는 변수가 보관된 위치를 블록들이 공유하며, 블록은 capture된 변수의 값을 복사하지 않음을 보여준다. 그저 변수의 위치를 참조하는데, 여러 개의 블록이 동일한 위치를 참조할 수 있다.

실험 4: 변수 검색은 실행 시간에 이루어진다. 아래 예제는 변수의 값이 런타임 시 검색되며, 블록 생성 중에 복사되지 않음을 보여준다. 먼저 Bexp 클래스에 인스턴스 변수 block 을 추가한다.

```
Object subclass: #Bexp
    instanceVariableNames: 'block'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'BlockExperiment'
```

여기서 변수 t의 초기 값은 42다. 블록은 생성된 후 인스턴스 변수 block에 보관되지만 t에 대한 값은 블록이 평가되기 전에 69로 변경된다. 그리고 이것은 효과적으로 출력되는 마지막 값 (69) 인데, 그 이유는 실행 시에 검색되기 때문이다. Bexp new setVariableAndDefineBlock4 을 실행하면 69가 출력된다.

```
Bexp>>setVariableAndDefineBlock4
    | t |
    t := 42.
    block := [ t traceCr: t ].
    t := 69.
    self evaluateBlock: block

Bexp new setVariableAndDefineBlock4
    → 69.
```

실험 5: 메서드 인자에 관해. 자연적으로 우리는 메서드 인자는 정의하는 메서드의 컨텍스트에 바인딩될 것으로 예상할 것이다. 이 점을 설명해보자. 아래 메서드를 정의하라.

```
Bexp>>testArg
    self testArg: 'foo'.

Bexp>>testArg: arg
    block := [arg crLog].
    self evaluateBlockAndIgnoreArgument: 'zork'.

Bexp>>evaluateBlockAndIgnoreArgument: arg
    block value.
```

이제 Bexp new testArg: 'foo' 를 실행하면 'foo' 를 출력하며, evaluateBlockAndIgnore-Argument: 메서드에서 임시 arg가 재정의되더라도 마찬가지다.

실험 6: 자체 바인딩(self binding). 이제 self도 capture되는지가 궁금할 수 있다. 이를 테스트하기 위해선 다른 클래스가 필요하다. 간단히 새 클래스와 몇 개의 메서드를 정의해보자. 인스턴스 변수 x를 클래스 Bexp 로 추가하고 아래와 같이 initialize 메서드를 정의해보자.

```
Object subclass: #Bexp
    instanceVariableNames: 'block x'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'BlockExperiment'

Bexp>>initialize
    super initialize.
    x := 123.
```

Bexp2라는 클래스를 하나 더 정의하라.

```
Object subclass: #Bexp2
    instanceVariableNames: 'x'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'BlockExperiment'

Bexp2>>initialize
    super initialize.
    x := 69.

Bexp2>>evaluateBlock: aBlock
    aBlock value
```

다음으로 Bexp2에 정의된 메서드를 호출하게 될 메서드를 정의하라.

```
Bexp>>evaluateBlock: aBlock
    Bexp2 new evaluateBlock: aBlock

Bexp>>evaluateBlock
    self evaluateBlock: [self crTrace ; traceCr: x]

Bexp new evaluateBlock
    → a Bexp123 "and not a Bexp269"
```

이제 Bexp new evaluateBlock 를 실행하면 Transcript에Bexp123 이 출력되면서 블록이 self 또한 capture하였음을 표시하는데, Bexp2의 인스턴스가 블록을 평가하였지만 출력된 객

체(self)는 블록 생성 시 접근 가능한 원본 Bexp 인스턴스이기 때문이다.

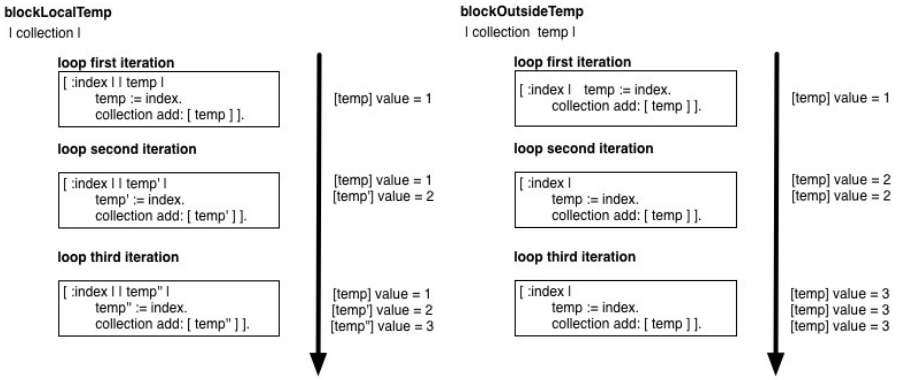
결론. 블록이 실행되는 컨텍스트가 아니라 블록이 정의되는 컨텍스트에서 접근하는 변수를 블록이 capture함을 보여준다. 블록은 다수의 블록이 공유 가능한 변수 위치로 참조를 유지한다.

Block-local 변수

앞서 살펴보았듯 블록은 그것이 정의된 장소에 연결된 어휘적 클로저다. 다음으로는 블록 지역 변수가 그들의 생성에 대한 실행 컨텍스트 연계(link)에서 할당된다는 사실을 보임으로써 이러한 연결을 설명할 것이다. 변수가 블록에 국부적(local)일 때와 메서드에 국부적일 때 차이를 설명하겠다 (그림 14.2 참고).

블록 할당. 아래의 blockLocalTemp 메서드를 구현하라.

```
Bexp>>blockLocalTemp
    | collection |
    collection := OrderedCollection new.
    #(1 2 3) do: [ :index |
        | temp |
        temp := index.
        collection add: [ temp ] ].
    ^ collection collect: [ :each | each value ]
```



(a) blockLocalTemp execution (b) blockOutsideTemp execution

그림 14.2: blockXX execution

코드에 대해 언급해보자. 우리는 루프에서 생성된 임시 변수 temp로 현재 색인을 보관하는 루프를 생성한다. 이후 이 변수로 접근하는 블록을 컬렉션에 보관한다. 루프 다음에 우리는 접근하는 블록마다 평가하고 값의 컬렉션을 리턴한다. 해당 메서드를 실행하면 1, 2, 3이 있는 컬렉션을 얻는다. 이러한 결과는 컬렉션 내 각 블록이 서로 다른 temp 변수를 참조함을 보여준다. 이는 각 블록 생성(각 루프 단계에서)마다 실행 컨텍스트가 생성되고 블록[temp]이 이러한

컨텍스트를 참조한다는 사실 때문에 발생하는 것이다.

메서드 할당. temp가 블록 변수가 아니라 메서드 변수라는 것만 제외하고 BlockLocalTemp와 동일한 새 메서드를 생성하라.

```
Bexp>>blockOutsideTemp
    | collection temp |
    collection := OrderedCollection new.
    #(1 2 3) do: [ :index |
        temp := index.
        collection add: [ temp ] ].
    ^ collection collect: [ :each | each value ]
```

blockOutsideTemp 메서드를 실행하면 이제 3, 3, 3이 있는 컬렉션을 얻는다. 이러한 결과는 컬렉션 내 각 블록이 이제는 blockOutsideTemp 컨텍스트에 할당된 단일 변수 temp를 참조하여 temp가 블록들에 의해 공유되는 사실로 이어짐을 보여준다.

변수는 그들을 정의한 메서드보다 오래 생존할 수 있다

블록에 의해 참조되는 non-block 지역 변수는 메서드 실행이 종료되더라도 다른 표현식에서 접근 및 공유할 수 있다. 이것은 변수가 그들을 정의한 메서드 실행보다 오래 생존한다고 말한다. 예를 몇 가지 들어보겠다.

메서드와 블록 간 공유. 메서드와 블록 간에 변수가 공유됨을 보여주는 (앞의 실험과 마찬가지로) 간단한 예로 시작하겠다. 임시 변수 a를 정의하는 메서드 foo를 정의해보자.

```
Bexp>>foo
    | a |
    [ a := 0 ] value.
    ^ a

Bexp new foo
    → 0
```

Bexp new foo를 실행하면 nil이 아니라 0을 얻는다. 여기서 보이는 값은 메서드 body와 블록에서 공유하는 값이다. 메서드 body 내부에서 우리는 블록 평가에 의해 값이 설정된 변수로 접근이 가능하다. 메서드 body와 블록 body 모두 동일한 임시 변수 a로 접근한다.

조금만 더 복잡하게 만들어보자. 아래와 같이 twoBlockArray 메서드를 정의하라.

```
Bexp>>twoBlockArray
    | a |
    a := 0.
    ^ {[ a := 2] . [a]}
```

twoBlockArray 메서드가 임시 변수 a를 정의한다. 또한 a의 값을 0으로 설정하고, a의 값을 2로 설정하는 블록을 첫 번째 요소로 하고, 임시 변수 a의 값을 리턴하기만 하는 블록을 두 번째 요소로 하는 배열을 리턴한다.

이제 twoBlockArray가 리턴하는 배열을 보관하고, 배열에 보관된 블록을 평가한다. 이는 아래 코드를 통해 이루어진다.

```
| res |
res := Bexp new twoBlockArray.
res second value. → 0
res first value.
res second value. → 2
```

아니면 아래와 같이 코드를 정의하고 transcript를 열어 결과를 확인하는 방법도 있다.

```
| res |
res := Bexp new twoBlockArray.
res second value traceCr.
res first value.
res second value traceCr.
```

잠시 물러나서 중요한 요점을 살펴보자. 앞의 코드 조각에서는 res second value 과 res first value 표현식이 실행되면 twoBlockArray 메서드는 이미 실행을 끝낸 상태로 더 이상 실행 스택에 남아 있지 않다. 여전히 임시 변수 a로 접근하여 새 값으로 설정할 수는 있다. 이러한 실험을 통해 우리는 블록에 의해 참조되는 변수가 그들을 참조하는 블록을 생성한 메서드보다 오래 생존할 수 있다는 점이다. 따라서 변수가 그들을 정의하는 메서드의 실행보다 오래 생존한다고 말한다.

이번 예제에서 임시 변수는 어떻게든 활성 컨텍스트에 보관되는데 구현은 그보다 좀 더 미묘하다는 사실을 확인할 수 있다. 블록 구현은 실행 스택에 있지 않고 heap 에서 상주하는 구조에 참조(referenced) 변수를 유지할 필요가 있다.

블록 내부로부터 리턴하기

이번 절에서는 당신이 인스턴스 변수로 전달하거나 인스턴스 변수에 보관한 블록 내부에 (예: [^ 33]) 리턴 문을 갖는 것이 왜 바람직하지 않은지를 설명하겠다. 명시적 리턴 문이 있는 블록을 non-local returning block이라 부른다. 몇 가지 기본 요점을 먼저 설명하겠다.

리턴에 관한 기본 내용

기본적으로 리턴된 메서드의 값은 메시지의 수신자에 해당하는데 가령 self를 예로 들 수 있겠다. 리턴 표현식(문자 ^로 시작되는 표현식)은 메시지의 수신자가 아닌 값도 리턴하도록 허용한다. 뿐만 아니라 리턴 문의 실행은 현재 실행된 메서드를 나가서(exit) 그 호출자에게 리턴한다. 이는 리턴 문 다음에 오는 표현식을 무시한다.

실험 7: 리턴의 기존 행위. 아래 메서드를 정의하라. Bexp new testExplicitReturn 을 실행하면 'one' 과 'two'를 출력하지만 not printed는 출력하지 않을 것인데, testExplicitReturn 메서드가 이전에 리턴했을 것이기 때문이다.

```
Bexp>>testExplicitReturn
    self traceCr: 'one'.
    0 isZero ifTrue: [ self traceCr: 'two'. ^ self].
    self traceCr: 'not printed'
```

리턴 표현식은 블록 body에서 마지막 문이어야 함을 주목하라.

non-local 리턴의 행위 escape하기

표현식 흐름은 현재 호출하는 메서드로 곧바로 뛰어들 것이기 때문에 리턴 표현식은 escaping 메커니즘과도 같다. 이러한 행위를 설명하기 위해 아래와 같이 jumpingOut 이라는 새 메서드를 정의해보자.

```
Bexp>>jumpingOut
    #(1 2 3 4) do: [:each |
        self traceCr: each printString.
        each = 3
            ifTrue: [^ 3]].
    ^ 42

Bexp new jumpingOut
    → 3
```

예를 들어 표현식 Bexp new jumpingOut은 42가 아니라 3을 리턴할 것이다. ^42 는 절대 도달되지 않을 것이다. 표현식 [^3]은 깊이 중첩(nested)될 수 있으며, 그 실행은 모든 수준을 빠져나와(jump out) 메서드 호출자에게 리턴한다. 일부 오래된 코드(앞의 예외를 소개한 바 있음)는 non-local returning 블록을 전달하여 복잡한 흐름을 야기하고 코드의 유지가 힘들게 된다. 이러한 스타일은 사용하지 않을 것을 권하는데, 복잡한 코드와 버그로 이어지기 때문이다. 다음 절에서는 return이 실제로 리턴하는 경우를 주의 깊게 살펴볼 것이다.

리턴 이해하기

리턴이 사실상 현재 실행을 escape 하는지를 확인하기 위해 약간 더 복잡한 호출 흐름을 빌드하겠다. 4개의 메서드를 정의할 것인데, 그 중 하나(defineBlock)는 escaping 블록을 생성하고, 하나(arg:)는 이 블록을 평가하며, 다른 하나(evaluatingBlock:)는 블록을 실행한다. 리턴의 escaping 행위를 강조하기 위해 우리는 evaluatingBlock: 을 정의하여 그 인자의 평가가 끝나면 무한 루프에 빠지게 됨을 주목하라.

```
Bexp>>start
    | res |
    self traceCr: 'start start'.
    res := self defineBlock.
    self traceCr: 'start end'.
    ^ res

Bexp>>defineBlock
    | res |
    self traceCr: 'defineBlock start'.
    res := self arg: [ self traceCr: 'block start'.
        1 isZero ifFalse: [ ^ 33 ].
        self traceCr: 'block end'. ].
    self traceCr: 'defineBlock end'.
    ^ res

Bexp>>arg: aBlock
    | res |
    self traceCr: 'arg start'.
    res := self evaluateBlock: aBlock.
    self traceCr: 'arg end'.
    ^ res

Bexp>>evaluateBlock: aBlock
    | res |
    self traceCr: 'evaluateBlock start'.
    res := self evaluateBlock: aBlock value.
    self traceCr: 'evaluateBlock loops so should never print that one'.
    ^ res
```

Bexp new start 를 실행하면 아래와 같이 출력된다 (호출의 흐름을 강조하기 위해 들여쓰기를 적용하였다).

```
start start
    defineBlock start
        arg start
            evaluateBlock start
                block start
start end
```

호출하는 메서드 start가 완전히 실행되었음을 확인할 수 있다. 메서드 defineBlock은 완전히 실행되지 않았다. 사실 그 escaping block[^33]은 메서드 evaluateBlock:에서 두 번의 호출만큼 떨어져 실행된다. 블록의 평가는 블록 홈 컨텍스트 전송자(예: 블록을 생성한 메서드를 호출한 컨텍스트)로 리턴한다.

블록의 리턴 문이 evaluateBlock: 메서드에서 실행되면 실행은 보류 중(pending)인 계산을 버리고 블록의 홈 컨텍스트를 생성한 메서드 실행 포인트로 리턴한다. 블록은 defineBlock 메서드에서 정의된다. 블록의 홈 컨텍스트는 defineBlock 메서드의 정의를 표현하는 활성 컨텍스트다. 따라서 리턴 표현식은 defineBlock 실행 직후에 start 메서드 실행으로 리턴한다. 그것이 바로 arg: 와 evaluateBlock: 의 보류 중인 실행이 파기되는 이유이며, start 메서드의 실행이 끝나는 것을 목격하는 이유이다.

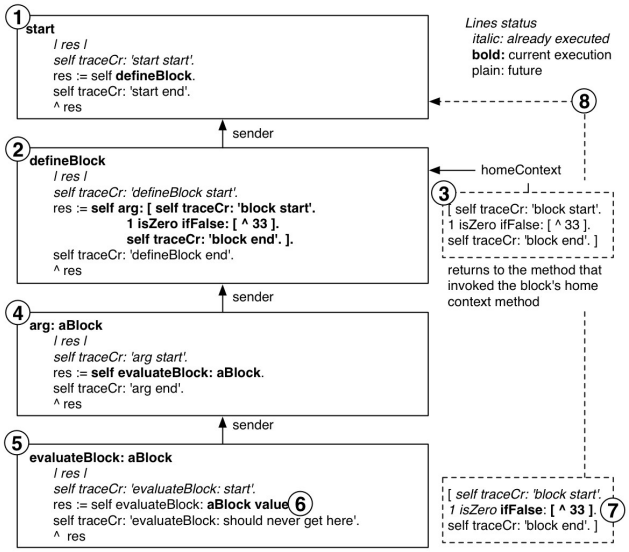


그림 14.3: non-local 리턴 실행의 블록은 블록 홈 컨텍스트를 활성화한 메서드 실행으로 리턴한다. 프레임은 컨텍스트를 표현하고, 점선은 다른 실행 시간에서 동일한 블록을 나타낸다.

그림 14.3에 나타난 바와 같이 [^33]은 그것의 홈 컨텍스트 텍스트의 전송자에게 리턴할 것이다. [^33] 홈 컨텍스트는 defineBlock 메서드의 실행을 표현하는 컨텍스트이므로 그 결과를 start 메서드로 리턴할 것이다.

- Step 1은 defineBlock 메서드의 호출이 이루어지기까지 실행을 나타낸다. Trace 'start start'가 출력된다.
- Step 3는 Step 2에서 이루어지는 블록 생성까지의 실행을 나타낸다. 블록의 홈 컨텍스트는 defineBlock 메서드 실행 컨텍스트다.
- Step 4는 메서드 호출까지의 실행을 의미한다.
- Step 5는 블록 평가까지의 실행을 나타낸다. 'evaluateBlock start'가 출력된다.
- Step 6은 조건문까지의 블록 실행을 나타내며, 'block start'가 출력된다.
- Step 7은 리턴 문까지의 실행을 나타낸다.
- Step 8은 리턴 문의 실행을 표현한다. 이는 블록 홈 컨텍스트의 전송자로 리턴하는데, 가령 start 메서드에서 defineBlock 메서드의 호출 직후를 예로 들 수 있겠다. 실행은 계속되고 'start end'가 출력된다.

> Non local return [^ ···] 은 블록 홈 컨텍스트의 전송자로 리턴한다. 즉, 블록을 생성한 메서드를 호출한 메서드 실행 포인트로 리턴한다는 의미다.

정보 접근하기. 블록의 홈 컨텍스트를 수동으로 확인하고 찾기 위해서는 다음을 실행할 수 있다. thisContext home inspect 표현식을 defineBlock 메서드의 블록에 추가한다. 현재 실행 컨텍스트를 통해 클로저로 접근하고 그 홈 컨텍스트를 얻는 thisContext closure home inspect 표현식을 추가할 수도 있다. 두 가지 경우에서 만일 evaluteBlock: 메서드의 실행 도중에 블록이 평가된다 하더라도 블록의 홈 컨텍스트는 defineBlock 메서드라는 사실을 주목한다.

그러한 표현식은 블록 평가 중에 실행될 것이다.

```
Bexp>>defineBlock
    | res |
    self traceCr: 'defineBlock start'.
    res := self arg: [ thisContext home inspect.
        self traceCr: 'block start'.
        1 isZero ifFalse: [ ^ 33 ].
        self traceCr: 'block end'. ].
    self traceCr: 'defineBlock end'.
    ^ res
```

실행이 어디서 끝나는지 확인하기 위해서는 thisContext home sender copy inspect 표현식을 사용할 수 있는데, 이를 사용하면 start 메서드에서 할당(assignment)을 가리키는 메서드 컨텍스트를 리턴한다.

몇 가지 추가 예제. 아래 예제들은 escaping 블록이 그들의 홈 컨텍스트의 전송자로 jump 함을 보여준다. 예를 들어, 앞의 예제들은 메서드 start가 완전히 실행되었음을 보였다. valuePassingEscapingBlock을 아래와 같이 BlockClosure 클래스에 정의한다.

```
BlockClosure>>valuePassingEscapingBlock
    self value: [ ^nil ]
```

그리고 메서드 인자가 false인 경우 메서드가 오류를 발생한다는 간단한 assert를 정의한다.

```
Bexp>>assert: aBoolean
    aBoolean ifFalse: [Error signal]
```

다음으로 아래 메서드를 정의한다.

```
Bexp>>testValueWithExitBreak
    | val |
    [ :break |
        1 to: 10 do: [ :i |
            val := i.
            i = 4 ifTrue: [ break value ] ] ] valuePassingEscapingBlock.
    val traceCr.
    self assert: val = 4.
```

해당 메서드는 루프의 step 4에 도달하는 즉시 break 인자가 평가되는 블록을 정의한다. Bexp new testValueWithExitBreak의 실행은 오류를 발생하지 않고 실행되고, Transcript 상에 4를 출력한다. 이로써 루프가 중지되었고, 값이 출력되었으며, 주장이 확인되었다.

위의 testValueWithExitBreak 메서드에서 value:[nil] 이 전송한 valuePassingEscaping-Block 메시지를 변경할 경우 testValueWithExitBreak 메서드의 실행이 종료될(exit) 것이기 때문에 당신은 블록이 평가될 때 추적(trace)을 얻지 못할 것이다. 이러한 경우 valuePassingEscap-

ingBlock을 호출하는 것은 value:[nil] 을 호출하는 것과 동일하지 않은데, escaping block[nil] 의 홈 컨텍스트가 다르기 때문이다. 원본 valuePassingEscapingBlock을 이용하면 블록 [nil] 의 홈 컨텍스트는 testValueWithExitContinue 메서드 자체가 아니라 valuePassingEscaping-Block에 해당한다. 따라서 평가할 경우 escaping 블록은 실행 흐름을 testValueWithExitBreak 내의 valuePassingEscapingBlock 메시지로 변경한다 (흐름이 defineBlock 메시지의 호출 직후로 돌아왔던 앞의 예제와 비슷하게). assert: 앞에 self halt를 놓으면 확신할 수 있을 것이다. 우리 예제에서는 halt에 도달하겠지만 다른 경우는 그렇지 않을 것이다.

Non-local 리턴 블록. 블록은 항상 자신의 홈 컨텍스트에서 평가되기 때문에 이미 리턴된 메서드 실행으로부터 리턴을 시도하는 것이 가능하다. 이러한 런타임 오류 상태는 가상 머신에 의해 잡힌다.

```
Bexp>>returnBlock
    ^ [ ^ self ]

Bexp new returnBlock value   Exception
```

returnBlock을 실행하면 메서드는 그 호출자에게 (여기서는 최상위 수준 실행) 블록을 리턴한다. 블록을 평가할 때는 블록을 정의하는 메서드가 이미 종료되었고 블록은 일반적으로 블록 홈 컨텍스트의 전송자에게 리턴하는 리턴 표현식을 포함하고 있으므로 오류가 시그널링된다.

결론. non-local 표현식 ([^ …]) 이 있는 블록은 블록 홈 컨텍스트의 전송자로 리턴한다 (컨텍스트는 블록 생성으로 이어지는 실행을 나타낸다).

컨텍스트 : 메서드 실행 표현하기

변수를 검색할 때 블록은 홈 컨텍스트를 참조함을 확인했다. 따라서 이제는 컨텍스트를 살펴볼 것이다. 컨텍스트는 프로그램 실행을 표현한다. Pharo 실행 엔진은 아래의 정보가 있는 현재 실행 상태를 나타낸다.

1. 바이트코드가 실행 중인 CompiledMethod
2. 그 CompiledMethod에서 실행될 다음 바이트코드의 위치. 이는 해석기(interpreter)의 프로그램 포인터다.
3. CompiledMethod를 호출한 메시지의 인자와 수신자.
4. CompiledMethod가 필요로 하는 임시 변수.
5. 콜 스택.

Pharo에서 MethodContext 클래스는 이러한 실행 정보를 나타낸다. MethodContext 인스턴스는 특정 실행 포인트에 관한 정보를 보유한다. Pseudo-variable thisContext는 현재 실행 포인트로의 접근성을 제공한다.

Contexts와 상호작용하기

예를 들어보도록 하겠다. 우선 Bexp new first: 33을 이용해 아래 메서드를 정의하고 실행하라.

```
Bexp>>first: arg
    | temp |
    temp := arg*2.
    thisContext copy inspect.
    ^ temp
```

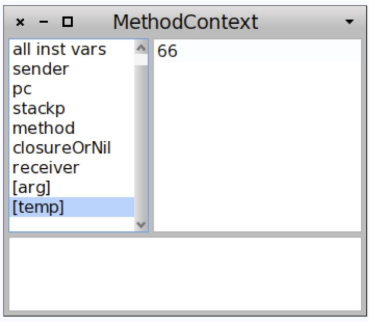


그림 14.4: 주어진 실행 포인트에서 임시 변수 temp의 값으로 접근할 수 있는 메서드 컨텍스트.

그림 14.4에서 보이는 바와 같이 인스펙터를 얻을 것이다. 가상 머신은 컨텍스를 재사용함으로써 메모리 소모량을 제한하기 때문에 우리는 thisContext를 이용해 얻은 현재 컨텍스트를 복사함을 명시한다.

MethodContext는 메서드 실행의 활성 컨텍스트를 나타낼 뿐만 아니라 블록에 대해서도 나타낸다. 현재 컨텍스트의 값을 몇 가지 살펴보자.

- sender는 현재 컨텍스트의 생성을 야기한 이전 컨텍스트를 가리킨다. 여기서 표현식을 실행하면 컨텍스트가 생성되고, 이 컨텍스트는 현재 컨텍스트의 전송자가 된다.
- method는 현재 실행 중인 메서드를 가리킨다.
- pc 는 마지막으로 실행된 명령어에 대한 참조를 유지한다. 여기서 그 값은 27이다. 어떤 명령어가 참조되는지를 확인하려면 method 인스턴스 변수를 더블 클릭하고 all bytecodes 필드를 선택하면 그림 14.5에 표시된 바와 같이 다음으로 실행될 명령어는 pop 임을 (명령어 28) 알 수 있다.
- stackp는 컨텍스트에서 변수의 스택 깊이를 정의한다. 대부분의 경우 그 값은 보관된 임시 변수(인자 포함)의 개수가 된다. 하지만 특정 경우, 가령 메시지 전송 중인 경우 스택의 깊이가 증가하여 수신자가 밀리고 (pushed) 나서 인자가 밀리고, 마지막으로 메시지 전송이 실행되어 스택의 깊이는 이전 값으로 복귀된다.
- closureOrNil은 현재 실행 중인 클로저 또는 nil로의 참조를 보유한다.
- receiver는 메시지 수신자를 의미한다.

MethodContext 클래스와 그 슈퍼클래스들은 특정 컨텍스트에 관한 정보를 얻기 위한 다수의 메서드를 정의한다. 예를 들어, tempNamed: 를 전송함으로써 특정 임시 변수의 값을 얻고, arguments 메시지를 전송함으로써 인자의 값을 얻을 수 있다.

블록 중첩과 컨텍스트

이제 중첩 블록의 사례와 그것이 홈 컨텍스트에 미치는 영향에 대해 살펴보자. 사실상 블록은 그것이 생성된 컨텍스트를 가리키는데, 이러한 컨텍스트를 외부 컨텍스트(outer context)라고 한다. 상황에 따라 블록의 외부 컨텍스트는 블록의 홈 컨텍스트가 되기도 하고 그렇지 않기도 하다. 복잡하지 않다. 각 블록은 어떤 컨텍스트 내부에서 생성된다는 말이다. 이것이 블록의 외부 컨텍스트다. 블록은 이러한 외부 컨텍스트의 내부에서 직접적으로 생성된다. 홈 컨텍스트는 메서드 수준에서의 컨텍스트다. 블록이 중첩되지 않은 경우 외부 컨텍스트도 블록 홈 컨텍스트가 된다.

하지만 블록이 다른 블록 실행 내부에서 중첩될 경우 외부 컨텍스트는 블록 실행 컨텍스트를 의미하고, 블록 실행의 블록에 해당하는 outerContext는 홈 컨텍스트가 된다. 외부 컨텍스트 단계의 수는 중첩 수준만큼 존재한다.

아래 예를 살펴보자. 실행 시 ok를 누르면 대화창이 팝업된다.

```
| homeContext b1 |
homeContext := thisContext.
b1 := [| b2 |
    self assert: thisContext closure == b1.
    self assert: b1 outerContext == homeContext.
    self assert: b1 home = homeContext.
    b2 := [self assert: thisContext closure == b2.
        self assert: b2 outerContext closure outerContext == homeContext].
        self assert: b2 home = homeContext.
    b2 value].
b1 value
```

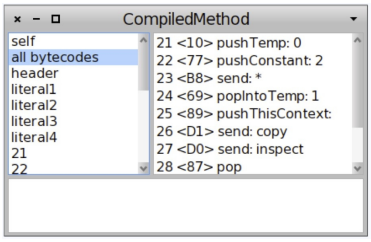


그림 14.5: pc 변수는 27을 보유하는데, 실행된 마지막 (바이트코드) 명령어가 메시지 전송 inspect였기 때문이다.

- 먼저 블록 생성 이전의 컨텍스트 homeContext에서 설정한다. homeContext는 블록 b1과 b2의 홈 컨텍스트인데, 그 이유는 이 블록들이 그에 해당하는 실행 도중에 정의되기 때문이다.

- ThisContext closure == b1 은 블록 b1의 실행 내부의 컨텍스트가 b1을 향하는 포인터를 갖고 있음을 보여준다. Assignment 이후부터 시작해 실행 도중에 b1가 정의되기 때문에 새로운 것은 없다. b1의 홈 컨텍스트는 그 외부 컨텍스트와 동일하다.
- B2 실행 내부에서 현재 컨텍스트는 b2자체를 가리키는데 그것이 바로 클로저이기 때문이다. b2가 정의된 클로저의 외부 컨텍스트가 더 흥미로운데, 가령 b1는 homeContext를 가리킨다. 마지막으로 b2의 홈 컨텍스트는 homeContext 이다. 마지막 포인트는 모든 중첩 블록이 구분된 외부 컨텍스트를 갖고 있으나 같은 홈 컨텍스트를 공유함을 보여준다.

메시지 실행

가상 머신은 메서드 또는 현재 실행 중인 (활성화된다는 용어가 사용되기도 한다) 블록마다 하나의 컨텍스트 객체로서 실행 상태를 나타낸다. Pharo에서 메서드 및 블록 실행은 MethodContext 인스턴스에 의해 표현된다. 본 장의 나머지 부분에서는 컨텍스트, 메서드 실행, 블록 클로저 평가를 다루겠다.

메시지 전송하기

수신자에게 메시지를 전송하기 위해 VM은 다음을 수행해야 한다.

1. 수신자 객체의 헤더(header)를 이용해 수신자의 클래스를 찾는다.
2. 클래스 메서드 dictionary에서 메서드를 검색한다. 메서드가 발견되지 않으면 각 슈퍼클래스에서 이 검색을 반복한다. 슈퍼클래스 사슬에서 어떤 클래스도 이 메시지를 이해할 수 없는 경우 VM은 수신자에게 doesNotUnderstnad: 메시지를 전송하여 오류가 해당 객체에 적절한 방식으로 처리되도록 한다.
3. 적절한 메서드가 발견되면,
   (a) 메서드 헤더를 읽어서 메서드와 연관된 프리미티브를 확인한다.
   (b) 프리미티브가 있다면 그것을 실행한다.
   (c) 프리미티브가 성공적으로 완료되면 결과 객체를 메시지 전송자에게 리턴한다.
   (d) 프리미티브가 존재하지 않거나 실패할 경우 다음 단계를 계속한다.
4. 새 컨텍스트를 생성한다. 프로그램 카운터, 스택 포인터, 홈 컨텍스트를 준비하고 메시지를 전송하는 컨텍스트의 스택으로부터 인자와 수신자를 새 스택으로 복사한다.
5. 해당하는 새 컨텍스트를 활성화하고 새 메서드에서 명령어를 실행하기 시작한다.

메시지 전송 전의 실행 상태는 기억해야 하는데, 메시지 전송 이후 명령어는 메시지가 리턴할 때 실행되어야 하기 때문이다. 상태는 컨텍스트를 이용해 저장된다. 시스템에는 항상 다수의 컨텍스트가 존재한다. 현재 실행 상태를 나타내는 컨텍스트를 활성 컨텍스트라 부른다.

메시지 전송이 활성 컨텍스트에서 발생하면 활성 컨텍스트는 보류되고, 새 컨텍스트가 생성되어 활성화된다. 보류된 컨텍스트는 다시 활성화될 때까지 본래 컴파일된 메서드와 연관된

상태를 유지한다. 컨텍스트는 자신이 보류되었다는 사실을 기억해야 하며, 그래야만 결과가 리턴 시 보류된 컨텍스트가 재개될 수 있다. 보류된 컨텍스트를 새 컨텍스트의 전송자라 부른다. 그림 14.6은 컴파일된 메서드와 컨텍스트 간 관계를 표현한다. 메서드는 현재 실행된 메서드를 가리킨다. 프로그램 카운터는 컴파일된 메서드의 마지막 명령어를 가리키며, 전송자는 이전에 활성화된 컨텍스트를 가리킨다.

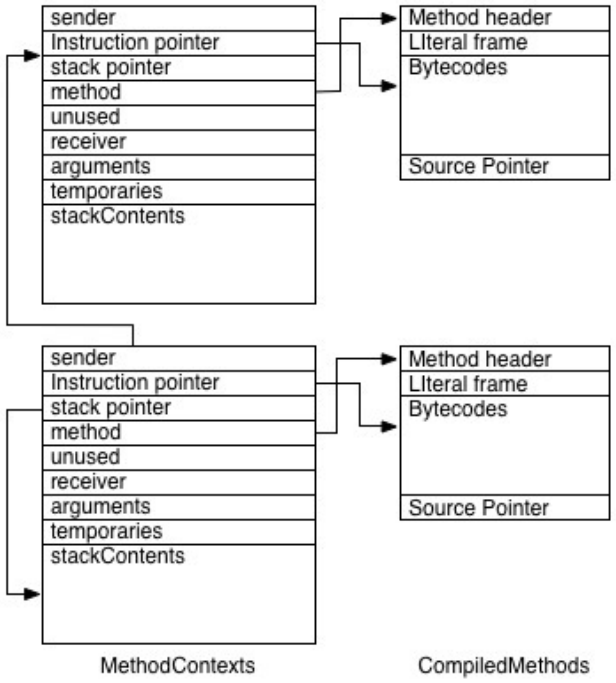


그림 14.6: 컨텍스트와 컴파일된 메서드의 관계.

구현의 개요

블록에 대한 임시 변수와 인자는 메서드에서와 같은 방식으로 처리된다. 인자는 스택 상에서 전달되고 임시 변수는 해당하는 컨텍스트에 보유된다. 그럼에도 불구하고 블록은 메서드보다 더 많은 변수로 접근이 가능한데, 블록은 enclosing 메서드로부터의 임시 변수와 인자로 참조할 수 있기 때문이다. 앞서 살펴보았듯 블록은 자유롭게 전달이 가능하며 언제든 활성화될 수 있다. 어떤 경우든 블록은 그것이 정의된 메서드로부터 변수를 수정 및 접근할 수 있다.

그림 14.7에 표시된 예를 살펴보자. ExampleReadInBlock 메서드의 블록에서 사용된 temp 변수는 non-local 변수이거나 원격 변수다. temp는 메서드 body에서 초기화 및 변경되고, 후에 블록에서 읽힌다. 변수의 실제 값은 블록 컨텍스트에 보관되지 않고 정의하는 메서드 컨텍스트, 즉 홈 컨텍스트에서 보관된다. 일반적인 구현에서 블록의 홈 컨텍스트는 그 closure를 통해 접근이 가능하다. 이러한 접근법은 메서드와 블록을 포함해 모든 객체가 일급(first-class)

객체일 경우에 효과가 좋다. 블록은 홈 컨텍스트 밖에서 평가되면서도 여전히 원격 변수를 참조할 수 있다. 따라서 모든 홈 컨텍스트는 메서드 활성보다 오래 생존할 수 있다.

구현. 앞에서 블록 컨텍스트와 관련해 언급한 접근법의 경우, 저수준 관점에서 볼 때 단점이 있다. 메서드와 블록 컨텍스트가 일반 객체인 경우 언젠가는 쓰레기 수집되어야 함을 의미한다. 다른 많은 객체를 호출하는 작은 메서드를 이용하는 일반적인 코딩 실습을 겸한다면 스몰토크 시스템은 수많은 컨텍스트를 생성할 수 있다.

메서드 컨텍스트를 처리하는 가장 효율적인 방식은 애초에 생성하지 않는 것이다. 가상 머신 수준에서 이는 실제 스택 프레임을 이용해 달성할 수 있겠다. 메서드 컨텍스트는 스택 프레임으로 쉽게 매핑이 가능하여, 메서드를 호출할 때마다 우리는 새 프레임을 생성하고, 메서드로부터 리턴할 때마다 현재 프레임을 삭제한다. 이 점에서 보면 스몰토크는 C와 별반 다르지 않다. 즉, 메서드로부터 리턴할 때마다 메서드 컨텍스트(스택 프레임)가 즉시 제거됨을 뜻한다. 따라서 고수준의 쓰레기 수집이 전혀 필요하지 않다. 그럼에도 불구하고 블록을 지원해야 한다면 스택의 이용은 훨씬 더 복잡해진다.

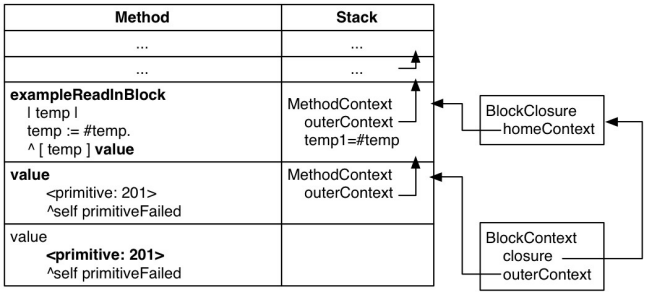


그림 14.7: 클로저 처음 이해하기.

앞서 언급했듯이 홈 컨텍스트로서 사용되는 메서드 컨텍스트는 그 활성 컨텍스트보다 오래 지속된다. 메서드 컨텍스트가 지금까지 설명한 바와 같이 작동한다면 스택 프레임이 제거되었는지 매번 홈 컨텍스트를 확인해야 할 것이다. 이는 성능에 큰 불이익을 가져온다. 그렇기 때문에 컨텍스트에 스택을 사용하기 위한 다음 단계는 메서드로부터 리턴할 때 메서드 컨텍스트가 안전하게 제거되도록 확보하는 일이다.

그림 14.8은 non-local 변수가 더 이상 홈 컨텍스트에 직접 보관되지 않고 할당된 힙(heap), 즉 구분된 원격 배열에 저장되는 모습을 보여준다.

요약

이번 장에서는 어휘적 클로저(lexical closures)라고도 불리는 블록의 사용법과 구현 방법을 학습했다. 블록을 정의하는 메서드가 리턴한다 하더라도 블록을 사용할 수 있음을 확인하였다. 블록은 고유의 변수를 비롯해 인스턴스 변수, 임시 변수, 정의하는 메서드의 인자에 해당하는 non local 변수로도 접근이 가능하다. 블록이 메서드를 어떻게 종료하고 전송자에게 값을 리턴하는지도 살펴보았다. 이러한 블록들은 non-local returning block이라 부르며, 오류를

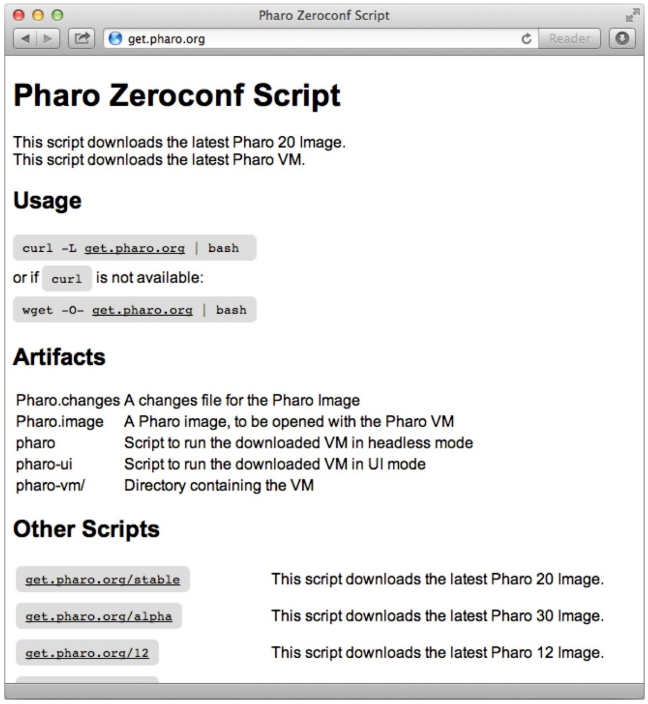


그림 14.8: 메서드가 리턴할 때 계속해서 떠나도록(leave) VM이 원격 변수를 보관하는 방법.

피하기 위해 특별히 주의를 기울여야 하는데, 블록이 이미 리턴한 메서드를 종료할 수도 있기 때문이다. 마지막으로 컨텍스트가 무엇인지, 그들이 블록 생성과 실행에서 얼마나 중요한 역할을 하는지도 설명하였다. 뿐만 아니라 thisContext pseudo 변수가 무엇인지, 이를 사용해 컨텍스트의 실행에 관한 정보를 얻고 잠재적으로 변경할 수 있는 방법도 살펴보았다.

내용을 명료하게 정리해준 Eliot Miranda에게 감사의 말을 전한다.

제 15 장

소수(little numbers) 탐구하기

우리는 항상 숫자를 조작한다. 따라서 이번 장에서는 정수가 이진수 표현으로 매핑되는 방식을 간단히 살펴보고자 한다. 상자를 열고 언어 구현자 관점을 취해 작은 정수들이 어떻게 표현되는지 기꺼이 탐구할 것이다.

가장 먼저 디지털 세계의 기본인 수학을 몇 가지 상기시키고자 한다. 이후 정수, 특히 작은 정수가 어떻게 인코딩되는지 살펴볼 것이다. 시간이 지나면서 사람들이 잊어버리곤 하는 단순한 지식이니만큼 우리는 이를 상기시키는 데에 목표를 둔다.

2 제곱과 숫자

단순한 수학 문제부터 시작해보자. 디지털 세계에서 정보는 2의 제곱으로 인코딩된다. 새로운 건 없다. 스몰토크에서 숫자를 거듭제곱하는 것은 숫자로 raisedTo: 메시지를 전송하여 이루어진다. 몇 가지 예를 들어보겠다. 그림 15.1은 2 제곱을 보여준다.

```
2 raisedTo: 0
    → 1
2 raisedTo: 2
    → 4
2 raisedTo: 8
    → 256
```

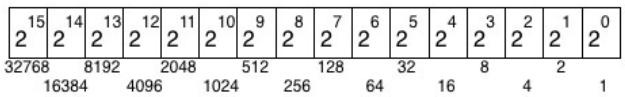


그림 15.1: 2제곱과 그에 해당하는 수치.

2제곱의 시퀀스를 이용해 숫자를 인코딩할 수 있다. 예를 들어, 13이란 숫자는 어떻게 인코딩할까? 24=16이므로 24보다 클 수는 없다. 따라서 8+4+1, 23+22+20이어야 한다. 이제 2곱을 그림 15.2에 표시된 시퀀스로 정렬하면 13이 1101로 인코딩됨을 볼 수 있다. 따라서 2제곱의

시퀀스를 이용해 숫자를 인코딩할 수 있는 것이다. 시퀀스의 요소는 비트라고 부른다. 13은 4 비트로 인코딩되고, 1 ∗ 23+1∗ 22+0∗ 21+1∗ 20에 해당한다. 이러한 비트의 시퀀스는 숫자에 대한 이진수 표기법을 나타낸다. 최상위 비트는 비트 시퀀스에서 가장 왼쪽에, 최하위 비트는 가장 오른쪽에 위치한다 (20이 가장 오른쪽 비트).

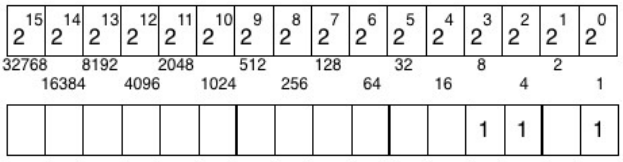


그림 15.2: 13=1 ∗ 23+1∗ 22+0∗ 21+1∗ 20.

이진수 표기법

스몰토크는 숫자를 다른 밑(base)으로 표현하기 위한 구문을 갖고 있다. 2r1101으로 작성하면 2는 밑 또는 기수를 뜻하고 여기서는 2에 해당하며, 나머지는 이를 밑으로 표현된 숫자를 나타낸다. 2r01101 또는 2r0001101으로 작성할 수도 있는데, 해당 표기법은 최하위 비트가 가장 오른쪽에 위치하는 규칙을 따르기 때문이다.

```
2r1101
    → 13
13 printStringBase: 2
    → '1101'
Integer readFrom: '1101' base: 2
    → 13
```

마지막 두 메시지, printStringBase: 와 readFrom:base: 는 후에 살펴보겠지만 음수의 내부적 인코딩을 잘 처리하지 못한다. -2 printStringBase: 2는 -10을 리턴하지만 이는 내부적 숫자 표현(2의 보수로 알려진)이 아니다. 이러한 메시지들은 그저 주어진 밑으로 숫자를 출력/읽을 뿐이다.

기수 표기법을 이용해 여러 밑으로 숫자를 명시할 수도 있다. 십진수로 작성된 15는 (10r15) 15를 리턴하는 반면 밑이 16인 15는 아래 표현식에서 보여주듯 16+5=21을 리턴한다.

```
10r15
    → 15
16r15
    → 21
```

비트 쉬프팅(bit shifting)은 2 제곱을 곱하는 것이다

정수는 비트의 시퀀스로서 표현되므로, 모든 비트를 주어진 양부터 쉬프팅한다면 또 다른 정수를 얻게 된다. 비트의 쉬프팅은 2의 곱셈/나눗셈을 실행하는 것과 같다. 그림 15.3은 이러한 요점을 보여준다. 스몰토크는 비트를 쉬프팅하는 세 개의 메시지를 제공하는데, ≫ aPositiveIn-

teger, ≪ aPositiveInteger, 그리고 bitShift: anInteger가 그것들이다. ≫ 는 수신자 나누는 반면 ≪ 는 2 제곱으로 곱한다.

아래 예제들을 통해 사용 방법을 소개하겠다.

```
2r000001000
    → 8
2r000001000 >> 1 "we divide by two"
    → 4
(2r000001000 >> 1) printStringBase: 2
    → '100'
2r000001000 << 1 "we multiply by two"
    → 16
```

bitShift: 메시지는 ≫ 와 ≪ 에 동일하지만 방향을 전환 시 음의 정수와 양의 정수를 사용한다. 양의 인자는 ≪ 와 동일한 행위를 제공하고, 수신자를 2제곱으로 곱한다. 음의 인자는 ≫ 와 유사하다.

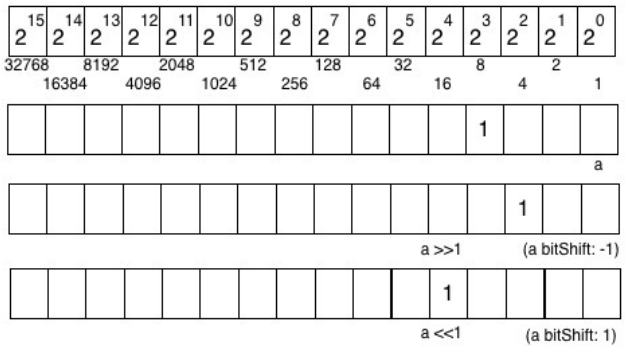


그림 15.3: 2로 곱하고 나누기.

```
2r000001000
    → 8
2r000001000 bitShift: -1
    → 4
2r000001000 bitShift: 1
    → 16
```

물론 한 번에 1 비트 이상 쉬프팅도 가능하다.

```
2r000001000
    → 8
2r000001000 >> 2 "we divide by four"
    → 2
(2r000001000 >> 2) printStringBase: 2
    → '10'
2r000001000 << 2 "we multiply by four"
    → 32
```

앞의 예제들은 1 비트 또는 2 비트로 된 숫자의 비트 쉬프팅을 보여주는데, 이 수준에서는 제약이 없다. 완전한 비트의 시퀀스는 아래와 그림 15.4의 2r000001100과 같이 쉬프팅될 수

있다.

```
(2 raisedTo: 8) + (2 raisedTo: 10)
    → 1280
2r010100000000
    → 1280
2r010100000000 >> 8
    → 5
```

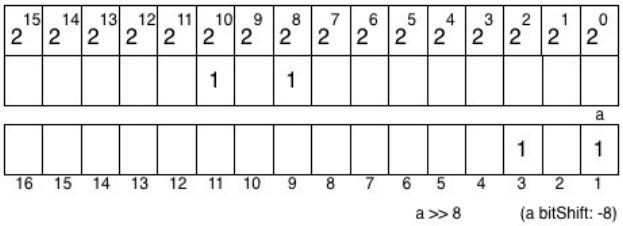


그림 15.4: 오른쪽으로 8회 이동한다. 따라서 1280으로부터 5를 얻는다.

지금까진 특별한 내용이 없다. 기본 수학 수업에서 배웠겠지만 항상 산을 오르기 전에는 언덕부터 걷는 연습이 필요하기 때문에 소개한 내용이다.

비트 조작과 접근

Pharo는 비트 조작을 위한 일반 부울 연산을 몇 가지 제공한다. 따라서 bitAnd:, bitOr:, bitXor: 과 같은 메시지들을 number로 전송할 수 있다. 그들은 연관된 Boolean 연산을 비트별로 적용할 것이다.

```
2r000001101 bitAnd: 2r01
    → 1
2r000001100 bitAnd: 2r01
    → 0
2r000001101 bitAnd: 2r1111
    → 1101
```

bitAnd: 는 숫자의 일부를 선택하는 데에 사용되기도 한다. 가령, bitAnd: 2r111 는 첫 3개의 비트를 선택한다.

```
2r000001101 bitAnd: 2r111
    → 5
2r000001101 bitAnd: 2r0
    → 0
2r0001001101 bitAnd: 2r1111
    → 13 "1101"
2r000001101 bitAnd: 2r111000
    → 8 "1000"
2r000101101 bitAnd: 2r111000
    → 40 "101000"
```

bitShift: 와 함께 bitAnd: 를 사용하면 숫자의 일부를 선택할 수 있다. 세 개의 숫자를 10 비트에서 인코딩한다고 가정해보자. 즉 3 비트에서 하나의 숫자를 인코딩하고 (0과 7 사이 숫자-ZZZYYYYXXX에서 XXX로 표기), 하나의 숫자는 4 비트에서 인코딩되며 (0과 15 사이 숫자-ZZZYYYYXXX에서 YYYY로 표기), 마지막으로 세 번째 숫자는 3비트에서 인코딩되고 ZZZYYYYXXX에서 ZZZ로 표기된다고 가정하자. 아래 표현식을 이용하면 쉽게 조작이 가능하다. 두 번째 숫자로 접근할 때는 단순히 bitShift: 만 이용해선 불가능한데, 세 번째 숫자가 여전히 존재할 것이기 때문이다. (2r1001111001bitShift: -3)는 79를 리턴하는데 우리가 얻고자 하는 값은 15다. 해결책은 세 번째 숫자를 삭제하기 위해 bitAnd: 를 사용하고 bitShift:를 실행하거나, bitShift:를 실행하고 세 번째 숫자를 삭제하는 방법이 있다. BitAnd: 인자는 인코딩의 오른쪽 부분을 선택하도록 조정되어야 한다.

```
(2r1001111001 bitShift: -3)
    → 79
(2r1001111001 bitAnd: 2r0001111000)
    → 120
(2r1001111001 bitAnd: 2r0001111000) bitShift: -3
    → 15
(2r1001111001 bitShift: -3) bitAnd: 2r0001111
    → 15
```

비트 접근. 스몰토크는 비트 정보로 접근하도록 해준다. BitAt: 메시지는 주어진 위치에 비트 값을 리턴한다. 이는 컬렉션의 색인이 1부터 시작되는 스몰토크 규칙을 따른다.

```
2r000001101 bitAt: 1
    → 1
2r000001101 bitAt: 2
    → 0
2r000001101 bitAt: 3
    → 1
2r000001101 bitAt: 4
    → 1
2r000001101 bitAt: 5
    → 0
```

시스템 자체로부터 학습한다면 흥미로울 것이다. Integer 클래스에 bitAt: 메서드의 구현을 살펴보자.

```
IInteger>>bitAt: anInteger
    ”Answer 1 if the bit at position anInteger is set to 1, 0 otherwise.
   self is considered an infinite sequence of bits, so anInteger can be any strictly positive
       integer.
    Bit at position 1 is the least significant bit.
    Negative numbers are in two-complements.

    This is a naive implementation that can be refined in subclass for speed”

    ^(self bitShift: 1 - anInteger) bitAnd: 1
```

정수 – 1에서부터 (따라서 1 – anInteger) 오른쪽으로 쉬프팅하고, bitAnd: 를 이용해 위치에 1이 있는지 0이 있는지를 알게 된다. 2r000001101를 갖고 있다고 가정할 때, 2r000001101 bitAt: 를 실행하면 4부터 쉬프팅되어 해당 비트를 선택하는 bitAnd: 1을 실행할 것이다 (예 : 1에 있을 경우 1을 리턴하고, 그 외의 경우 0을 리턴하므로 그 값이 되겠다). bitAnd: 1 을 실행

하는 것은 최하위 비트에 1이 있는지 알려주는 것과 동일하다.

다시 말하지만 사실상 특별한 내용은 없으며 모두 기억하고 있는 것을 상기시키는 데에 목적을 둔다. 이제 숫자가 2의 보수를 이용해 Pharo에서 내부적으로 인코딩되는 방법을 살펴볼 것이다. 먼저 10의 보수를 이해한 후 2의 보수를 살펴보겠다.

숫자에 대한 10의 보수

2의 보수를 완전히 이해하기 위해서는 그것이 십진수와 어떻게 작용하는지 살펴보는 편이 흥미롭다. 10의 보수에 대한 사용을 분명히 보여주는 예는 없지만 여기서 보여주고픈 요점은 보수가 덧셈을 뺄셈으로 대체하는 것이란 사실이다 (예: A의 보수를 B로 더하는 것은 B에서 A를 제하는 것과 같다).

양의 십진 정수 n에 대한 10의 보수는 10 의 (k) 제곱에서 마이너스 n인데 여기서 k는 n의 십진 표현으로 된 자릿수를 의미한다. Complement10(n) = 10k− n. Complement10(8) = 101− 8, Complement10(1968) = 104− 1968 = 8032를 예로 들 수 있겠다.

이는 아래와 같은 방법으로 계산할 수 있다.

1. 숫자의 각 자릿수 d를 9-d로 대체하고,
2. 결과 숫자에 1을 추가하라.

예제. 1968에 대한 10의 보수는 9-1, 9-9, 9-6, 9-8+1로, 예를 들자면 8031+1, 가령 8032가 된다. 규칙 2를 이용해 9-1, 9-9, 9-6, 10-8를 계산하므로 8032를 예로 들 수 있다. 따라서 1968에 대한 10의 보수는 8032가 된다. 사실 1968+8032=10000=10^5이다. 따라서 위의 정의를 올바로 따르면 8032는 10000-1968의 결과가 된다.

190680에 대한 10의 보수는 9-1, 9-9, 9-0, 9-6, 9-8, 9-0+1, 가령 809319+1, 따라서 809320가 된다. 확인해보자. 190680+809320=10000000.

숫자에 대한 10의 보수를 계산하기 위해서는 각 자릿수마다 9-d를 실행하여 결과에 추가하는 것만으로 충분하다.

일부 서적에서는 10의 보수를 계산하는 또 다른 방법을 제시한다. (1) 숫자 오른쪽 끝의 모든 0은 0으로 계속 유지된다. (2) 숫자의 오른쪽에서 0이 아닌 자릿수 d는 10-d로 대체된다. (3) 그 외 다른 자릿수 d는 9-d로 대체된다.

컴퓨터 과학자들은 첫 번째 방법을 아마도 선호할 것인데, 좀 더 일반적일 뿐더러 1을 추가하는 것이 다른 테스트를 만드는 것보다 덜 번거롭기 때문이다.

뺄셈 작용

보수 기법의 핵심은 뺄셈을 덧셈으로 전환한다는 데에 있다. 확인해보자.

예제. 8-3=5 뺄셈을 실행하길 원한다고 가정하자. 10의 보수를 이용해 뺄셈을 덧셈으로 변형할 것이다. 3에 대한 10의 보수는 9-3+1=7이다. 8에 7을 더해 15를 얻는다. 덧셈에서 얻은 carry를 버리고 (drop) 5를 얻는다. 사실 8-3=8-(10-7)=8+7-10=15-10=5가 이루어진다.

이제 98-60을 계산해보자. 60에 대한 10의 보수는 9-6, 9-0으로, 가령 39+1, 즉 40이 된다. 98 − 60 = 98 + 40 − 100 = 138 − 100 = 38. 따라서 98 − 60 = 98 + 40 = (1)38 이라 말할 수 있으며, carry를 버린다.

이제 –98 + 60 를 실행하기 위해 98에 대한 10의 보수를 계산하고 합을 계산한 후 합에 대한 10의 보수를 계산하고 반전 (negate) 하는데, –98 + 60는 2 + 60 = 62가 되고, 62에 대한 10의 보수는 38 이 되므로 –98 + 60 = –38가 된다.

다시 살펴보기. 숫자를 10의 보수로 대체하는 것은 a − b = a − (10 − c)를 기반으로 하는데, 여기서 10 = b + c 를 충족한다. 결과가 81443인 아래 표현식 190680 – 109237 을 생각해보자. 10의 보수는 109237 역시 999999-890762 또는 1000000-890763과 동일하며 890763가 109237에 대한 10의 보수라는 사실을 이용한다.

```
109237 = 999999 -- 890762
109237 = 999999 -- 890762 (+ 1 -- 1)
109237 = 1000000 -- 890762 -- 1
```

이제 첫 번째 뺄셈은 아래와 같이 표현된다.

```
190680 -- 109237
= 190680 -- (1000000 -- 890762 -- 1)
= 190680 -- 1000000 + 890762 + 1
= 190680 + 890762 + 1 -- 1000000
= 1081443 -- 1000000
= 81443
```

음수

음수의 값은 간단히 알 수 있다. 본 장 앞에서 설명했듯이 이진수 표현에 의해 주어진 모든 2 제곱을 합하면 된다. 음수 값을 얻는 것도 이제 꽤 간단해졌다. Sign bit를 음으로 계수하고, 나머지는 모두 양으로 계수한다는 점만 제외하면 앞과 동일하다. Sign bit는 최상위 비트로서, 가장 큰 수를 나타내는 비트이다(그림 15.5 참고). 예를 들어 8 비트 표현에서 이는 weight 27 과 연관된 비트일 것이다.

이를 설명해보겠다. -2는 8 비트에서 1111 1110으로 인코딩되어 표시된다. 비트 표현으로부터 값을 얻기 위해서는 –27+ 26+25+ 24+ 23+ 22+ 21+ 0 ∗ 20를 더하여, –128 + 64 + 32 + 16 + 8 + 4 + 2가 되고 결국 -2를 얻는다.

-69는 8비트에서 1011 1011로 표현된다. 비트 표현에서 값을 얻기 위해선 –27+ 0 ∗ 26+ 25+ 24+ 23+ 0∗22+ 21+ 20를 합해 –128 + 32 + 16 + 8 + 2 + 1 가 되고 결국 –69를 얻는다.

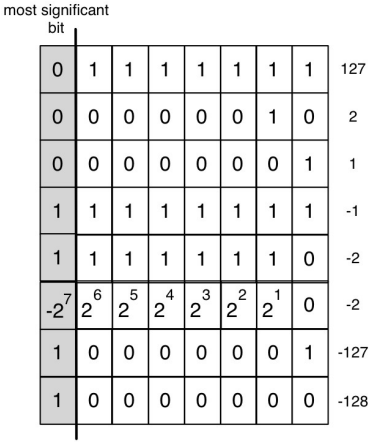


그림 15.5: 8 비트에서의 음수.

같은 규칙에 따라 -1 값이 그림 15.5에 설명한 값인지 확인하라.

비트를 세어보자. 8 비트 표현에서 우리는 0을 255의 음의 정수 또는 -128부터 64 + 32 + 16 + 8 + 4 + 2 + 1 127까지 인코딩할 수 있다. 사실 우리는 -1∗27에서 27-1로 인코딩할 수도 있다. 좀 더 일반적으로 N 비트에서는 -1∗2N-1에서 2N-1-1 정수 값으로 인코딩이 가능하다.

숫자에 대한 2의 보수

이제 모든 퍼즐 조각은 손에 있다. 양수와 음수를 인코딩하는 방법을 알고, 뺄셈을 덧셈으로 바꾸기 위해 보수를 사용하는 법도 알고 있다. 이제 2의 보수를 이용해 숫자를 반전하고 뺄셈을 실행해보자.

2의 보수는 부호가 있는(signed) 정수를 표현하기 위한 일반적인 메서드이다. 사용 시 이점으로는, 비연산자의 부호를 확인하거나 2의 보수가 0에 대해 하나의 표현만 갖는지 (음의 제로를 피하기 위해) 확인할 필요 없이 덧셈과 뺄셈이 구현된다는 데에 있다. 2의 보수를 이용해 인코딩된 부호의 숫자를 추가할 때에는 어떤 특별한 처리도 필요로 하지 않는다. 결과의 부호가 자동으로 결정되기 때문이다. 양수에 대한 2의 보수는 해당 숫자의 음수 형태를 나타낸다.

10의 보수는, 베이스 시스템에서 이용 가능한 최대수로 된 각 digit와 십진수로 된 9의 차(difference)에 1을 더하면 계산할 수 있음을 보여준다. 이제 이진수의 경우, 최대값은 1이 된다. 1-0=1과 1-1=0이란 사실 때문에 숫자에 대한 2의 보수를 계산하는 것은 1의 보수(1's)를 0의 보수(0's)로, 혹은 그 반대로 뒤집어(flipping) 1을 더하는 것과 정확히 동일하다.

숫자 2를 살펴보자. 그림 15.6에서와 같이 2는 8 비트에서 000000010로 인코딩되고, -2는 11111110로 인코딩된다. 뒤집은 000000010은 11111101이고 그에 1을 더하면 11111110을 얻는다. -2(11111110)와 126(01111110)의 차는 정수 부호를 전달하는 최상위 비트에 의해 주어진다.

Pharo에서 아래 표현식을 시도하고 실험해보라. 우리는 직접 역전(direct inversion; bitwise NOT)을 계산 후 1을 더한다. Bitwise NOT은 어떤 1이든 0으로 바꾸고, 0은 1로 바꾼다. 숫자의 비트 표현을 얻는 데에 bitString을 사용할 수도 있다. 출력된 결과의 길이를 확인하면

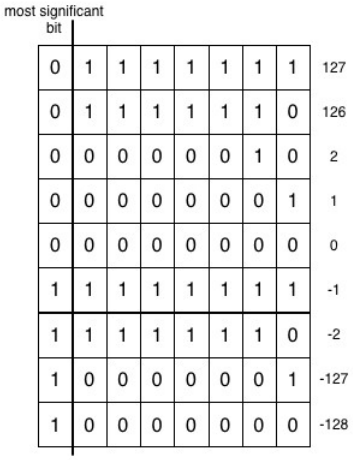


그림 15.6: 8비트에서 2의 보수 개요.

32대신 31비트임을 눈치챌 것이다. 이는 Pharo와 스몰토크 구현에서는 작은 정수가 특별히 인코딩되고, 해당 인코딩은 1 비트를 필요로 하기 때문이다.

```
2 bitString
    '00000000000000000000000000000010'
2 bitInvert bitString
    '11111111111111111111111111111101'
(2 bitInvert + 1) bitString
    '11111111111111111111111111111110'
-2 bitString
    '11111111111111111111111111111110'
```

음수에 대한 2의 보수는 다음 표현식에 의해 표시되는 해당 양수 값이다. -2에 대한 2의 보수는 2다. 먼저, 직접 역전(bitwise NOT)을 계산하고 1을 더한다.

```
-2 bitString
    → '11111111111111111111111111111110'
-2 bitInvert bitString
    → '00000000000000000000000000000001'
(-2 bitInvert + 1) bitString
    → '00000000000000000000000000000010'
2 bitString
    → '00000000000000000000000000000010'
```

숫자를 반전하고 2의 보수를 계산하면 동일한 이진수 표현을 제공함을 알 수 있다.

```
(2r101 bitInvert + 1) bitString
    returns '11111111111111111111111111111011'
2r101 negated bitString
    returns '11111111111111111111111111111011'
```

한 가지 예외가 있다. 주어진 비트 번호에서, 즉 그림 15.6에서처럼 비트 8이라고 가정하면, 우리는 음수를 계산하지만 가장 큰 음수를 제하고는 숫자에 대한 2의 보수(모든 비트를 뒤집고

1을 추가)를 계산한다. 8비트 표현에서 가장 큰 음수는 -128(1000 0000)으로 역전하면 (0111 1111)이며, 1을 더하면 (1000 0000)이 된다. 8비트 부호가 있는 규칙에서 128를 인코딩할 수는 없다. 여기서 carry는 sign bit에 의해 "먹힌다(eaten)".

뺄셈. 숫자를 어떤 숫자로 빼기 위해서는 두 번째 숫자에 대한 2의 보수를 첫 번째 숫자로 추가하면 된다.

가령 110110-101을 계산하려면 101에 대한 2의 보수를 계산해 추가한다. 110110과 111011을 더하여 110001을 얻는다. 따라서 54-5=49가 올바르겠다.

```
1110110 - 101
  110110
+ 111011
----------
  110001
```

Pharo에서 테스트해보자.

```
(2r110110 - 2r101) bitString
    → '00000000000000000000000000110001'
(2r110110 bitString)
    → '00000000000000000000000000110110'
2r101 bitString
    → '00000000000000000000000000000101'
2r101 negated bitString
    → '11111111111111111111111111111011'
```

8 비트만 사용해 덧셈을 제시하면 아래와 같은 모습이 발생한다.

```
carry 1111111
      00110110
+     11111011
----------------
      00110001
```

overflowing carry가 버려졌음을 주목하라.

이제 결과가 음수인 경우 역시 잘 처리된다. 예를 들어, 15-35를 계산하고 싶다면 -20를 얻어야 하고 바로 그 값을 얻게 될 것이다. 15가 0000 1111로 인코딩되고 35은 0010 0011로 인코딩되는 모습을 살펴보자. 35에 대한 2의 보수는 1101 1101이다.

```
      0011111 (carry)
      0000 1111
      1101 1101
-----------------
  1111111101100
```

결과적으로 -20을 얻는다.

Pharo에서 SmallIntegers

스몰토크의 작은 정수는 31 비트에서 2의 보수 산술을 이용한다. N-비트의 2의 보수 기수법(two's complement numeral system)은 –1 ∗ 2N–1부터 2N–1– 1범위의 모든 정수를 표현

가능하다. 따라서 31 비트의 경우 스몰토크 시스템 작은 정수 값은 -1 073 741 824 부터 1 073 741 823 범위에 해당한다. 스몰토크에서는 정수가 특수 객체이며, 그 표기 시 1비트가 소모되므로 32비트에선 작고 부호가 있는 정수에 대해 31비트를 갖고 있음을 기억하라. 물론 자동 강제변환(automatic coercion)도 갖고 있으므로 최종 프로그래머에겐 문제가 되지 않는다. 본문에서는 언어 구현의 관점을 취하고 있다. 해당 비트를 확인해보자 (이제 다루어도 될 시기다).

SmallInteger 최대값 인코딩. Pharo의 작은 정수는 31비트 상에서 인코딩되고 (내부 표현에 1비트가 필요하므로), 최소 (small integer) 음의 정수는 SmallInteger maxVal negated -1이다. 여기서는 가장 큰 음의 정수에 대한 예외를 살펴보겠다.

```
"We negate the maximum number encoded on a small integer"
SmallInteger maxVal negated
    → -1073741823
"we still obtain a small integer"
SmallInteger maxVal negated class
    → SmallInteger
"adding one to the maximum number encoded on a small integer gets a large positive integer"
(SmallInteger maxVal + 1) class
    →LargePositiveInteger
"But the smallest negative is one less than the negated largest positive small integer"
(SmallInteger maxVal negated - 1)
    → -1073741824
(SmallInteger maxVal negated - 1) class
    → SmallInteger
```

일부 메서드 이해하기. SmallInteger를 표현하는 데에 사용된 비트 수를 알기 위해선 아래를 평가하라.

```
SmallInteger maxVal highBit + 1
    returns 31
```

SmallInteger maxVal highBit는 양의 SmallInteger를 나타내는 데에 사용 가능한 최상위 비트를 알려주고, +1은 SmallInteger의 sign bit를 설명한다 (양수는 0, 음수는 1). 조금 더 살펴보자.

```
2 raisedTo: 29
    → 536870912
536870912 class
    → SmallInteger
2 raisedTo: 30
    → 1073741824
1073741824 class
    → LargePositiveInteger
(1073741824 - 1) class
    →SmallInteger
-1073741824 class
    → SmallInteger
2 class maxVal
    returns 1073741823
-1*(2 raisedTo: (31-1))
    → -1073741824
(2 raisedTo: 30) - 1
    → 1073741823
(2 raisedTo: 30) - 1 = SmallInteger maxVal
    → true
```

16진법

16진법을 언급하지 않고는 본 장을 완료할 수가 없을 것이다. 스몰토크에서는 이진수와 동일한 구문이 16진법에도 사용된다. 16rF는 F가 16 을 밑으로 하여 인코딩된다는 의미다.

hex메시지를 이용하면 숫자에 대한 16진 값을 얻을 수 있다. printStringHex 메시지를 이용하면 기수 표기법 없이 16진법으로 출력된 숫자를 얻는다.

```
15 hex
    returns '16rF'
15 printStringHex
    returns 'F'
16rF printIt
    returns 15
```

아래는 어떤 숫자와 그 16진 값에 동일한 결과를 열거한다.

```
{(1->'16r1'). (2->'16r2'). (3->'16r3'). (4->'16r4'). (5->'16r5'). (6->'16r6'). (7->'16r7').
    (8->'16r8'). (9->'16r9'). (10->'16rA'). (11->'16rB'). (12->'16rC'). (13->'16rD'). (14
    ->'16rE'). (15->'16rF')}
```

비트 조작을 할 때는 이진법보다 16진 표기법을 사용하는 편이 더 간결한 경우가 종종 있다. bitAnd:의 경우에는 이진수 표기법이 가독성이 더 뛰어날지 모르지만 말이다.

```
16rF printStringBase: 2
    returns '1111'
2r00101001101 bitAnd: 2r1111
    returns 2r1101
2r00101001101 bitAnd: 16rF
    returns 2r1101
```

요약

스몰토크는 내부의 작은 정수 표현에 2의 보수 인코딩을 사용하고, 내부 표현에 대해 비트 조작을 지원한다. 이는 간단한 인코딩을 이용해 알고리즘의 속도를 높이고 싶을 때 유용하다. 앞에서 살펴본 내용을 요약하자면 다음과 같다.

- 절대값은 음의 값을 인코딩하기 위해 보수를 사용한다.
- 비트를 왼쪽으로 여러 번 쉬프팅하면 비트에 2를 곱하고 그 인코딩 크기의 최대 값을 modulo 한다.
- 반대로 비트를 오른쪽으로 쉬프팅하면 비트를 2로 나눈다.
- 비트 연산은 어떤 절대값에서도 실행 가능하다.
- 보수는 덧셈을 뺄셈으로 바꾸는 데에 유용하므로 연산을 간소화한다.
- SmallInteger는 Pharo에서 31 비트에 인코딩된다.

Pharo는 원하는 만큼 메모리를 사용하도록 크기를 제한하는 대수(large number)를 지원함을 주목하라.

제 16 장

Floats 갖고 놀기

***Nicolas Cellier** 참여 (nicolas.cellier.aka.nice@gmail.com)*

Floats는 그 특성상 정밀하지 않아 프로그래머들을 헷갈리게 만들기도 한다. 따라서 본 장에서는 이러한 문제를 소개하고 실용적인 해결책을 몇 가지 제시하고자 한다. Floats는 다른 것이 아니라 부정확하지만 빠른 숫자라는 것이 기본 메시지다.

본 장에 설명된 대부분의 상황은 Floats가 하드웨어에 의해 구성되고 Pharo에 관련되지 않은 결과다. 다른 프로그래밍 언어에서도 마찬가지로 발생할 수 있는 문제다.

floats를 대상으로 동등성(equality)을 절대 테스트하지 말라

첫 번째 기본 규칙은 float 동등성을 절대로 비교하지 않는 것이다. 간단한 예를 들어보자. 두 개의 float를 더해도 그들의 합을 표현하는 float와 같지 않다. 예를 들어, 0.1+0.2는 0.3이 아니란 뜻이다.

```
(0.1 + 0.2) = 0.3
    → false
```

예상치 못했을 것이다. 학교에서 배운 내용이 아니다, 안 그런가? 이는 사실 놀라운 행위지만 floats는 부정확한 숫자이므로 정상이다. 이해해야 할 중요한 사항이 있다면, floats이 출력되는 방식 또한 우리의 이해에 영향을 미친다는 사실이다. 일부 접근법들은 다른 접근법들에 비해 사실을 더 간단히 표현하여 출력한다. 0.1+0.2를 출력하면, Pharo의 초기 버전들은 0.3을 출력했지만 지금은 0.30000000000000004를 출력한다. 이러한 변화는 사용자에게 거짓말을 하지 않는 편이 더 낫다는 개념에서 비롯된다. float의 부정확성이 오히려 숨기는 것보다 나은데, 언젠가 큰 코를 다치게 될 일이 발생할지도 모르기 때문이다.

```
(0.2 + 0.1) printString
    → '0.30000000000000004'
0.3 printString
    → '0.3'
```

16진 값을 살펴보면 두 개의 숫자를 확인할 수 있다.

```
(0.1 + 0.2) hex
    → '3FD3333333333334'
0.3 hex
    → '3FD3333333333333'
```

storeString 메서드 또한 두 개의 숫자가 있음을 전달한다.

```
(0.1 + 0.2) storeString
    → '0.30000000000000004'
0.3 storeString
    → '0.3'
```

closeTo:에 관하여. 두 개의 floats가 거의 같은 숫자처럼 보일 만큼 충분히 가깝다는 사실을 알기 위해선 closeTo: 메시지를 사용한다.

```
((0.1 + 0.2) closeTo: 0.3
    → true
0.3 closeTo: (0.1 + 0.2)
    → true
```

closeTo: 메서드는 비교 대상인 두 숫자의 차이가 0.0001보다 작은지를 확인한다. 그 소스 코드를 소개하겠다.

```
closeTo: num
    "are these two numbers close?"
    num isNumber ifFalse: [^[self = num] ifError: [false]].
    self = 0.0 ifTrue: [^num abs < 0.0001].
    num = 0 ifTrue: [^self abs < 0.0001].
    ^self = num asFloat
        or: [(self - num) abs / (self abs max: num abs) < 0.0001]
```

Scaled Decimal에 관하여. Scaled Decimals는 정확한 부동 소수점 수가 절대적으로 필요할 때 해결책이다. 이들은 정확수(exact number)로서 당신이 예상한 행위를 보인다.

```
0.1s2 + 0.2s2 = 0.3s2
    → true
```

이제 아래의 행을 실행하면 표현식이 같지 않음을 확인할 것이다.

```
(0.1 asScaledDecimal: 2) + (0.2 asScaledDecimal: 2) = (0.3 asScaledDecimal: 2)
    → false
```

차이가 뭘까? 가령, 0.1s2를 실행하면 처음부터 분수가 생성되므로 0.1s2 asFraction은 1/10을 리턴한다. AsScaledDecimal: 메시지는 또 다른 행위를 가진다. (0.1 asScaledDecimal: 2)는 float 0.1 (0.1 asScaledDecimal: 2)와 정확히 동일한 수를 표현하고, asFraction은 0.1 asFraction 값을 리턴한다. 0.1+0.2=0.3은 false를 리턴하므로 당연히 이 표현식의 scaled decimal은 false를 리턴할 것으로 예상할 것이다.

Float 분해하기

위의 덧셈에 수반되는 연산을 이해하기 위해서는 floats가 컴퓨터 내에서 내부적으로 어떻게 표현되는지를 알아야 한다. Pharo의 Float 포맷은 대부분 컴퓨터에서 널리 사용되는 표준, IEEE 754-1985 64비트 배정도(double precision)다 (상세한 정보는 http://en.wikipedia.org/wiki/IEEE_754-198 참고). 해당 포맷을 이용해 Float은 아래 공식에 의해 밑이 2로 표현된다.

$$sign \cdot mantissa \cdot 2^{exponent}$$

- 부호는 1 비트로 표현된다.
- 지수는 11 비트로 표현된다.
- 가수(mantissa)는 밑이 2인 분수로, 소수점 앞에 1이 따라오고 fraction point 다음에 52 개 이진 숫자로 구성된다. Pharo에서 가수를 얻기 위한 메서드는 Float»significand 이다. 이번 장에서 몇 가지 예를 제시한다.

예를 들어, 일련의 52 비트는:

01100100000000000000000000000000000000000000000000

가수가 다음과 같다는 뜻이며,

1.01100100000000000000000000000000000000000000000000

이는 아래 분수를 표현하기도 한다.

$$1 + \frac{0}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{0}{2^4} + \frac{0}{2^5} + \frac{1}{2^6} + \cdots + \frac{0}{2^{52}}$$

가수 값은 그에 따라 일반 숫자에 대해 1(포함)과 2(제외) 사이에 해당한다.

```
1 + ((1 to: 52) detectSum: [:i | (2 raisedTo: i) reciprocal]) asFloat
    → 1.9999999999999998
```

Float 빌드하기. 그러한 가수를 구성해보도록 하자.

```
(#(0 2 3 6) detectSum: [:i | (2 raisedTo: i) reciprocal]) asFloat.
    → 1.390625
```

이제 23으로 곱하여 null이 아닌 지수를 얻자.

```
(#(0 2 3 6) detectSum: [:i | (2 raisedTo: i) reciprocal]) asFloat*(2 raisedTo: 3).
    → 11.125
```

아니면 timesTwoPower 메서드를 이용한다.

```
(#(0 2 3 6) detectSum: [:i | (2 raisedTo: i) reciprocal]) asFloat timesTwoPower: 3.
    → 11.125
```

Pharo에서는 이러한 정보를 검색할 수가 있다.

```
11.125 sign.
    → 1

11.125 significand.
    → 1.390625

11.125 exponent.
    → 3
```

Pharo에는 정규화된 가수를 직접 처리하는 메시지가 없다. 대신 52 비트를 좌측으로 쉬프팅한 다음 가수를 정수로서 처리하는 것이 가능하다. 이를 행하는 데에는 한 가지 합당한 이유가 있다. 산술이 정확하기 때문에 Integer에서 연산하기가 더 수월하다는 점이다. 결과는 앞에 붙은 1을 포함하므로 일반 숫자에 대해 53 비트 길이어야 한다 (이것이 float precision이다).

```
11.125 significandAsInteger
    → 6262818231812096

11.125 significandAsInteger printStringBase: 2.
    → '10110010000000000000000000000000000000000000000000000'

'10110010000000000000000000000000000000000000000000000' size
    → 53

11.125 significandAsInteger highBit.
    → 53

Float precision.
    → 53
```

Float에 대한 내부 표현에 상응하는 정확한 분수를 검색할 수도 있다.

```
11.125 asTrueFraction.
    → (89/8)

(#(0 2 3 6) detectSum: [:i | (2 raisedTo: i) reciprocal])*(2 raisedTo: 3).
    → (89/8)
```

정확한 입력을 검색할 때까지 우리는 Float으로 추가한다. 결국 Float 연산은 정확한가? 음, 아니다, 2 제곱을 분모로 가진 분수와 분자에 몇 비트만 실험을 해봤을 뿐이다. 이러한 조건 중 하나라도 충족되지 않으면 우리는 숫자에 대한 정확한 Float 표현을 찾지 못할 것이다. 예를 들자면, 1/5를 이진 숫자의 유한수(finite number)로는 표현하는 것이 불가능하다. 따라서 0.1과 같은 소수(decimal fraction)는 위의 표현식으로는 정확히 표현할 수 없다.

```
(1/5) asFloat = (1/5).
    → false

(1/5) = 0.2
    → false
```

1/5의 분수 비트(fractional bits), 가령 2r1/2r101는 어떻게 얻는지 상세히 살펴보자. 이를 위해 먼저 나눗셈을 열거해야 한다.

```
1          | 101
10         | 0.00110011
100        |
1000       |
-101       |
  11       |
  110      |
 -101      |
    1      |
    10     |
    100    |
    1000   |
    -101   |
      11   |
      110  |
     -101  |
        1  |
```

우리 눈에 보이는 것은 하나의 사이클로, 4회의 유클리드(Euclidean) 나눗셈마다 몫으로 2r0011, 나머지 1을 얻는다. 즉, 밑이 2인 1/5 를 표현하기 위해서는 해당 비트 패턴의 무한 series가 필요하다는 의미다. (1/5)를 Float로 변환하기 위해 Pharo가 처리한 방식을 살펴보자.

```
(1/5) asFloat significandAsInteger printStringBase: 2.
    → '1100110011001100110011001100110011001100110011001100110011010'

(1/5) asFloat exponent.
    → -3
```

마지막 비트 001이 010로 올림된 것만 제외하면 우리가 예상한 비트 패턴이다. 이는 Float의 기본 올림/내림(rounding) 모드로서, 가장 가까운 짝수로 올림/내림한다. 이제 머신이 0.2를 부정확하게 표현한 이유를 이해할 수 있을 것이다. 이것은 0.1에 대한 동일한 가수로, 그 지수는 -4다.

```
0.2 significand
    → 1.6

0.1 significand
    → 1.6

0.2 exponent
    → -3

0.1 exponent
    → -4
```

이제 0.1 + 0.2로 진입하는데, (1/10)+(1/5)를 정확히 얻지는 못했다. 대신 아래의 결과를 얻었다.

```
0.1 asTrueFraction + 0.2 asTrueFraction.
    → (10808639105689191/36028797018963968)
```

이것이 전부가 아니다… 위의 분수에서 비트 패턴을 살펴보고, 피트 패턴의 span을 확인해 보면 가장 큰 비트의 위치가 1(가장 왼쪽)로 설정되고 최하위 비트의 위치는 1(가장 오른쪽)로 설정된다.

```
10808639105689191 printStringBase: 2.
    → '10011001100110011001100110011001100110011001100111'

10808639105689191 highBit.
    → 54

10808639105689191 lowBit.
    → 1

36028797018963968 printStringBase: 2.
    → '100000000000000000000000000000000000000000000000000000'
```

분모는 예상한 대로 2 제곱이지만 분자를 보관하기 위해서는 54 비트의 정밀도가 필요하다… Float은 53 비트만 제공한다. Float 표현에 들어맞기에는 또 다른 올/내림 오류가 존재할 것이다.

```
(0.1 asTrueFraction + 0.2 asTrueFraction) asFloat = (0.1 asTrueFraction + 0.2 asTrueFraction).
    → false

(0.1 asTrueFraction + 0.2 asTrueFraction) asFloat significandAsInteger.
    → '10011001100110011001100110011001100110011001100110100'
```

10진법 표현이 Float 표현으로 변환된 것을 포함해 무슨 일이 일어났는지 요약하자면 다음과 같다.

(1/10) asFloat 0.1 inexact (rounded to upper; 위로 올림)

(1/5) asFloat 0.2 inexact (rounded to upper; 위로 올림)

(0.1 + 0.2) asFloat · · · inexact (rounded to upper; 위로 올림)

3개의 부정확한 연산이 발생했고, 아쉽게도 3개의 올/내림 연산이 올림되어 전멸되기보다는 더 축적되었다. 반면 0.3의 해석은 asFloat으로서 하나의 올/내림 오류(3/10)를 야기한다. 이제 0.1+0.2=0.3을 기대할 수 없는 이유를 이해할 것이다.

연습으로 1.3∗1.3≠1.69가 되는 이유를 보이겠다.

Floats를 이용하면 출력이 부정확해진다

위의 예제에서 알게 된 가장 큰 함정은 0.1이 마치 정확한 수치인 것처럼 '0.1'로 출력되지만 사실은 정확하지 않다는 것이다. printString이 내부적으로 사용하는 absPrintExactlyOn:base:라는 이름이 약간은 혼동스러운데, 정확히 출력하지 않고 다시 읽었을 때 동일한 Float으로 올/내림될 가장 짧은 10진법 표현을 출력한다 (Pharo는 10진법 표현을 항상 가장 가까운 Float으로 변환한다).

정확한 출력을 위해 존재하는 또 다른 메시지, printShowingDecimalPlaces:를 대신 사용할 필요가 있겠다. 모든 무한 Float은 무한의 10진 숫자로 된 10진법 표현을 갖는다 (5 제곱으로 된 분모에 분자를 곱하면 숫자를 얻게 될 것이다). 아래를 살펴보자.

```
0.1 asTrueFraction denominator highBit.
    → 56
```

분수의 분모가 255이며, 0.1의 내부적 표현을 정확히 출력하기 위해서는 소수점 다음에 55개의 10진 숫자가 필요하다는 의미다.

```
0.1 printShowingDecimalPlaces: 55.
    → '0.1000000000000000055511151231257827021181583404541015625'
```

그리고 아래를 이용해 숫자를 검색할 수 있다.

```
0.1 asTrueFraction numerato*(5 raisedTo: 55).
    → 1000000000000000055511151231257827021181583404541015625
```

아래를 이용하면 결과를 확인할 수 있다.

```
1000000000000000055511151231257827021181583404541015625/(10 raisedTo: 55) = 0.1 asTrueFraction
    → true
```

위에서 확인할 수 있듯이 머신에서 표현되는 내용의 정확한 표현을 출력하기란 가능한 일이지만 상당히 번거롭다. 의심이 간다면 1.0e100을 정확하게 출력해보라.

Float 올/내림도 부정확하다

float 동등성에 대한 악명이 높긴 하나 float의 다른 측면들에도 주의를 기울여야 한다. 아래 예제를 이용해 요점을 설명하겠다.

```
2.8 truncateTo: 0.01
    → 2.8000000000000003

2.8 roundTo: 0.01
    → 2.8000000000000003
```

2.8 truncateTo:0.01이 2.8을 리턴하지 않고 2.8000000000000003을 리턴한다는 것은 놀랍지만 틀린 답은 아니다. 이는 truncateTo:와 roundTo:가 floats에서 여러 개의 연산을 실행하

기 때문인데, 부정확한 숫자에서 부정확한 연산을 실행하면 위에서 확인한 것과 같이 누적 올/내림 오류가 발생할 수 있으며, 위의 결과도 이로 인한 것이다.

정확히 연산을 실행한 후 가장 가까운 Float으로 올/내림한다 하더라도 초기에 2.8과 0.01의 부정확한 표현 때문에 결과는 여전히 부정확하다.

```
(2.8 asTrueFraction roundTo: 0.01 asTrueFraction) asFloat
    → 2.8000000000000003
```

0.01 대신 0.01s2 를 이용하면 예제가 제대로 작동하는 듯 보인다.

```
2.80 truncateTo: 0.01s2
    → 2.80s2

2.80 roundTo: 0.01s2
    → 2.80s2
```

하지만 이는 순전히 운이며, 2.8이 부정확하다는 사실만으로 아래와 같이 갑작스러운 상황을 야기하기엔 충분하다.

```
2.8 truncateTo: 0.001s3.
    → 2.799s3

2.8 < 2.800s3.
    → true
```

Float의 세계에서 자르기(truncating)는 극도로 불안정하다. 그나마 ScaledDecimal을 올/내림에 이용하면 그러한 불일치를 야기할 가능성이 낮으나, 마지막 숫자를 대상으로 할 때는 이마저도 불안정하다.

부정확한 표현 갖고 놀기

마무리를 짓기 위해 부정확한 표현을 조금 더 갖고 놀아보자. 여러 숫자들 간 차이를 확인해보자.

```
{
((2.8 asTrueFraction roundTo: 0.01 asTrueFraction) - (2.8 predecessor)) abs -> -1.
((2.8 asTrueFraction roundTo: 0.01 asTrueFraction) - (2.8)) abs -> 0.
((2.8 asTrueFraction roundTo: 0.01 asTrueFraction) - (2.8 successor)) abs -> 1.
} detectMin: [:e | e key ]

    → 0.0->1
```

아래 표현식은 0.0->1을 리턴하며 이는 (2.8 asTrueFraction roundTo: 0.01 asTrueFraction) asFloat = (2.8 successor) 을 뜻한다.

하지만 아래를 기억하라.

```
(2.8 asTrueFraction roundTo: 0.01 asTrueFraction) ~= (2.8 successor)
```

(2.8 asTrueFraction roundTo: 0.01 asTrueFraction)에 가장 가까운 Float은 (2.8 successor)라고 해석되어야 한다.

한계를 확인하고 싶다면 아래를 시도해보라.

```
((2.8 asTrueFraction roundTo: 0.01 asTrueFraction) - (2.8 successor asTrueFraction))
    asFloat
    → -2.0816681711721685e-16
```

요약

Floats는 광범위한 10진 값을 지원하는 실수의 근사치에 해당한다. 본 장에서는 다음과 같은 요점을 살펴보았다.

- float을 비교 시 =를 절대 사용하지 말라 (예 : (0.1+0.2)=0.3은 false를 리턴한다)
- 대신 closeTo:를 사용하라 (예 : (0.1+0.2) closeTo: 0.3은 true를 리턴한다)
- 부동 소수점 수는 밑이 *sign x mantisa(가수) x* $2^{exponent(지수)}$ (예 : 1.2345=12345x10^{-4})로 표현된다
- float로 올림(rounding up)하거나 자를 경우 truncateTo: 와 roundTo: 가 항상 작동하는 것은 아니다 (예 : 2.8 roundTo: 0.02은 2.800…003을 리턴한다)

floats에 관해서는 알아야 할 사항들이 아직 더 많으므로 충분히 학습했다면 아래 링크를 확인해보는 것도 좋은 생각이다. "What Every Computer Scientist Should Know About Floating-Point Arithmetic" (http://www.validlab.com/goldberg/paper.pdf).

제 V 부

Tools

제 17 장

애플리케이션 프로파일링하기

소프트웨어 공학이 시작되던 때부터 프로그래머들은 애플리케이션 성능과 관련된 문제에 직면해왔다. 더 낫고 빠른 개발 프로세스를 지원하도록 프로그래밍 환경이 크게 개선되었지만 프로그래밍 시 성능 문제를 다루기 위해서는 꽤 많은 재주가 필요하다.

대체적으로 애플리케이션을 최적화하는 일은 그다지 어렵지 않다. 일반적인 개념은, 느리고 자주 호출되는 메서드를 더 빠르거나 덜 자주 호출되도록 만드는 데에 있다. 애플리케이션의 최적화는 보통 애플리케이션을 복잡하게 만든다. 따라서 애플리케이션에 대한 요구조건이 잘 이해하고 다루었을 때에만 최적화를 권한다. 다시 말하자면, 무엇을 하는지 확신이 설 때에만 애플리케이션을 최적화해야 한다는 말이다. Kent Back이 말했듯이, 첫째, 작동하게 만들고, 둘째, 올바로 만들고, 셋째, 빠르게 만들도록 하라.

프로파일링은 무엇을 의미하는가?

애플리케이션의 프로파일링은 제어된 프로그램 실행으로부터 동적 정보를 얻음을 나타낼 때 자주 사용되는 용어다. 수집된 정보는 프로그램 실행을 향상시키는 방법에 관한 중요한 힌트를 제공하는 데에 목적이 있다. 이러한 힌트는 주로 수치상 측정(numerical measurements)으로서, 프로그래밍 실행을 서로 쉽게 비교할 수 있다.

이번 장에서는 메서드 실행 시간과 메모리 소모량에 관련된 측정을 고려할 것이다. 프로그래밍 실행에서 다른 유형의 정보도 추출이 가능하며, 특히 메서드 호출 그래프를 예로 들 수 있겠다.

프로그램 실행이 주로 보편적인 80-20 규칙을 따르기 때문에 총 메서드 중에서 소량만이 (20%라고 해보자) 이용 가능한 자원에서 가장 큰 부분을 (메모리와 CPU 소모의 80%) 소모한다는 관찰은 참 흥미롭다. 애플리케이션의 최적화는 따라서 기본적으로 균형(tradeoff)의 문제다. 이번 장에서는 이러한 20%의 메서드를 재빠르게 확인하는 데에 이용 가능한 툴의 사용법을 살펴보고, 우리가 제시한 프로그램 개선에 따른 경과를 측정하는 방법도 살펴볼 것이다.

경험상 프로그램의 최적화 시 프로그램 구문을 망가뜨리지 않도록 확보하는 데에 있어 단위 시험(unit test)은 필수적이다. 기존 알고리즘이 다른 것으로 대체되더라도 프로그램은 본래의

기능을 수행하도록 확보해야 한다.

간단한 예제

메서드 Collection≫select:thenCollect:를 고려해보자. 주어진 컬렉션에서 해당 메서드는 술부(predicate)를 이용해 요소를 선택한다. 이후 선택된 요소마다 블록 함수를 적용한다. 언뜻 보면 이러한 행위는 컬렉션에 2회의 run을 의미하는 것처럼 보여, 하나는 select:thenCollect:의 사용자가 제공하고, 나머지 하나는 선택된 요소들을 포함하는 중간 컬렉션(intermediate)처럼 보인다. 하지만 중간 컬렉션이 꼭 필요한 것은 아닌데, 선택과 함수 애플리케이션은 한 번의 run으로도 실행 가능하기 때문이다.

timeToRun 메서드. 하나의 프로그램 실행을 프로파일링하는 것만으로는 최적화되어야 하는 내용을 완전히 식별하고 확인하기엔 충분치 않다. 프로파일된 실행을 최소 두 개를 비교하는 것은 매우 유익하다. TimeToRun 메시지를 bloc으로 전송하여 블록을 평가하는 데에 소요된 시간을 밀리 초로 얻을 수도 있다. 의미 있고 표현적인 측정을 갖기 위해서는 루프를 이용해 프로파일링을 "증폭"시킬 필요가 있겠다.

결과를 조금만 소개하겠다.

```
| coll |
coll := \#(1 2 3 4 5 6 7 8) asOrderedCollection.
[100000 timesRepeat: [ (coll select: [:each | each > 5]) collect: [:i |i\(\ast{}\)i]]] timeToRun
"Calling select:, then collect: -- −→ ∼ 570 -- 600 ms"

| coll |
coll := \#(1 2 3 4 5 6 7 8) asOrderedCollection.
[100000 timesRepeat: [ coll select: [:each | each > 5] thenCollect:[:i |i\(\ast{}\)i]]] timeToRun
"Calling select:thenCollect: -- −→ ∼ 397 -- 415 ms"
```

두 실행 간 차이는 몇 백 밀리초에 불과하지만 둘 중 하나는 애플리케이션 속도를 상당히 저하시킬 수 있다!

select:thenCollect:의 정의를 자세히 살펴보자. 최적화되지 않은 원본 구현은 Collection에서 찾을 수 있다. (Collection은 Pharo 컬렉션 라이브러리의 루트 클래스임을 명시하라.) 좀 더 효율적인 구현은 해당 연산을 효율적으로 실행하기 위해 정렬된 컬렉션의 구조를 고려하는 OrderedCollection에서 정의된다.

```
Collection>>select: selectBlock thenCollect: collectBlock
    "Utility method to improve readability."

    ^ (self select: selectBlock) collect: collectBlock

OrderedCollection>>select: selectBlock thenCollect: collectBlock
    " Utility method to improve readability.
    Do not create the intermediate collection. "

    | newCollection |
    newCollection := self copyEmpty.
    firstIndex to: lastIndex do: [:index |
        | element |
        element := array at: index.
        (selectBlock value: element)
            ifTrue: [ newCollection addLast: (collectBlock value: element) ]].
    ^ newCollection
```

이미 짐작했겠지만 Set나 Dictionary와 같은 다른 컬렉션들은 최적화 버전에서 이익을 얻지 못한다. 다른 추상적 데이터 타입에 대한 효율적인 구현은 각자의 연습을 위해 남겨두겠다. 공동체를 위해 혹시 select:thenCollect: 이나 다른 메서드에서 더 낫고 최적화된 버전이 생각나면 Pharo에 연락할 것을 잊지 말라.

bench 메서드. bench 메시지는 블록으로 전송 시 해당 블록이 초당 평가되는 횟수를 추정한다. 예를 들어, 표현식 [1000 factorial] bench 는 1000 factorial 이 초당 약 350회 실행될 것이라고 말한다.

Pharo 에서 코드 프로파일링하기

timeToRun 메서드는 표현식이 실행되는 데에 소요되는 시간을 알려주는 데 유용하다. 하지만 표현식을 평가하여 트리거된 계산에 실행 시간이 어떻게 분배되는지를 이해하기엔 적절하지 않다. Pharo에는 MessageTally라는 코드 프로파일러가 제공되어 계산에 대한 시간 분배를 정밀하게 분석하도록 해준다.

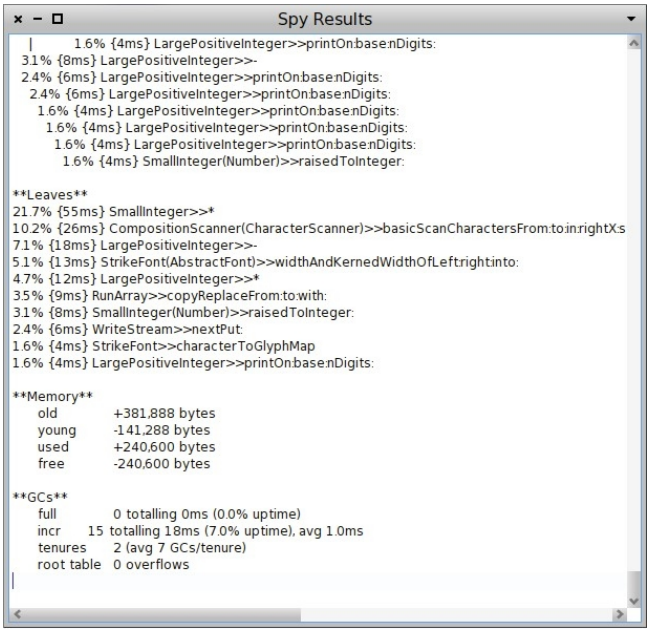


그림 17.1: 실행 중인 MessageTally.

MessageTally

MessageTally는 동일한 이름을 가진 유일한 클래스로서 구현된다. 이는 사용법이 꽤 간단하다. 세부적인 실행 분석을 얻기 위해선 블록 표현식을 인자로 한 spyOn: 메시지를 MessageTally로 전송하면 된다. MessageTally spyOn: ["your expression here"]를 평가하면 아래 정보가 포함된 창이 열린다.

1. 표현식이 실행되는 동안 실행된 메서드와 함께 연관된 실행 시간을 표시하는 계층구조 리스트.
2. 실행의 leaf 메서드. Leaf 메서드는 다른 메서드를 (예: 프리미티브, 접근자) 호출하지 않는 메서드다.
3. 메모리 소모량이나 쓰레기 수집기 관계 여부를 알려주는 통계.

요점은 후에 하나씩 설명하겠다.

그림 17.2는 MessageTally spyOn: [20 timesRepeat: [Transcript show: 1000 factorial printString]] 표현식 결과를 보여준다. 메시지 spyOn: 는 제공된 블록을 새 프로세스에서 실행한다. 분석은 하나의 프로세스, 즉 프로파일링할 블록을 실행하는 프로세스에만 집중한다. SpyAllOn: 메시지는 실행 도중에 활성화된 모든 프로세스를 프로파일한다. 이는 여러 프로세스 상에서 이루어지는 계산 분배를 분석하는 데에 유용하다.

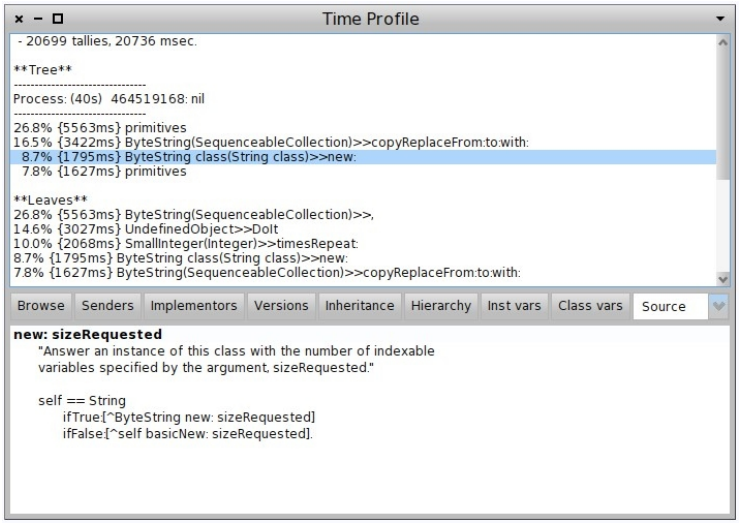


그림 17.2: MessageTally를 이용해 실행된 메서드를 탐색하는 TimeProfiler.

MessageTally보다 조금 더 간단한 툴로 TimeProfileBrowser가 있다. 이는 실행된 메서드의 구현도 표시한다 (그림 17.2). TimeProfileBrowser는 spyOn: 메시지를 이해한다. 즉, 아래 소스 코드에서 MessageTally를 TimeProfileBrowser로 대체하면 더 나은 사용자 인터페이스를 얻을 수 있다는 뜻이다.

프로그래밍 환경으로 통합

앞서 살펴봤듯이 프로파일러는 MessageTally 클래스로 spyOn: 와 spyAllOn: 을 전송하면 직접 호출할 수 있다. 그 외에도 여러 방법들을 통해 접근이 가능하다.

World 메뉴를 통해. World 메뉴(Pharo 창 밖을 클릭)는 System 하위메뉴를 통해 몇 가지 프로파일링 기능을 제공한다 (그림 17.3). Start profiling all processes는 텍스트 선택내용으로부터 블록을 생성하여 spyAllOn: 을 호출한다. Start profiling UI 엔트리를 선택하면 사용자 인터페이스 프로세스를 프로파일링한다. 이는 사용자 인터페이스를 디버깅 시 매우 유용하다!

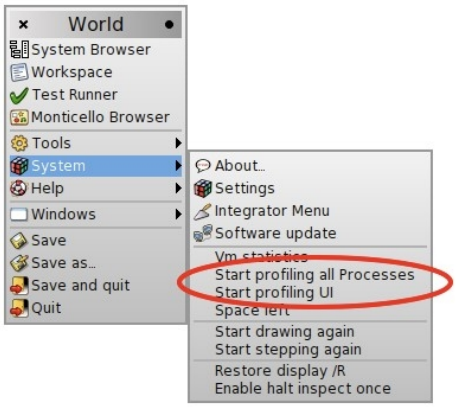


그림 17.3: 메뉴를 통한 접근.

Test Runner를 통해. 애플리케이션 크기가 증가하면서 단위 시험은 코드 프로파일링에 적합한 후보자가 되는 것이 보통이다. 테스트 실행 시간이 너무 길어질 경우 테스트의 실행은 지루해진다. Pharo의 Test Runner에는 Run Profiled 버튼이 제공된다 (그림 17.4).

이 버튼을 누르면 선택된 단위 시험을 실행하고 메시지 tally 보고서를 생성한다.

결과를 읽고 해석하기

메시지 tally 프로파일러는 기본적으로 두 가지 정보를 제공한다.

- 실행 시간은 프로파일링된 코드 실행을 나타내는 트리를 이용해 표시된다 (**Tree**). 해당 트리의 각 노드에는 각 leaf 메서드에서 소비된 시간이 표시된다 (**Leaves**).
- 메모리 활동은 메모리 소모량과 (**Memory**) 쓰레기 수집기 사용을 (**GC**) 포함한다.

설명을 위해 다음 시나리오를 살펴보겠다. 문자열 'A'는 초기의 빈 문자열에 9000번 누적으로 추가된다.

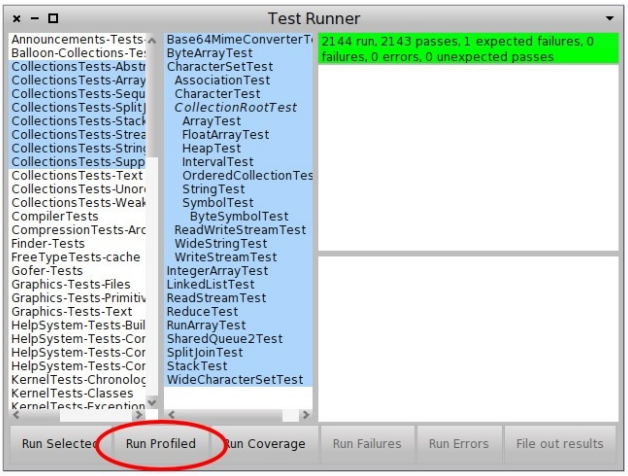


그림 17.4: TestRunner에 Tally 메시지를 생성하기 위한 버튼.

```
MessageTally spyOn:
    [ 500 timesRepeat: [
        | str |
        str := ''.
        9000 timesRepeat: [ str := str, 'A' ]]].
```

완전한 결과는 아래와 같다.

```
-- 24038 tallies, 24081 msec.

\begin{itemize}
\item \begin{itemize}
\item Tree\(\ast{}\)\(\ast{}\)
\end{itemize}
\end{itemize}
--------------------------------
Process: (40s) 535298048: nil
--------------------------------
29.7\% \{7152ms\} primitives
11.5\% \{2759ms\} ByteString(SequenceableCollection)>>copyReplaceFrom:to:with:
5.9\% \{1410ms\} primitives
5.6\% \{1349ms\} ByteString class(String class)>>new:

\begin{itemize}
\item \begin{itemize}
\item Leaves\(\ast{}\)\(\ast{}\)
\end{itemize}
\end{itemize}
  29.7\% \{7152ms\} ByteString(SequenceableCollection)>>,
  9.2\% \{2226ms\} SmallInteger(Integer)>>timesRepeat:
  5.9\% \{1410ms\} ByteString(SequenceableCollection)>>copyReplaceFrom:to:with:
  5.6\% \{1349ms\} ByteString class(String class)>>new:
  4.4\% \{1052ms\} UndefinedObject>>DoIt

\begin{itemize}
\item \begin{itemize}
\item Memory\(\ast{}\)\(\ast{}\)
\end{itemize}
\end{itemize}
  old +0 bytes
  young +9,536 bytes
  used +9,536 bytes
  free -9,536 bytes

\begin{itemize}
\item \begin{itemize}
\item GCs\(\ast{}\)\(\ast{}\)
\end{itemize}
\end{itemize}
  full 0 totalling 0ms (0.0\% uptime)
  incr 9707 totalling 7,985ms (16.0\% uptime), avg 1.0ms
  tenures 0
  root table 0 overflows
```

첫 행은 전체 실행 시간과 샘플링 개수(tally라고도 불리는데 샘플링에 관한 내용은 본 장의 끝 부분에서 다시 설명할 것이다)를 제공한다.

****Tree**: 누계(Cumulative) 정보**

Tree 부분은 프로세스당 실행 트리를 나타낸다. 트리는 Pharo 해석기가 각 메서드에서 소비한 시간을 알려준다. 뿐만 아니라 호출 그래프를 이용해 여러 호출을 알려주기도 한다. 서로 다른 실행 흐름은 그들이 실행된 프로세스에 따라 구분된 채 유지된다. 프로세스 우선순위도 표시되어 서로 다른 프로세스를 구별하는 데에 도움이 된다. 예제는 아래와 같이 알려준다.

```
\begin{itemize}
\item \begin{itemize}
\item Tree\(\ast{}\)\(\ast{}\)
\end{itemize}
\end{itemize}
--------------------------------
Process: (40s) 535298048: nil
--------------------------------
29.7\% \{7152ms\} primitives
11.5\% \{2759ms\} ByteString(SequenceableCollection)>>copyReplaceFrom:to:with:
5.9\% \{1410ms\} primitives
5.6\% \{1349ms\} ByteString class(String class)>>new:
```

해당 트리는 해석기가 총 실행 시간 중 29.7%를 프리미티브의 실행에 소비하였음을 보여준다. 총 실행 시간 중에서 11.5%는 SequenceableCollection≫copyReplaceFrom:to:with:메서드에 소비되었다. 해당 메서드는 comma(,) 메시지를 이용해 문자열을 연결(concatenate)할 때 호출되는데, 콤마(,) 자체는 new:와 몇몇 가상 머신 프리미티브를 간접적으로 호출한다.

실행(execution)에는 총 실행 시간의 11.5%가 소비되었는데, 이는 해석기의 결과가 다른 프로세스와 공유됨을 의미한다. 코드부터 프리미티브로 연결된 호출 사슬은 상대적으로 짧다. 해당 예제는 후에 최적화할 것이다.

두 개의 프리미티브가 트리 leaf로서 열거된다. 이들은 서로 다른 프리미티브에 해당한다. 불행히도 MessageTally는 둘 중 어떤 프리미티브가 호출되었는지 알려주지 않는다.

Leaves: leaf 메서드

Leaves 부분은 프로파일링된 코드 블록에 대한 leaf 메서드를 열거한다. Leaf 메서드는 다른 메서드를 호출하지 않는 메서드를 의미한다. 좀 더 정확히 말하자면 이것은 메서드 m으로서, 메서드 m이 호출한 메서드는 어떤 것도 "감지되지" 않는다. 이는 변수 접근자(예: Point≫), 프리미티브 메서드, 매우 빠르게 실행되는 메서드의 경우에 해당한다. 앞의 예제에서 우리는 아래를 얻는다.

```
\begin{itemize}
\item \begin{itemize}
\item Leaves\(\ast{}\)\(\ast{}\)
\end{itemize}
\end{itemize}
29.7\% \{7152ms\} ByteString(SequenceableCollection)>>,
9.2\% \{2226ms\} SmallInteger(Integer)>>timesRepeat:
5.9\% \{1410ms\} ByteString(SequenceableCollection)>>copyReplaceFrom:to:with:
5.6\% \{1349ms\} ByteString class(String class)>>new:
4.4\% \{1052ms\} UndefinedObject>>DoIt
```

Memory

메모리 소모량에 관한 통계 부분을 보면 할당된 메모리 양과 쓰레기 수집기 사용에서 관찰된 변경 사항을 알려준다. 이러한 정보를 완전히 이해하려면, Pharo의 쓰레기 수집기(GC)는 scavenging GC여서, 오래된 객체가 심지어 더 오래 생존하기 위해 큰 변화를 갖는다는 원칙에 의존할 필요가 있다. 이는 객체가 향후에 참조된 채로 유지될 것이란 사실에 따라 설계된다. 반면 young 객체도 빠르게 참조해제(dereference)될 변경 내용을 상당히 많이 갖는다.

몇몇 메모리 존(zone)이 고려되며, 새로운 객체가 오래된 객체 전용의 공간으로 이동하면 tenured될 자격이 있다 (아래 미국 학술 과학자들에 대한 비유에 따라 정규직을 얻게 되면 그러한 자격이 생긴다).

MessageTally를 이용해 실현된 메모리 분석의 예제는 다음과 같다.

```
\begin{itemize}
\item \begin{itemize}
\item Memory\(\ast{}\)\(\ast{}\)
\end{itemize}
\end{itemize}
old       +0 bytes
young     +9,536 bytes
used    +9,536 bytes
free    -9,536 bytes
```

MessageTally는 네 가지 값을 이용해 메모리 사용을 설명한다.

1. old 값은 old 객체 전용의 메모리 공간 증가와 관련이 있다. 객체는 그 물리적 메모리 위치가 "오래된 메모리 공간"에 있을 때 "오래되었다"는 용어에 적격하다. 이는 전체 쓰레기 수집기가 트리거되거나, 너무 많은 객체 생존자들이 있는 경우(가상 머신에 명시된 몇몇 한계선에 따라) 발생한다. 이러한 메모리 공간은 전체적인 쓰레기 수집에 의해서만 청소된다. (따라서 점진적 GC는 그 크기를 축소시키지 않는다.)
   오래된 메모리 공간이 증가하는 이유는 메모리 누수 때문일 가능성이 크다. 가상 머신은 메모리를 해제(release)할 수 없었고, young 객체들을 old 객체로 발전시켰다.

2. young 값은 young 객체 전용의 메모리 공간의 증가를 알려준다. 객체가 생성되면 물리적으로 해당 메모리 공간에 위치한다. 해당 메모리 공간의 크기는 수시로 변경된다.

3. used 값은 총 사용된 메모리량이다.

4. free 값은 이용 가능한 메모리량이다.

우리 예제에서는 실행 중 생성된 어떤 객체도 old 객체로 발전하지 않았다. 9 536 바이트가 현재 프로세스에서 사용되었고, young 메모리 공간에 위치하였다. 이용 가능한 메모리량은 그에 따라 감소되었다.

GCs

GCs는 쓰레기 수집기에 관한 통계를 제공한다. 쓰레기 수집기 보고서의 예를 들자면 아래와 같다.

```
\begin{itemize}
\item \begin{itemize}
\item GCs\(\ast{}\)\(\ast{}\)
\end{itemize}
\end{itemize}
full        0 totalling 0ms (0.0\% uptime)
incr      9707 totalling 7,985ms (16.0\% uptime), avg 1.0ms
tenures      1 (avg 9707 GCs/tenure)
root table    0 overflows
```

네 가지 값이 이용 가능하다.

1. full 값은 전체 쓰레기 수집 양과 그에 소요된 시간량을 합계 낸다. 전체 쓰레기 수집은 그다지 자주 발생하지 않는다. 이는 큰 메모리 청크를 자주 할당한 결과이다.

2. incr는 점진적(incremental0 쓰레기 수집에 관한 것이다. 점진적 GC는 자주 발생하고 (초당 몇 회) 빠르게 실행되는 (약 1 또는 2ms) 것이 보통이다. 점진적 GC에 소요된 시간은 10% 미만으로 유지하는 것이 바람직하다.

3. tenures 개수는 old 메모리 공간으로 이동한 객체량을 말해준다. 이러한 이동은 young 메모리 공간의 크기가 주어진 한계치보다 위에 있을 때 발생한다. 이는 보통 애플리케이션을 시작할 때 발생하고, 모든 필수 객체가 생성되지 않았거나 참조되지 않았을 때 발생한다.

4. root table overflows는 쓰레기 수집기가 이미지를 탐색 시 사용하는 루트 객체의 양이다. 이러한 탐색은 시스템에 메모리가 부족하여 향후 프로그램 실행에 관련된 모든 객체를 수집할 필요가 있을 때 발생한다. Overflow 값은 점진적 GC에 의해 사용되는 루트 개수가 그 내부 테이블보다 큰 경우처럼 드문 상황을 식별한다. 이러한 상황이 발생하면 GC로 하여금 일부 객체를 강제로 tenured 상태로 만든다.

예제에서는 점진적 GC만 사용됨을 확인할 수 있다. 잇따라 살펴보겠지만 생성된 객체의 양은 성능을 최적화하고자 할 때와 관련이 있다.

실례를 이용한 분석

프로파일링 시 얻은 결과를 이해하는 것은 애플리케이션을 최적화하고자 할 때 취해야 하는 첫 번째 단계다. 하지만 지금쯤 느끼겠지만 계산이 번거로운 이유를 이해하는 일도 간과해서는 안 된다. 많은 예제를 바탕으로 우리는 여러 프로파일링 결과를 비교하는 것이 번거로운 메시지 호출을 식별하는 데에 어떻게 도움이 되는지 살펴볼 것이다.

",” 메서드는 새 문자열을 생성하고 수신자와 인자를 모두 그 문자열로 복사하기 때문에 속도가 느린 것으로 알려져 있다. Stream을 사용하면 문자열을 연결하기가 훨씬 빠르다. 하지만 nextPut:과 nextPutAll: 은 사용 시 주의를 기울여야 한다!

문자열 연결에 Stream 사용하기. 언뜻 보면 스트림은 주로 상대적으로 느린 입/출력과(예: 네트워크 소켓, 디스크 접근, Transcript) 함께 사용되기 때문에 스트림을 생성하기가 번거롭다고 생각할 수도 있다. 하지만 앞의 예제에 사용된 문자열을 스트림 연산으로 대체하면 약 10배가 빨라진다! 문자열을 9000회 연결하면 8999개의 중간 객체를 생성하고, 각 객체는 다른 객체의 내용으로 채워져 있다고 생각하면 이해가 쉬울 것이다. 스트림을 이용하면 각 반복(iteration)에서 문자를 추가해야 한다.

```
MessageTally spyOn:
    [ 500 timesRepeat: [
        | str |
        str := WriteStream on: (String new).
        9000 timesRepeat: [ str nextPut: $A ]]].
```

```
-- 807 tallies, 807 msec.

\begin{itemize}
\item \begin{itemize}
\item Tree\(\ast{}\)\(\ast{}\)
\end{itemize}
\end{itemize}
--------------------------------
Process: (40s) 535298048: nil
--------------------------------

\begin{itemize}
\item \begin{itemize}
\item Leaves\(\ast{}\)\(\ast{}\)
\end{itemize}
\end{itemize}
33.0\% \{266ms\} SmallInteger(Integer)>>timesRepeat:
21.2\% \{171ms\} UndefinedObject>>DoIt

\begin{itemize}
\item \begin{itemize}
\item Memory\(\ast{}\)\(\ast{}\)
\end{itemize}
\end{itemize}
old       +0 bytes
young     -18,272 bytes
used    -18,272 bytes
free    +18,272 bytes

\begin{itemize}
\item \begin{itemize}
\item GCs\(\ast{}\)\(\ast{}\)
\end{itemize}
\end{itemize}
full      0 totalling 0ms (0.0\% uptime)
incr    5 totalling 7ms (3.0\% uptime), avg 1.0ms
tenures   0
root table  0 overflows
```

문자열 사전할당. 컬렉션의 사전할당 없이 OrderedCollection을 이용하는 데에는 수고가 많이 드는 것으로 알려져 있다. 컬렉션이 가득 찰 때마다 그 내용은 더 큰 컬렉션으로 복사되어야 한다. 주의 깊게 선택한 사전할당은 정렬된 컬렉션을 이용하는 효과를 갖는다. new 메시지 대신 new: aNumber 메시지를 사용할 수 있다.

```
MessageTally spyOn:
    [ 500 timesRepeat: [
        | str |
        str := WriteStream on: (String new: 9000).
        9000 timesRepeat: [ str nextPut: $A ]]].
```

이 예제에서는 atAllPut: 메서드를 이용하여 스크립트를 개선하는 것이 가능하다. 아래 스크립트에 소요되는 시간은 몇 밀리초에 불과하다.

```
MessageTally spyOn:
    [ 500 timesRepeat: [
        | str |
        str :=String new: 9000.
        str atAllPut: $A ]].
```

실험. 벤치마크를 실행하는 것은 서로 다른 실행을 비교할 때 그 빛을 발한다. 앞의 코드 조각에서 값 9000을 500으로 대체하면 매우 유익하다. 9000 회 반복에 소요된 시간은 500에 소요된 시간보다 2.6배 정도 느리다. 스트림 대신 문자열 연결(예: , 메서드를 이용해)을 이용하면 factor 10과의 차이를 넓힐 수 있다. 이러한 실험은 문자열의 연결에 적절한 툴을 이용하는 중요성을 분명히 보여준다.

프로파일링된 실행 시간 또한 결과의 중요한 질적 요인이 된다. MessageTally는 코드를 프로파일에 샘플링 기법을 사용한다. default마다 MessageTally는 default별로 현재 실행되는 스레드(thread)를 샘플링한다. 따라서 계산에 관여된 모든 메서드가 결과 보고서에 나타나기 위해 "적정한" 시간만큼 실행된다. 프로파일링할 애플리케이션이 너무 짧은 경우(몇 밀리초에 불과할 경우) 여러 번 실행하면 보고서의 정확성을 향상시키는 데에 도움이 된다.

메시지 계수하기

지금까지 알아본 프로파일링은 메서드 실행 시간에 초점을 둔다. 메서드 호출 스택 샘플링의 이점으로는 실행에 상대적으로 적은 영향을 미친다는 점이 있다. 단점은 결과가 상대적으로 정밀하지 못하다는 점이다. 대부분의 경우, 이로 얻은 결과로 충분하지만 어찌되었든 항상 실제 실행의 근사치(approximation)에 해당한다.

MessageTally는 프로그램 해석을 기반으로 한 프로파일링을 허용한다. 이는 실행 샘플러 대신 바이트코드 해석기를 사용하는 개념이다. 주요 장점으로 결과의 정확도를 들 수 있다. tallySends: 메시지로 얻은 정보는 계산에 수반된 각 메서드가 실행된 시간을 나타낸다. 그림 17.5는 아래를 실행해 얻은 결과를 제시한다.

```
MessageTally tallySends:[1000 timesRepeat: [3.14159 printString]].
```

TallySends: 의 단점은 제공된 블록을 실행하는 데에 소요되는 시간을 들 수 있다. 프로파일링할 블록은 Pharo에서 작성된 해석기에 의해 실행되는데, 이는 가상 머신의 해석기보다 속도가 느리다. tallySends: 에 의해 프로파일링되는 코드 조각은 200배 가량이 느리다. 해석기는 ContextPart≫runSimulated: aBlock contextAtEachStep: block2 메서드로부터 이용 가능하다.

기억된 피보나치

앞서 살펴본 기법들을 약간 적용하기 위해 피보나치 함수 function (fib(n) = fib(n – 1) + fib(n – 2) with fib(0) = 0, fib(1) = 1)를 고려해보자. 우리는 두 가지 버전을 살펴볼 것인데, 하나는 재귀적 버전이고, 하나는기억된 버전이다. 저장하기(memoizing)는 중복 계산을 피하기 위해 캐시를 도입하는 특징이 있다.

수학적 정의에 가까운 아래 정의를 고려해보자.

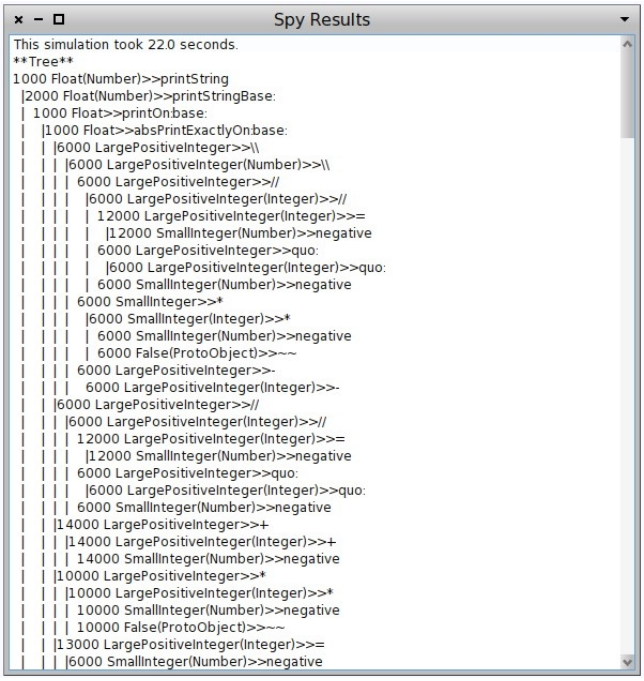


그림 17.5: 실행(execution) 중에 실행된 모든 메시지.

```
Integer>>fibSlow
    self assert: self >= 0.
    (self <= 1) ifTrue: [ ^ self].
    ^ (self - 1) fibSlow + (self - 2) fibSlow
```

fibSlow 메서드는 상대적으로 비효율적이다. 각 재귀는 계산의 중복을 나타낸다. 재귀의 각 branch에 의해 동일한 결과가 두 번 계산된다.

좀 더 효율적인 (하지만 약간은 더 복잡한) 버전은 중간(intermediary) 계산 값을 보관하는 캐시를 이용하여 얻는다. 이의 장점은 각 값이 한 번만 계산되기 때문에 계산이 중복되지 않는다는 데에 있다. 이러한 전통적인 프로그램 최적화 방식을 저장하기(memoizing)라 부른다.

```
Integer>>fib
    ^ self fibWithCache: (Array new: self)

Integer>>fibLookup: cache
    | res |
    res := cache at: (self + 1).
    ^ res ifNil: [ cache at: (self + 1) put: (self fibWithCache: cache ) ]

Integer>>fibWithCache: cache
    (self <= 1) ifTrue: [ ^ self].
    ^ ((self - 1) fibLookup: cache) + ((self - 2) fibLookup: cache)
```

실습으로 저장하기 방식의 장점을 확실히 살펴보도록 35 fibSlow와 35fib를 프로파일링하라.

Class별 메모리 소모량에 대한 SpaceTally

주어진 클래스의 메모리 소모량과 인스턴스의 개수를 아는 것은 때때로 중요하다. SpaceTally 클래스가 바로 이러한 기능을 제공한다.

SpaceTally newprintSpaceAnalysis 표현식은 시스템의 모든 클래스를 훑어보면서 각 클래스마다 코드 크기, 인스턴스의 개수, 인스턴스가 차지하는 총 메모리 공간을 수집한다. 결과는 인스턴스가 차지하는 총 메모리 공간에 따라 정렬되고, Pharo 이미지 바로 옆에 위치한 STspace.text라는 파일에 보관된다.

문자열, 컴파일된 메서드와 비트맵이 Pharo의 메모리에서 가장 큰 부분을 표현한다는 사실은 놀라운 일이 아니다. 다른 플랫폼에서는 다양한 애플리케이션을 위해 컴파일된 코드, 문자열, 비트맵이 차지하는 비율을 찾을 수 있을 것이다.

SpaceTally의 출력은 아래와 같이 구조화된다.

```
Class   code space  # instances   inst space  percent   inst average size
ByteString  2053  109613  9133154   31.20   83.32
Bitmap  3653  379   6122156   20.90   16153.45
CompiledMethod  20067   51579   3307151   11.30   64.12
Array   2535  85560   3071680   10.50   35.90
ByteSymbol  1086  35746   914367  3.10  25.58
```

각 행은 Pharo 클래스의 메모리 분석을 나타낸다. 클래스는 그들이 차지하는 공간에 따라 정렬된다. ByteString 클래스는 문자열을 설명하는데, 이는 종종 메모리의 1/3을 소모하기 위한 문자열을 갖는다. 코드 공간은 클래스와 그 메타클래스가 사용하는 바이트량을 제공한다. 클래스 변수에 의해 사용되는 공간은 포함하지 않는다. 그 값은 Behavior≫spaceUsed 메서드에 의해 주어진다.

instances 열은 인스턴스의 양을 제공한다. 이는 Behavior≫instanceCount의 결과이다. Inst space 열은 모든 인스턴스에 의해 소모되는 바이트량을 의미하는데, 객체 헤더도 이에 포함된다. 이는 Behavior≫instancesSizeInMemory의 결과다. 메모리 차지 비율은 percent 열에서 주어지고, 마지막 열은 인스턴스의 평균 크기를 제시한다.

모든 클래스 상에서 SpaceTally를 실행하는 데에는 몇 분의 시간이 소요된다. SpaceTally는 분석 시간을 증가시키기 위해 축소된 클래스 집합에서 실행되기도 한다. 아래를 고려해보자.

```
((SpaceTally new spaceTally: (Array with: TextMorph with: Point))
    asSortedCollection: [:a :b | a spaceForInstances > b spaceForInstances])
```

SpaceTally≫spaceTally: 메서드는 그 인자인 각 클래스가 소모하는 메모리를 분석한다. SpaceTallyItem 의 인스턴스 리스트를 리턴한다.

몇 가지 조언

프로그램을 측정하고 최적화하는 다수의 전략을 살펴보았다. 본문에 사용된 예제들은 상대적으로 규모가 작다. 프로그램의 최적화가 항상 쉬운 작업은 아니다. 캐시를 삽입하기 위한 메서드 후보(candidate)를 식별하는 일은 단순하면서 효율적인데, 단 (i) 캐시를 무효화하는 방법을 숙지하고, (ii) 코드를 삽입 시 전체 실행에 미치는 영향을 인식할 때에 한해서다.

일반적으로 leaf 메서드의 최적화를 시도하는 대신 전체 알고리즘을 이해하는 편이 더 중요하다. 데이터가 구조화되는 방식 또한 최적화의 기회를 제공할지도 모른다. 예를 들어 정렬 컬렉션이나 연계된(linked) 리스트는 비순환 그래프를 나타낼 때 사용하기엔 적절하지 않을지도 모른다. 해시값이 적절하게 잘 분배된 경우 dictionary 또는 집합을 이용하면 더 나은 성과를 제공할 수도 있다.

메모리 소모량은 중요한 역할을 수행하기도 한다. 때때로 쓰레기 수집이 요청된다면 전체적인 성능은 크게 감소할지도 모른다. 객체를 재활용하고 불필요한 객체 생성을 피하면 쓰레기 수집기의 요청을 감소시키는 데에 도움이 된다.

MessageTally는 어떻게 구현되는가?

MessageTally는 Pharo의 반영적인 기능을 사용하는 방식을 보여주는 뛰어난 예에 해당한다. SpyEvery: millisecs on: aBlock 메서드는 전체적인 프로파일링 로직을 포함한다. 이 메서드는 spyOn: 에 의해 간접적으로 호출된다. millisecs 값은 각 샘플 간 밀리초를 나타내며, default 별로 1에 설정된다. 프로파일링될 블록은 aBlock이다.

프로파일링 활동의 핵심은 아래에 발췌된 코드 조각에서 주어진다.

```
observedProcess := Processor activeProcess.
Timer := [
    [ true ] whileTrue: [
        | startTime |
        startTime := Time millisecondClockValue.
        myDelay wait.
        self
            tally: Processor preemptedProcess suspendedContext
            in: (observedProcess == Processor preemptedProcess
                ifTrue: [ observedProcess ] ifFalse: [ nil ])
            by: (Time millisecondClockValue - startTime) // millisecs ].
    nil] newProcess.
Timer priority: Processor timingPriority-1.
```

Timer는 우선순위가 높게 설정된 새 프로세스로, aBlock의 모니터링을 책임진다. 따라서 프로세스 스케줄러가 순조롭게 활성화할 것이다 (timingPriority는 시스템 프로세스의 프로세스 우선순위다). 이것은 메서드 콜 스택의 스냅샷 이전에 필요한 밀리초(myDelay) 만큼 기다리는 무한 루프를 생성한다. 관찰해야 할 프로세스는 observedProcess로, 이는 spyEvery: millisecs on: aBlock 메시지가 전송된 프로세스다.

프로파일링을 하는 목적은 각 메서드 컨텍스트와 카운터를 연관시키는 데에 있다. 이러한 연관은 MessageTally 클래스의 (해당 클래스는 class, method, process 변수를 정의한다) 인스턴스를 이용해 얻는다.

규칙적 간격으로 (myDelay) 각 스택 프레임의 카운터는 지연된(elapsed) 밀리초만큼 증가된다. 스택 프레임은 방금 선점된(preempted) 프로세스로 suspendedContext를 전송하면 얻을 수 있다.

tally: context in: aProcess by: count 메서드는 count에 의해 주어진 밀리초만큼 각 스택 프레임을 증가시킨다.

메모리 통계는 소모된 메모리량을 프로파일링 전과 후로 나누어 제공된다. SmalltalkImage 클래스의 인스턴스인 Smalltalk는 이용 가능한 메모리량을 질의하기 위한 다수의 접근(accessing) 메서드를 포함한다.

요약

이번 장을 통해 우리는 Pharo에서 프로파일링의 기본을 살펴보았다. 본 장에서는 MessageTally의 기능을 제시하였고, 성능 병목현상을 다시 흡수하기 위한 원칙을 여러 가지 소개하였다.

- timeToRun과 bench 메서드는 단순한 벤치마킹을 제공하며, 블록으로 전송되어야 한다.
- MessageTally는 샘플링 기반의 코드 프로파일러이다.
- MessageTally spyOn: ["an expression"]을 평가하면 제공된 블록을 평가하고 보고서를 표시한다.
- 정확도는 프로파일링된 블록의 실행 시간을 증가시켜 달성할 수 있다.
- Pharo 프로그래밍 환경은 여러 가지 편리한 프로파일링 방식을 제공한다.
- 메시지를 계수하는 것은 느리지만 정확한 프로파일링 기법이다.
- 저장하기(memoization)는 실행을 증가시키는 일반적이면서 효율적인 코드 패턴이다.
- SpaceTally는 메모리 소모량에 관해 보고한다.

제 18 장

PetitParser: Modular 파서 빌드하기

작성자:
Jan Kurs *(kurs@iam.unibe.ch)*
Guillaume Larcheveque *(guillaume.larcheveque@gmail.com)*
Lukas Renggli *(renggli@gmail.com)*

데이터를 분석하고 변형하기 위해 파서를 빌드하는 것은 소프트웨어 개발에서 공통적인 작업에 해당한다. 따라서 이번 장에서는 PetitParser라는 강력한 파서 프레임워크를 제시하고자 한다. PetitParser는 다양한 파싱 기술로부터 아이디어를 결합하여 문법과 파서를 객체로서 모델링하여 동적으로 재설정(reconfigured) 할 수 있도록 한다. PetitParser는 Lukas Renggli가 Helvetia 시스템[1]을 작업하면서 그 일부로 작성하였으나 독립적인 라이브러리로 사용할 수도 있다.

PetitParser를 이용해 파서 작성하기

PetitParser는 다른 유명한 파서 생성기와는 다른 파싱 프레임워크다. PetitParser는 스몰토크 코드를 이용해 파서를 정의하고 문법을 동적으로 재사용, 작성, 변형, 확장하기 수월하게 만든다. 우리는 이를 결과적 문법에 반영하고 상황에 따라 수정할 수 있다. 따라서 PetitParser는 스몰토크의 동적인 특성에 잘 맞는다.

게다가 PetitP는 SmaCC이나 ANTLR과 달리 테이블을 기반으로 하지 않는다. 대신 네 가지의 대안적 파서 방법론을 결합하여 사용하는데, 이는 scannerless parser, parser combinator, parsing expression grammer, packrat parser가 해당한다. 그러한 PetitParser는 무엇을 파싱할 수 있는지에 더 강력한 기능을 보인다. 네 가지 파서 방법론을 간략하게 살펴보자.

Scannerless parser는 두 개의 독립된 툴(스캐너와 파서)이 하는 일을 하나로 결합한다. 따라

[1] http://scg.unibe.ch/research/helvetia

서 문법을 작성하기 훨씬 간단해지고, 문법을 작성할 때 공통적으로 직면하는 문제를 피하도록 해준다.

Parser Combinator는 구성 가능한 객체의 그래프로서 모델화된 파서에 대한 빌딩 블록이다. 이들은 모듈식으로서 관리가 가능하고(maintainable), 변경, 재구성, 변형, 반영 가능하다.

Parsing expression grammars (PEGs)는 정렬된 선택의 개념을 제공한다. Parser combinator와 달리 PEGs의 정렬된 선택은 항상 첫 번째 매칭 대안책을 따르고 다른 대안들은 무시한다. 유효한 입력의 경우 언제나 정확히 하나의 파스 트리를 야기하고, 파싱의 결과는 결코 모호하지 않다.

Packrat Parsers는 선형의 파싱 시간 보장을 제공하고 PEGs에서의 왼쪽 재귀(left recursion)에 발생하는 공통 문제를 피하도록 해준다.

PetitParser 로딩하기

충분히 설명했으니 본격적으로 시작해보자. PetitParser는 Pharo에서 개발되며, Java와 Dart에 이용 가능한 버전도 있다. 사용할 준비가 된 이미지를 다운로드할 수 있다[2]. PetitParser를 기존 이미지에 로딩하기 위해서는 아래 Gofer 표현식을 평가하라.

스크립트 18.1: PetitParser 설치하기

```
Gofer new
    smalltalkhubUser: 'Moose' project: 'PetitParser';
    package: 'ConfigurationOfPetitParser';
    load.
(Smalltalk at: #ConfigurationOfPetitParser) perform: #loadDefault.
```

PetitParser를 사용하는 방법에 관한 상세한 정보는 Moose book[3]의 petit parser와 관련된 장에서 찾을 수 있다.

간단한 문법 작성하기

PetitParser으로 문법을 작성하기란 스몰토크 코드를 작성하는 것 만큼이나 간단한 일이다. 가령, 문자로 시작해 그 뒤에 0 또는 더 많은 문자나 숫자가 붙은 식별자를 파싱하는 문법은 아래와 같이 정의되고 사용된다.

스크립트 18.2: 식별자를 파싱하기 위한 첫 번째 파서 생성하기.

```
|identifier|
identifier := \#letter asParser , \#word asParser star.
identifier parse: 'a987jlkj' → \#(\$a \#(\$9 \$8 \$7 \$j \$l \$k \$j))
```

[2] https://ci.inria.fr/moose/job/petitparser/
[3] http://www.themoosebook.org/book/internals/petit-parser

그림 18.1: 스크립트 18.2에 정의된 식별자 파서에 대한 구문 해석도(syntax diagram) 표현.

그래픽 표기법

그림 18.1은 식별자 파서의 구문 해석도를 나타낸다. 각 상자마다 하나의 파서를 표시한다. 상자들 간 화살표는 입력이 소모되는 흐름을 나타낸다. 모서리가 둥근 상자는 기본 파서들(terminals)이다. 네모난 상자(그림에 표시되지 않음)는 다른 파서들로 (non-terminal) 구성된 파서들이다.

앞의 스크립트에서 identifier 객체를 살펴보면 이것이 PPSequenceParser의 인스턴스임을 눈치챌 것이다. 해당 객체를 더 깊숙이 살펴보면 여러 파서 객체로 된 아래의 트리를 목격할 것이다.

스크립트 18.3: 식별자 파서에 사용된 파서의 구성.

```
PPSequenceParser (accepts a sequence of parsers)
    PPPredicateObjectParser (accepts a single letter)
    PPPossessiveRepeatingParser (accepts zero or more instances of another parser)
        PPPredicateObjectParser (accepts a single word character)
```

루트 파서는 시퀀스 파서인데 그 이유는 ,(콤마) 연산자가 (1) 문자 파서, (2) 0개 또는 그 이상의 워드(word) 문자 파서에 대한 시퀀스를 생성하기 때문이다. 루트 파서 첫 번째 자식은 #letter asParser 표현식으로 생성된 서술(predicate) 객체 파서이다. 해당 파서는 Character≫isLetter 메서드가 정의한 바와 같이 단일 문자를 파싱할 수 있다. 두 번째 자식은 star 호출에 의해 생성된 반복 파서로서 입력 상에서 그 자식 파서를 (또 다른 서술 객체 파서) 가능한 한 많이 사용한다 (예 : 욕심이 많은 파서이다). 그 자식 파서는 #word asParser 표현식으로 생성된 서술 객체 파서이다. 해당 파서는 Character≫isDigit과 Character≫isLetter 메서드가 정의한 대로 한 자리 숫자 또는 문자의 파싱이 가능하다.

일부 입력 파싱하기

문자열(또는 스트림)을 실제로 파싱하기 위해 아래와 같이 PPParser≫parse:를 이용한다.

스크립트 18.4: 식별자 파서를 이용해 일부 입력 문자열 파싱하기.

```
identifier parse: 'yeah'. → \#(\$y \#(\$e \$a \$h))
identifier parse: 'f123'. → \#(\$f \#(\$1 \$2 \$3))
```

문자를 리턴 값으로 한 중첩 배열을 얻는 것이 이상하게 보이겠지만 이는 입력이 파스 트리로 분해되는 기본적인 모습이다. 맞춤설정하는 방식은 머지않아 살펴보겠다.

유효하지 않은 입력을 파싱할 경우 PPFailure의 인스턴스를 응답으로 얻는다.

스크립트 18.5: 무효한 입력을 파싱 시 실패를 야기한다

```
identifier parse: '123'. → letter expected at 0
```

이러한 파싱은 실패를 야기하는데, 그 이유는 첫 번째 문자(1)가 글자(letter)가 아니기 때문이다. PPFailure의 인스턴스는 당신이 #isPetitFailure 메시지를 전송할 때 true를 응답하는 시스템 내의 객체에 불과하다. 오류가 발생할 경우 예외를 던지기 위해 PPParser≫parse:onError:를 사용할 수도 있다.

```
identifier
    parse: '123'
    onError: [ :msg :pos | self error: msg ].
```

주어진 문자열(또는 스트림)의 매칭 여부에만 관심이 있다면 다음 구조체를 사용해도 좋다.

스크립트 18.6: 일부 입력이 식별자인지 확인하기

```
identifier matches: 'foo'. → true
identifier matches: '123'. → false
identifier matches: 'foo()'. → true
```

마지막 결과를 보고 놀랄지도 모른다. 사실 괄호는 #word asParser 표현식에 명시된 바와 같이 숫자도 아니고 글자도 아니다. 사실상 identifier 파서는 "foo"에 매치하고, 이것만으로 PPParser≫matches: 호출이 true를 리턴하기엔 충분하다. 결과는 #($f #($o$o))를 리턴하게 될 parse:를 사용할 때와 유사하다.

전체 입력이 확실히 매칭하길 원한다면 아래처럼 PPParser≫end 메시지를 사용하라.

스크립트 18.7: PPParser≫end를 이용해 전체 입력이 매칭하도록 하기

```
identifier end matches: 'foo()'. → false
```

PPParser≫end 메시지는 입력의 끝에 미창하는 새로운 파서를 생성한다. 쉽게 파서를 구성할 수 있으려면 기본 값으로 파서가 입력의 끝에 매칭하지 않도록 하는 것이 중요하다. 이러한 점 때문에 당신은 PPParser≫matchesSkipIn:와 PPParser≫matchesIn: 메시지를 이용해 파서가 매칭할 수 있는 모든 장소를 찾는 데에 흥미를 가질지도 모른다.

스크립트 18.8: 입력에서 모든 매치 찾기

```
identifier matchesSkipIn: 'foo 123 bar12'.
    → an OrderedCollection(#($f #($o $o)) #($b #($a $r $1 $2)))

identifier matchesIn: 'foo 123 bar12'.
    → an OrderedCollection(#($f #($o $o)) #($o #($o)) #($o #()) #($b #($a $r $
    1 $2)) #($a #($r $1 $2)) #($r #($1 $2)))
```

PPParser≫matchesSkipIn: 메서드는 매칭한 내용을 포함하는 배열의 컬렉션을 리턴한다. 해당 함수는 동일한 문자를 두 번 파싱하는 것을 피한다. PPParser≫matchesIn: 도 비슷한 일을 하지만 가능한 하위 파싱된(sub-parsed) 요소를 모두 포함한 컬렉션을 리턴하는데, 가령 identifier matchesIn: 'foo 123 bar12'를 평가하면 6개 요소의 컬렉션을 리턴한다.

이와 유사하게 주어진 입력에서 모든 매칭 범위를 (첫 번째 문자의 색인과 마지막 문자의 색인) 찾기 위해서는 PPParser≫matchingSkipRangesIn: 또는 PPParser≫matchingRangesIn: 를 아래 스크립트와 같이 사용할 수 있다.

스크립트 18.9: 입력에 매칭하는 모든 범위 찾기

```
identifier matchingSkipRangesIn: 'foo 123 bar12'.
    → an OrderedCollection((1 to: 3) (9 to: 13))

identifier matchingRangesIn: 'foo 123 bar12'.
    → an OrderedCollection((1 to: 3) (2 to: 3) (3 to: 3) (9 to: 13) (10 to: 13) (11 to: 13))
```

그 외 파서 유형

PetitParser는 복잡한 언어를 소모하거나 임의로 변형하기 위해 구성할 수 있는 즉시 사용 가능한 파서의 거대한 집합을 제공한다. 그 중에서 터미널(terminal) 파서가 가장 간단하다. 이미 몇 가지를 살펴봤고, 몇 가지만 더 프로토콜 Table 18.1에서 정의하겠다.

PPPredicateObjectParser 의 클래스 측은 좀 더 복잡한 terminal 파서를 빌드하는 데에 사용할 수 있는 다른 팩토리(factory) 메서드를 많이 제공한다. 이를 사용하기 위해서는 팩토리 메서드의 이름을 포함한 부호로 PPParser≫asParser 메시지를 전송하라(예: #punctuation asParser).

다음 파서 집합은 다른 파서들을 결합하는 데에 사용되며, 프로토콜 표 18.2 에서 정의된다.

표 18.1: PetitParser 는 다수의 terminal 파서를 사전 정의한다.

terminal 파서	**설명**
$a asParser	$a 문자를 파싱한다.
'abc' asParser	'abc ' 문자열을 파싱한다.
# any asParser	어떤 문자든 파싱한다.
# digit asParser	한 자리 숫자 (0..9) 를 파싱한다.
# letter asParser	하나의 글자를 파싱한다 (a..z 그리고 A..Z).
# word asParser	한 자리 숫자 또는 글자를 파싱한다.
# blank asParser	공백이나 도표 (tabulation) 를 파싱한다.
# newline asParser	carriage return 또는 행 피드 문자를 파싱한다.
# space asParser	새로운 행을 포함해 어떤 흰 공백 문자든 파싱한다.
# tab asParser	탭 문자를 파싱한다.
# lowercase asParser	소문자로 된 문자를 파싱한다.
# uppercase asParser	대문자로 된 문자를 파싱한다.
nil asParser	아무 것도 파싱하지 않는다.

표 18.2: PetitParser는 다수의 parser combinator를 사전 정의한다.

Parser Combinators	**설명**
p1 , p2	p1을 파싱하고 나서 p2를 파싱한다 (시퀀스).
p1 / p2	p1을 파싱하고, 효과가 없으면 p2를 파싱한다.
p star	0개 또는 그 이상의 p를 파싱한다.
p plus	1개 또는 그 이상의 p를 파싱한다.
p optional	가능하다면 p를 파싱한다.
p and	p를 파싱하되 그 입력은 소모하지 않는다.
p negate	p를 파싱하고, p가 실패하면 성공한다.
p not	p를 파싱하고, p가 실패하면 성공하지만 그 입력은 소모하지 않는다.
p end	p를 파싱하고, 입력 끝에서만 성공한다.
p times: n	p를 정확히 n 횟수만큼 파싱한다.
p min: n max: m	p를 최소 n 배 이상 m 배 이하로 파싱한다.
p starLazy: q	star와 같지만 q가 성공하면 소모를 중단한다.

파서 결합 (parser combination) 의 간단한 예제로 아래 identifier2 파서의 정의가 앞서 소개한 identifier의 정의와 동일함을 보이겠다.

스크립트 18.10: identifier 파서를 표현하는 다른 방법

```
identifier2 := \#letter asParser , (\#letter asParser / \#digit asParser) star.
```

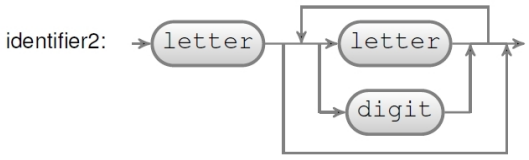


그림 18.2: 스크립트 18.10에 정의된 identifier2 파서에 대한 구문 해석도 표현.

파서 액션

파서에서 액션이나 변형을 정의하기 위해서는 프로토콜 Table 18.3에 정의된 PPParser≫==>, PPParser≫flatten, PPParser≫token, PPParser≫trim 중 하나를 사용할 수 있다.

표 18.3: PetitParser 는 많은 액션 파서를 사전 정의한다.

액션 파서	설명
p flatten	p의 결과로부터 문자열을 생성한다.
p token	flatten과 비슷하지만 세부 내용과 함께 PPToken을 리턴한다.
p trim	p 전후에 흰 공백을 제거(trim)한다.
p trim: trimParser	trimParesr가 파싱할 수 있는 것은 무엇이든 제거한다 (예 : 주석).
p ==> aBlock	주어진 aBlock 내에서 변형을 실행한다.

매칭한 요소의 배열을 얻는 대신 파싱된 식별자의 문자열을 리턴하려면 그 곳으로 PPParser≫flatten 메시지를 전송하여 파서를 구성하라.

스크립트 18.11: flatten을 이용해 파싱 결과가 문자열이 되도록 하기

```
|identifier|
identifier := (\#letter asParser , (\#letter asParser / \#digit asParser) star).
identifier parse: 'ajka0'     →  \#(\$a \#(\$j \$k \$a \$0))

identifier flatten parse: 'ajka0'     →   'ajka0'
```

PPParser≫token 메시지는 flatten과 유사하지만 token이 위치한 컬렉션, 컬렉션 내 그것의 위치 등 컨텍스트적 정보를 훨씬 더 많이 제공하는 PPToken을 리턴한다.

PPParser≫trim 메시지를 전송하면 파싱된 결과의 시작과 끝에 있는 흰 공백은 무시하도록 파서를 설정한다. 다음으로, 입력 상에서 첫 번째 파서를 사용할 경우 오류를 야기하는데, 파서가 공백(space)을 허용하지 않기 때문이다. 두 번째 파서를 이용해 공백이 무시되고 결과로부터 제거된다.

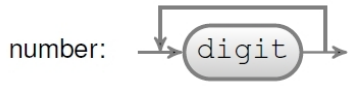


그림 18.3: 스크립트 18.14에 정의된 숫자(number) 파서의 구문 해석도 표현.

스크립트 18.12: PPParser≫trim를 이용해 공백 무시하기

```
|identifier|
identifier := (\#letter asParser , \#word asParser star) flatten.
identifier parse: ' ajka '  →   letter expected at 0

identifier trim parse: ' ajka '     →   'ajka'
```

trim 메시지를 전송하는 것은 #space asParser를 매개변수로 하여 PPParser≫trim:를 호출하는 것과 동일하다. 즉, trim:은 입력으로부터 다른 데이터, 즉 소스 코드 주석과 같은 데이터를 무시하는 데에 유용하다.

스크립트 18.13: PPParser≫trim를 이용해 주석 무시하기

```
| identifier comment ignorable line |
identifier := (\#letter asParser , \#word asParser star) flatten.
comment := '//' asParser, \#newline asParser negate star.
ignorable := comment / \#space asParser.
line := identifier trim: ignorable.
line parse: '// This is a comment
oneIdentifier // another comment'   →   'oneIdentifier'
```

PPParser≫==> 메시지는 파서가 입력에 매칭할 때 실행되어야 할 블록을 명시하도록 해준다. 다음 절에는 이와 관련된 예제가 몇 가지 제시되어 있을 것이다. 문자열 표현으로부터 숫자(number)를 얻는 간단한 방식을 아래 소개하겠다.

스크립트 18.14: 정수 파싱하기

```
number := \#digit asParser plus flatten ==> [:str | str asNumber].
number parse: '123'     →         123
```

표 18.3는 파서를 빌드 시 기본적인 요소들을 보여준다. PPParser의 operators 프로토콜에는 이보다 더 잘 문서화되고 테스트된 팩토리 메서드들이 몇 가지 더 있다. 이러한 팩토리 메서드에 대해 더 알고 싶다면 이러한 프로토콜을 살펴보라. 그 중에 흥미로운 메서드로 separatedBy:를 들 수 있는데, 이는 입력을 다른 파서에 의해 명시된 구분(separation)을 바탕으로 하여 한 번 또는 그 이상 파싱하는 새 파서를 응답한다.

조금 더 복잡한 문법 작성하기

이제 간단한 산술 표현식을 평가하기 위해 좀 더 복잡한 문법을 작성한다. 위에서 정의된 숫자에 대한 (사실 정수) 문법을 이용하면, 우선순위 순서대로 덧셈과 곱셈의 production을 정의하는

것이 다음 단계가 된다. production은 서로 재귀적으로 참조하기 때문에 PPDelegateParser로서 인스턴스화함을 주목하라. #setParaser: 메서드는 이러한 재귀를 해결한다. 아래 스크립트는 덧셈, 곱셈, 괄호에 대해 3개의 파서를 정의한다 (이와 연관된 구문 해석도는 그림 18.4를 참고).

스크립트 8.15: 산술 표현식 파싱하기

```
term := PPDelegateParser new.
prod := PPDelegateParser new.
prim := PPDelegateParser new.

term setParser: (prod , $+ asParser trim , term ==> [ :nodes | nodes first + nodes last ])
        / prod.
prod setParser: (prim , $*asParser trim , prod ==> [ :nodes | nodes first*nodes last ])
        / prim.
prim setParser: ($( asParser trim , term , $) asParser trim ==> [ :nodes | nodes second ])
        / number.
```

term 파서는 (1) prod 다음에 '+', 그 다음에 다른 term이 따라오도록 정의되거나 (2) prod로서 정의된다. (1)의 경우 액션 블록은 파서에게 첫 번째 노드(prod)와 마지막 노드(term)의 값의 산술 덧셈을 계산해줄 것을 요청한다. Prod 파서는 term 파서와 유사하다. Prim 파서는 term 전후에 왼쪽과 오른쪽 괄호를 허용하는데 이를 단순히 무시하는 액션 블록을 갖고 있다는 점에서 흥미롭다.

productions의 우선순위를 이해하기 위해서는 그림 18.5를 참고한다. 그림의 트리에서 루트는 (term) 먼저 시도된 production이다. Term은 + 이거나 prod에 해당한다. term production이 먼저 오는데, +는 산술에서 우선순위가 가장 낮기 때문이다.

파서가 입력을 모두 소모하도록 확신하기 위해선 end 파서를 이용해 start production으로 래핑(wrap)한다.

```
start := term end.
```

이게 모두다. 그러면 파서를 테스트할 수 있다.

스크립트 18.16: 산술 표현식 evaluator 시도하기

```
start parse: '1 + 2 \(\ast{}\) 3'. → 7
start parse: '(1 + 2) \(\ast{}\) 3'. → 9
```

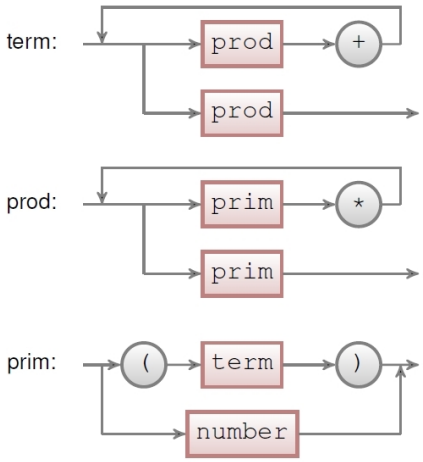


그림 18.4: 스크립트 18.15에 정의된 term, prod, prim 파서에 대한 구문 해석도 표현.

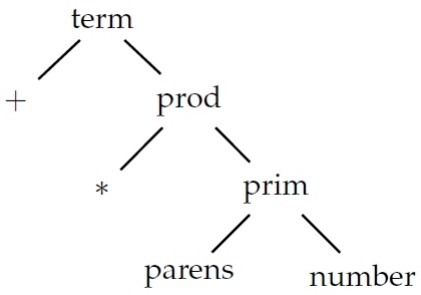


그림 18.5: productions의 우선순위를 이해하는 방법을 설명한다. 표현식은 term으로서, sum 또는 production에 해당한다. 가장 낮은 우선순위를 가진 sums를 먼저 인식할 필요가 있다. production은 곱셈(multiplication)이거나 프리미티브에 해당한다. 프리미티브는 괄호로 된 표현식이나 숫자로 되어 있다.

PetitParser를 이용한 Composite 문법

앞 절에서는 PetitParser의 기본 원리를 살펴보고 입문자를 위한 예제들을 몇 가지 제시하였다. 이번 절에서는 좀 더 복잡한 문법을 정의하는 방법을 소개할 것이다. 산술 표현식 문법을 계속해서 예로 들겠다.

앞에서와 같이 파서를 스크립트로서 작성하는 일은 번거로울 수 있으며, 특히 문법 production이 상호 재귀적이고 서로를 복잡한 방식으로 참조한다면 더 귀찮을 것이다. 게다가 단일 스크립트에 명시된 문법은 문법 중에서 특정 부분들을 재사용하기가 힘들 수밖에 없도록 만든다. 천만 다행으로 이를 위해 PPCompositeParser가 존재한다.

문법 정의하기

마지막 절에서 빌드한 것과 동일한 표현식 문법을 이용해 composite 파서를 생성하되, 이번에는 PPCompositeParser의 서브클래스에 정의해보자.

스크립트 18.17: 산술 표현식 문법을 보유하기 위한 클래스 생성하기

```
PPCompositeParser subclass: #ExpressionGrammar
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PetitTutorial'
```

다시 말하지만 정수 숫자에 대한 문법으로 시작한다. 아래와 같이 number 메서드를 정의하자.

스크립트 18.18: 첫 번째 파서를 메서드로서 구현하기

```
ExpressionGrammar>>number
    ^ #digit asParser plus flatten trim ==> [ :str | str asNumber ]
```

ExpressionGrammar 내의 모든 production은 그 파서를 리턴하는 메서드로서 명시된다. 우리도 이와 유사하게 term, prod, mul, prim productions를 정의한다. production들은 동일한 이름으로 된 각각의 인스턴스 변수를 읽음으로써 서로를 참조하고, PetitParser는 당신을 위해 이러한 인스턴스 변수를 자동으로 초기화해준다. 그리고 필요한 인스턴스 변수들을 처음으로 참조하면 Pharo가 자동으로 추가하도록 한다. 아래의 클래스 정의를 얻는다.

스크립트 18.19: 산술 표현식 문법을 보유하는 클래스 생성하기

```
PPCompositeParser subclass: #ExpressionGrammar
    instanceVariableNames: 'add prod term mul prim parens number'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PetitTutorial'
```

스크립트 18.20: 더 많은 표현식 문법 파서 정의하되, 연관된 액션 없이 정의하기

```
ExpressionGrammar>>term
    ^ add / prod

ExpressionGrammar>>add
    ^ prod , $+ asParser trim , term

ExpressionGrammar>>prod
    ^ mul / prim

ExpressionGrammar>>mul
    ^ prim , $* asParser trim , prod

ExpressionGrammar>>prim
    ^ parens / number

ExpressionGrammar>>parens
    ^ $( asParser trim , term , $) asParser trim
```

앞의 구현과 반대로 아직 production action를 정의하지 않고 (앞에서는 PParser≫==> 를 이용해 정의함), 덧셈(add), 곱셈(mul), 괄호(parens)에 해당하는 부분들을 구분된 production들로 뽑아낸다. 이는 추후 재사용 가능성을 향상시킨다. 예를 들어 서브클래스는 그러한 메서드를 오버라이드하여 약간 다른 production 출력을 생성할 수도 있다. 주로 production 메서드들은 grammar로 명명된 프로토콜에서 (필요 시 가령 grammar-literals와 같이 좀 더 구체적인 프로토콜 이름으로 재정의 가능한) 분류된다.

표현식 문법의 시작점을 마지막으로 정의할 것인데 그 중요성은 앞에서 소개한 것 못지 않다. 이는 ExpressionGrammar 클래스에서 PPCompositeParser≫start 를 오버라이드함으로써 이루어진다.

스크립트 18.21: 표현식 문법 파서의 시작점 정의하기

```
ExpressionGrammar>>start
    ^ term end
```

ExpressionGrammar를 인스턴스화하면 기본 추상적-구문 트리를 리턴하는 표현식을 제공한다.

스크립트 18.22: 간단한 산술 표현식에서 파서 테스트하기

```
parser := ExpressionGrammar new.
parser parse: '1 + 2 \(\ast{}\) 3'. → \#(1 \$+ \#(2 \$ \(\ast{}\) 3))
parser parse: '(1 + 2) \(\ast{}\) 3'. → \#(\#(\$( \#(1 \$+ 2) \$)) \$ \(\ast{}\) 3)
```

의존 문법(dependent grammars) 작성하기

다른 문법이 정의한 파서도 쉽게 재사용이 가능하다. 예로, 방금 정의한 ExpressionGrammar에 숫자의 정의를 재사용하는 새 문법을 생성한다고 가정해보자. 이를 위해서는 ExpressionGrammar로 의존성을 선언해야 할 것이다.

스크립트 18.23: ExpressionGrammar 문법으로부터 number 파서 재사용하기

```
PPCompositeParser subclass: #MyNewGrammar
    instanceVariableNames: 'number'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PetitTutorial'

MyNewGrammar class>>dependencies
    "Answer a collection of PPCompositeParser classes that this parser directly
        dependends on."
    ^ {ExpressionGrammar}

MyNewGrammar>>number
    "Answer the same parser as ExpressionGrammar>>number."
    ^ (self dependencyAt: ExpressionGrammar) number
```

evaluator 정의하기

이제 문법을 정의했으니 evaluator를 구현하기 위해 해당 정의를 재사용할 수 있다. 이를 위해 ExpressionEvaluator라고 불리는 ExpressionGrammar의 서브클래스를 생성한다.

스크립트 18.24: 서브클래스를 생성함으로써 문법을 계산기로부터 분리하기

```
ExpressionGrammar subclass: #ExpressionEvaluator
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PetitTutorial'
```

우리의 평가 구문을 이용해 add, mul, parens의 구현을 재정의한다. 이는 아래 메서드에서 보이는 바와 같이 super 구현을 호출하고 리턴된 파서를 조정함으로써 가능하다.

스크립트 18.25: 산술 표현식을 평가하기 위해 일부 파서의 정의를 재정의하기

```
ExpressionEvaluator>>add
    ^ super add ==> [ :nodes | nodes first + nodes last ]

ExpressionEvaluator>>mul
    ^ super mul ==> [ :nodes | nodes first * nodes last ]

ExpressionEvaluator>>parens
    ^ super parens ==> [ :nodes | nodes second ]
```

이제 evaluator를 테스트할 준비가 되었다.

스크립트 18.26: 간단한 산술 표현식에 자신의 evaluator 테스트하기

```
parser := ExpressionEvaluator new.
parser parse: '1 + 2 \(\ast{}\) 3'.   → 7
parser parse: '(1 + 2) \(\ast{}\) 3'.   → 9
```

Pretty-Printer 정의하기

우리는 이제 간단한 pretty printer를 정의하기 위해 문법을 재사용할 수도 있다. 이는 ExpressionGrammar를 다시 서브클래싱하는 것 만큼이나 쉽다!

스크립트 18.27: 서브클래스를 생성함으로써 문법을 pretty printer로부터 분리하기

```
ExpressionGrammar subclass: #ExpressionPrinter
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PetitTutorial'

ExpressionPrinter>>add
    ^ super add ==> [:nodes | nodes first , ' + ' , nodes third]

ExpressionPrinter>>mul
    ^ super mul ==> [:nodes | nodes first , ' * ' , nodes third]

ExpressionPrinter>>number
    ^ super number ==> [:num | num printString]

ExpressionPrinter>>parens
    ^ super parens ==> [:node | '(' , node second , ')']
```

이러한 pretty printer는 아래 표현식에서와 같이 시도할 수 있다.

스크립트 18.28: 간단한 산술 표현식에 자신의 pretty printer 테스트하기

```
parser := ExpressionPrinter new.
parser parse: '1+2 \(\ast{}\) 3'. → '1 + 2 \(\ast{}\) 3'
parser parse: '(1+ 2 ) \(\ast{}\) 3'. → '(1 + 2) \(\ast{}\) 3'
```

PPExpressionParser를 이용한 손쉬운 표현식

PetitParser는 표현식을 생성하는 강력한 툴을 제시하는데, 바로 PPExpressionParser라는 것으로, 이는 전위(prefix), 후위(postfix), 좌측 연관 및 우측 연관 연산자를 이용해 표현식 문법을 편리하게 정의하는 파서이다. 연산자 그룹은 내림차순으로 정의된다.

스크립트 18.29: 앞서 정의한 ExpressionGrammar는 몇 행만으로 구현이 가능하다

```
| expression parens number |
expression := PPExpressionParser new.
parens := $( asParser token trim , expression , $) asParser token trim
==> [ :nodes | nodes second ].
number := #digit asParser plus flatten trim ==> [ :str | str asNumber ].

expression term: parens / number.

expression
    group: [ :g |
        g left: $ * asParser token trim do: [ :a :op :b | a * b ].
        g left: $/ asParser token trim do: [ :a :op :b | a / b ] ];
    group: [ :g |
        g left: $+ asParser token trim do: [ :a :op :b | a + b ].
        g left: $- asParser token trim do: [ :a :op :b | a - b ] ].
```

스크립트 18.30: 이제 파서는 뺄셈과 나눗셈도 관리할 수 있다

```
expression parse: '1-2/3'. → (1/3)
```

PPCompositeParser의 서브클래스를 생성하거나 PPExpressionParser를 인스턴스화할 때는 어떻게 결정하는가? 작은 업무용으로 작은 파서를 실행하고 싶다면 PPExpressionParser를 인스턴스화해야 한다. 반대로 다수의 파서로 구성된 문법을 갖고 있다면 PPCompositeParser를 서브클래싱해야 한다.

문법 테스트하기

PetitParser 는 당신의 문법을 테스트하기 위한 프레임워크를 포함한다. 문법의 테스트는 아래와 같이 PPCompositeParserTest의 서브클래싱을 통해 이루어진다.

스크립트 18.31: 자신의 산술 표현식 문법의 테스트를 보유하기 위한 클래스 생성하기

```
PPCompositeParserTest subclass: #ExpressionGrammarTest
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PetitTutorial'
```

이제 test case 클래스는 파서 클래스를 참조한다는 사실이 중요한데, 이는 ExpressionGrammarTest에 있는 PPCompositeParserTest≫parserClass 메서드의 오버라이드를 통해 이루어진다.

스크립트 18.32: 자신의 test case 클래스를 자신의 파서로 연결하기

```
ExpressionGrammarTest>>parserClass
    ^ ExpressionGrammar
```

테스트 시나리오의 작성은 ExpressionGrammarTest에 새 메서드를 구현함으로써 이루어진다.

스크립트 18.33: 자신의 산술 표현식 구문에 대한 테스트 구현하기

```
ExpressionGrammarTest>>testNumber
    self parse: '123 ' rule: #number.

ExpressionGrammarTest>>testAdd
    self parse: '123+77' rule: #add.
```

이러한 테스트들은 ExpressionGrammar 파서가 명시된 production 규칙을 이용해 일부 표현식을 파싱할 수 있도록 보장한다. evaluator와 pretty printer의 테스트도 마찬가지로 쉽다.

스크립트 18.34: evaluator와 pretty printer 테스트하기

```
ExpressionGrammarTest subclass: #ExpressionEvaluatorTest
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PetitTutorial'

ExpressionEvaluatorTest>>parserClass
    ^ ExpressionEvaluator

ExpressionEvaluatorTest>>testAdd
    super testAdd.
    self assert: result equals: 200

ExpressionEvaluatorTest>>testNumber
    super testNumber.
    self assert: result equals: 123

ExpressionGrammarTest subclass: #ExpressionPrinterTest
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PetitTutorial'

ExpressionPrinterTest>>parserClass
    ^ ExpressionPrinter

ExpressionPrinterTest>>testAdd
    super testAdd.
    self assert: result equals: '123 + 77'

ExpressionPrinterTest>>testNumber
    super testNumber.
    self assert: result equals: '123'
```

사례 연구: JSON 파서

이번 절에서는 JSON 파서의 개발을 통해 PetitParser를 설명하고자 한다. JSON은 http://www.json.org 에 정의된 가벼운 데이터 교환 포맷이다. 우리만의 JSON 파서를 정의하기 위해 해당 웹사이트에 실린 명세를 이용하고자 한다.

JSON은 중첩된 쌍(nested pairs)과 배열을 기반으로 한 간단한 포맷이다. 아래 스크립트는 Wikipedia http://en.wikipedia.org/wiki/JSON 에서 발췌한 예를 제시한다.

스크립트 18.35: JSON의 예제

```
{ "firstName" : "John",
    "lastName" : "Smith",
    "age" : 25,
    "address" :
        {   "streetAddress" : "21 2nd Street",
            "city" : "New York",
            "state" : "NY",
            "postalCode" : "10021" },
    "phoneNumber":
        [
            {   "type" : "home",
                "number" : "212 555-1234" },
            {   "type" : "fax",
                "number" : "646 555-4567" } ] }
```

JSON 은 객체 정의(중괄호 "{}" 사이 내용)와 배열(사각 괄호 "[]" 사이 내용)로 구성된다. 객체 정의는 키/값 쌍의 집합인 반면 배열은 값의 리스트이다. 앞의 JSON 예제는 몇 개의 키/값 쌍으로 된 (예: 사람의 이름, 성, 나이) 객체(개인)를 표현한다. 개인의 주소는 다른 객체에 의해 표현되는 반면 휴대전화 번호는 객체의 배열에 의해 표현된다.

먼저 문법을 PPCompositeParser의 서브클래스로서 정의한다. 이를 PPJsonGrammar라고 부르자.

스크립트 18.36: JSON 문법 클래스 정의하기

```
PPCompositeParser subclass: #PPJsonGrammar
    instanceVariableNames: ''
```

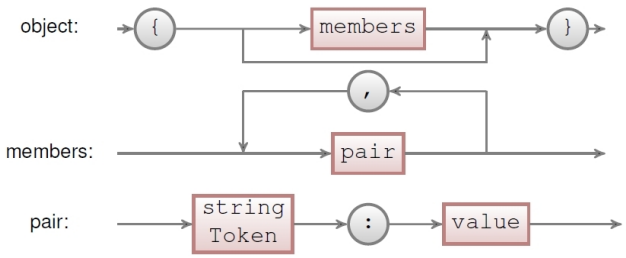


그림 18.6: 스크립트 18.37에 정의된 JSON 객체 파서에 대한 구문 해석도 표현.

```
classVariableNames: 'CharacterTable'
poolDictionaries: \textit{
category: 'PetitJson-Core'
```

후에 문자열의 파싱에 사용할 것이기 때문에 CharacterTable 클래스 변수를 정의하도록 한다.

객체와 배열 파싱하기

JSON 객체와 배열에 대한 구문 해석도를 그림 18.6과 그림 18.7에 소개하고 있다. 아래 코드를 이용해 JSON 객체에 대한 PetitParser를 정의할 수 있겠다.

스크립트 18.37: 그림 18.6에 표현된 바와 같이 객체에 대한 JSON 파서 정의하기

```
PPJsonGrammar>>object
    ^ ${ asParser token trim , members optional , $} asParser token trim

PPJsonGrammar>>members
    ^ pair separatedBy: $, asParser token trim

PPJsonGrammar>>pair
    ^ stringToken , $: asParser token trim , value
```

여기서 새로운 내용은 PPParser≫separatedBy: 라는 편의 메서드를 호출하는 것인데, 해당 메서드는 수신자(본문에서는 값)를 한 번 또는 그 이상 파싱하는 새 파서를 응답하며 각 파싱은 그 매개변수 파서(본문에서는 콤마)에 의해 구분된다.

배열은 스크립트 18.38에 묘사된 바와 같이 파싱이 훨씬 수월하다.

스크립트 18.38: 그림 18.7에 표현된 바와 같이 배열에 대한 JSON 파서 정의하기

```
PPJsonGrammar>>array
    ^ $[ asParser token trim ,
       elements optional ,
    $] asParser token trim

PPJsonGrammar>>elements
    ^ value separatedBy: $, asParser token trim
```

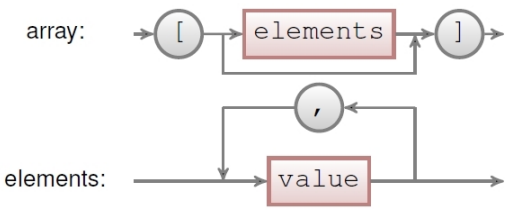


그림 18.7: 스크립트 18.38에 정의된 JSON 배열 파서에 대한 구문 해석도 표현.

값 파싱하기

JSON에서 값은 문자열, 숫자, 객체, 배열, 부울(true 또는 false), 널(null) 중에 하나에 해당한다. value 파서는 아래와 같이 정의되며 그림 18.8에 제시하였다.

스크립트 18.39: 그림 18.8에 제시된 바와 같이 값이 대한 JSON 파서 정의하기

```
PPJsonGrammar>>value
    ^ stringToken / numberToken / object / array /
        trueToken / falseToken / nullToken
```

문자열은 파싱 시 어느 정도 작업이 필요하다. 문자열은 쌍따옴표로 시작하고 끝난다. 쌍따옴표 안의 내용은 문자의 시퀀스다. 그 중 어떤 문자든 escape 문자, 8진 문자, 또는 일반 문자가 가능하다. escape 문자는 백슬래시 바로 다음에 특수 문자가 따라온다(예: '\n'은 문자열에 새 행을 얻는다). 8진 문자는 백슬래시 바로 뒤에 'u' 글자가 따라오고, 그 다음에 4개의 16진 숫자가 따라온다. 마지막으로 일반 문자는 쌍따옴표(문자열을 끝내는 데에 사용되는)와 백슬래시(escape 문자를 소개 시 사용되는)를 제외하고 어떤 문자든 가능하다.

스크립트 18.40: 그림 18.9에 제시된 바와 같이 문자열에 대한 JSON 파서 정의하기

```
PPPJsonGrammar>>stringToken
    ^ string token trim
PPJsonGrammar>>string
    ^ $” asParser , char star , $” asParser
PPJsonGrammar>>char
    ^ charEscape / charOctal / charNormal
PPJsonGrammar>>charEscape
    ^ $\ asParser , (PPPredicateObjectParser anyOf: (String withAll: CharacterTable keys))
PPJsonGrammar>>charOctal
    ^ ’\u’ asParser , (#hex asParser min: 4 max: 4)
PPJsonGrammar>>charNormal
    ^ PPPredicateObjectParser anyExceptAnyOf: ’\”’
```

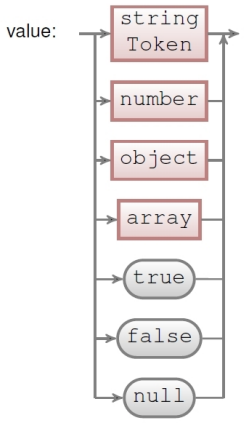


그림 18.8: 스크립트 18.39에 정의된 JSON value 파서에 대한 구문 해석도 표현.

슬래시 다음에 특수 문자가 허용되는데 특수 문자의 의미는 initialize 클래스 메서드에서 초기화하는 CharacterTable dictionary에 정의된다. 클래스를 시스템에 로딩하면 클래스 측에서 initialize 메서드가 호출됨을 주목하길 바란다. initialize 메서드를 방금 생성했다면 클래스는 메서드 없이 로딩된다. 이를 실행하려면 워크스페이스에 PPJsonGrammar initialize를 평가해야 한다.

스크립트 18.41: JSON 특수 문자와 그 의미 정의하기

```
PPJsonGrammar class>>initialize
    CharacterTable := Dictionary new.
    CharacterTable
        at: $\ put: $\;
        at: $/ put: $/;
        at: $” put: $”;
        at: $b put: Character backspace;
        at: $f put: Character newPage;
        at: $n put: Character lf;
        at: $r put: Character cr;
        at: $t put: Character tab
```

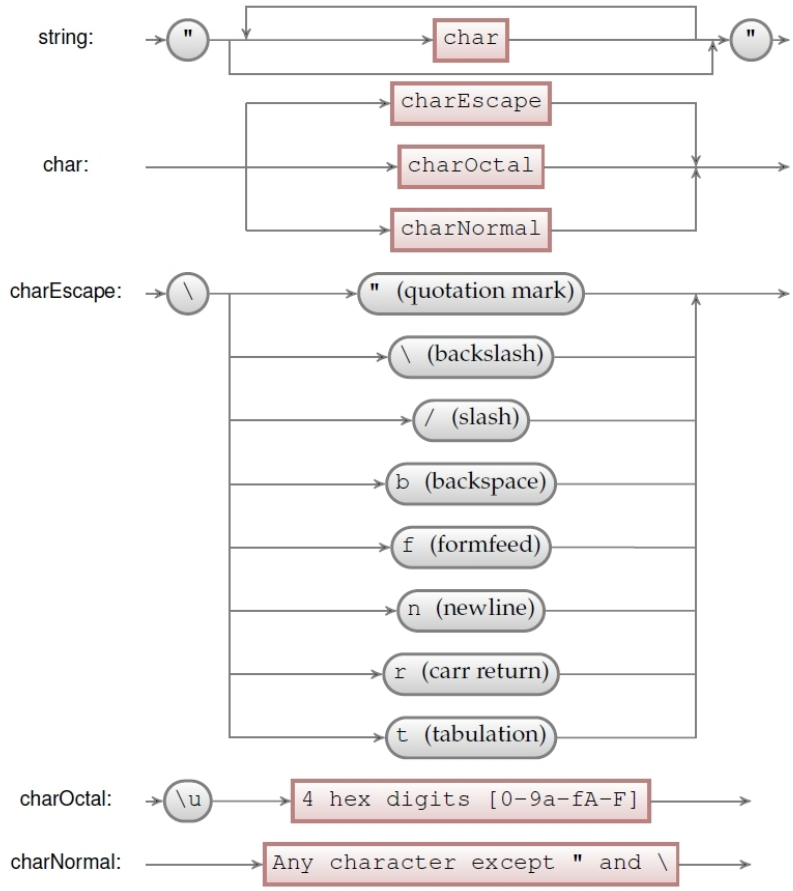


그림 18.9: 스크립트 18.40에 정의된 JSON 문자열 파서에 대한 구문 해석도 표현.

숫자를 파싱하는 것은 약간 더 간단한데, 숫자는 음수이거나 양수에 해당하는 동시 정수이거나 십진수(decimal)기 때문이다. 뿐만 아니라 십진수는 부동 소수점 숫자(floating number)

구문을 이용해 표현 가능하다.

스크립트 18.42: 그림 18.10에 제시된 바와 같이 숫자에 대한 JSON 파서 정의하기

```
PPJsonGrammar>>numberToken
    ^ number token trim
PPJsonGrammar>>number
    ^ $- asParser optional ,
    ($0 asParser / #digit asParser plus) ,
    ($. asParser , #digit asParser plus) optional ,
    (($e asParser / $E asParser) , ($- asParser / $+ asParser) optional , #digit asParser
        plus) optional
```

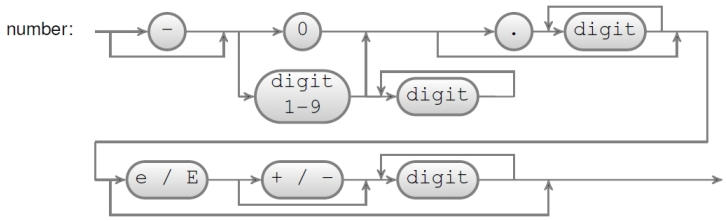


그림 18.10: 스크립트 18.42에 정의된 JSON 숫자 파서에 대한 구문 해석도 표현.

주의를 기울여 읽은 독자라면 그림 18.10의 구문 해석도와 스크립트 18.42 코드에 약간의 차이점을 눈치챘을 것이다. JSON에서 숫자는 앞에 0이 올 수가 없기 때문에, 가령 "01"과 같은 문자열은 유효하지 않은 숫자를 표현한다. 구문 해석도는 0 또는 1부터 9까지 숫자를 허용함으로써 이 점을 명시적으로 표현한다. 위의 코드에서는 파서 combinator $/가 정렬된다는 사실에 의존함으로써 규칙을 암묵적으로 만들어 냈으며, $/의 우측에 있는 파서는 왼쪽의 파서가 실패할 경우에만 시도되므로, ($0 asParser / #digit asParser plus)는 0으로 시작되지 않는 숫자의 시퀀스 또는 0으로 숫자를 정의한다.

다른 파서들은 평범한 편이다.

스크립트 18.43: 누락된 JSON 파서 정의하기

```
PPJsonGrammar>>falseToken
    ^ 'false' asParser token trim
PPJsonGrammar>>nullToken
    ^ 'null' asParser token trim
PPJsonGrammar>>trueToken
    ^ 'true' asParser token trim
```

유일하게 빠진 부분은 start 파서다.

스크립트 18.44: JSON start 파서를 값(그림 18.8) 외에 어떤 것도 따라오지 않는 것으로 정의하기

```
PPJsonGrammar>>start
    ^ value end
```

PetitParser Browser

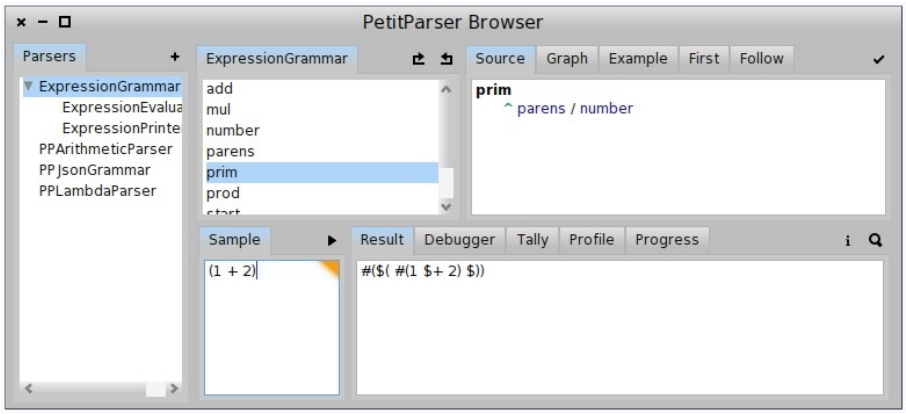


그림 18.11: PetitParser Browser 창.

PetitParser 에는 복잡한 파서를 개발하도록 도와주는 강력한 브라우저가 구비되어 있다. PetitParser Browser는 그래픽 시각성, 디버깅 지원, 재팩토링 지원을 비롯해 본 장에서 소개할 다른 여러 기능들을 제공한다. 이러한 기능들은 자신의 파서를 개발하는 데에 있어 매우 유용함을 확인할 것이다. 시스템에 Glamour를 로딩한 채 진행하도록 한다. Glamour를 로딩하려면 10을 참고한다. 이후 아래 표현식을 평가하여 PetitParser를 열어라.

스크립트 18.45: PetitParser 브라우저 열기

```
PPBrowser open.
```

PetitParser Browser 개요

그림 18.11에서 PPBrowser 창이 보일 것이다. 좌측 패널, Parsers는 시스템 내 모든 파서의 리스트를 포함한다. ExpressionGrammar와 그 서브클래스를 비롯해 본 장의 앞에서 정의한 PPJsonGrammar도 보일 것이다. 이 패인에서 파서를 하나 선택하면 브라우저 우측 상단이 활성화된다. 선택된 파서(예: prim)의 각 규칙과 관련해, 규칙에 연관된 5개의 탭이 보일 것이다.

Source 는 규칙에 대한 소스 코드를 표시한다. 코드는 해당 창에서 업데이트 및 저장이 가능하다. 뿐만 아니라 새 메서드의 이름과 body를 정의함으로써 새 규칙을 추가할 수 있다.

Graph 는 규칙의 그래픽 표현을 표시한다. 규칙 소스가 변경되면서 이 또한 업데이트된다. prim의 시각적 표현은 그림 18.12에서 확인할 수 있다.

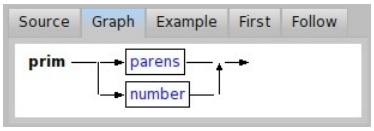


그림 18.12: prim 규칙의 그래픽 시각화.

Example 은 규칙의 정의를 기반으로 자동으로 생성된 예제를 보여준다 (그림 18.3의 prim 규칙에 대한 예제를 참고). 우측 상단 모서리의 reload 버튼은 동일한 규칙에 대한 새로운 예제를 생성한다 (그림 18.14는 prim 규칙에 대해 자동으로 생성되는 또 다른 예제를 보여주는데, 이번에는 괄호로 된 표현식이 이용되었다).

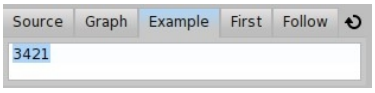


그림 18.13: 자동으로 생성된 prim 규칙의 예제. 이번 경우 prim 예제는 숫자이다.

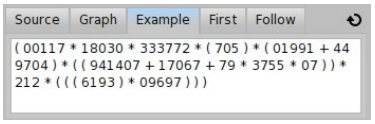


그림 18.14: reload 버튼을 클릭한 후 자동으로 생성된 prim 규칙의 또 다른 예제다. 이번 경우 prim 예제는 괄호로 된 표현식에 해당한다.

First 는 규칙이 시작된 직후 활성화가 가능한 terminal 파서 집합을 보여준다. 그림 18.15에서 볼 수 있듯이 prim의 첫 번째 집합은 숫자(digit) 또는 여는 괄호 '(' 가 된다. 즉, prim의 파싱을 시작하는 즉시 입력은 숫자 또는 '(' 를 계속해야 함을 의미한다.

첫 번째 집합을 이용해 문법이 올바로 명시되었는지 다시 확인할 수 있다. 예를 들어, prim의 첫 번째 집합에 '+' 가 보인다면 정의의 어딘가에 문제가 있다는 뜻인데, prim 규칙은 절대 이항 연산 기호로 시작되도록 만들어진 것이 아니기 때문이다.

terminal 파서는 다른 파서로 위임하지 않는 파서이다. 따라서 prim 첫 번째 집합에 parens가 보이지 않을 것인데, parens는 다른 파서-trimming 파서와 시퀀스 파서-로 위임하기 때문이다 (스크립트 18.46 참고). '(' 가 parens의 첫 번째 집합임을 확인할 수 있을 것이다. number 규칙도 마찬가진데, 이는 action 파서를 생성하고 해당 파서는 trimming 파서로 위임하며, 이 파서는 flattening 파서로 위임하고, 이는 다시 repeating 파서로 위임하며 이는 마지막으로 #digit 파서로 위임한다 (스크립트 18.46 참고). #digit 파서는 terminal 파서이므로 첫 번째 집합에서 '예상되는 숫자' 를 확인할 수 있다. 일반적으로 첫 번째 집합의 계산은 복잡할 수 있으므로 PPBrowser는 우리를 위해 이러한 정보를 계산해준다.

그림 18.15: prim 규칙의 첫 번째 집합.

스크립트 18.46: ExpressionGrammar에서 prim 규칙

```
ExpressionGrammar>>prim
    ^ parens / number

ExpressionGrammar>>parens
    ^ $( asParser trim, term, $} asParser trim

ExpressionGrammar>>number
    ^ #digit asParser plus flatten trim ==> [:str | str asNumber ]
```

Follow 는 규칙이 완료된 직후 활성화가 가능한 terminal 파서의 집합을 보여준다. 그림 18.16에서 볼 수 있듯이 잇따른 prim의 집합은 닫는 괄호 문자 파서 ')', 별표 문자 파서 '∗', 플러스 문자 파서 '+' 또는 엡실론 파서(빈 문자열을 나타냄)에 해당한다. 다시 말해, prim 규칙의 파싱을 완료하고 나면, 입력은 ')', '∗', '+' 문자 중 하나를 계속해야 하고, 그렇지 않으면 입력이 완전히 소모되어야 한다.

문법이 올바로 명시되었는지 확인하기 위해 추종 기호 집합(follow set)을 사용할 수도 있다. 예를 들어, prim 추종 기호 집합에서 '(' 가 보이면 자신의 문법 정의에 무언가 잘못된 것이다. prim 규칙 다음에는 이항 연산 기호나 닫는 괄호가 따라와야 하며 여는 기호가 뒤따라선 안 된다.

일반적으로 follow의 계산은 첫 번째 계산보다 더 복잡할 수 있으므로, PPBrowser 가 우리를 대신해 해당 정보를 계산해준다.

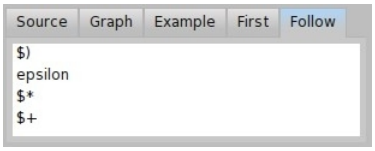


그림 18.16: prim 규칙의 추종 기호 집합.

브라우저의 우측 하단은 특정 파싱 입력과 연관된다. Sample 탭의 텍스트 영역을 채움으로써 입력 예제를 명시할 수 있다. play ▶ 버튼을 클릭하거나 Cmd-s 또는 Ctrl-s 버튼을 눌러 입력 예제를 파싱할 수도 있다. 이후 우측 하단 패인에 있는 탭을 검사함으로써 파싱 결과에 대한 정보를 얻을 수 있다.

Result 는 Inspect 과 Explore 버튼 중 하나를 클릭하면 검사할 수 있는 입력 예제의 파싱 결과를 보여준다. 그림 18.17은 (1+2)의 파싱 결과를 표시한다.

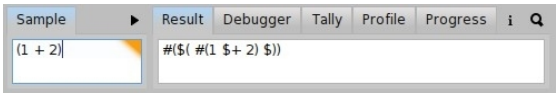


그림 18.17: (1+2) 예제 표현식의 파싱 결과.

Debugger 는 파싱 도중에 실행된 단계들의 트리 뷰를 표시한다. 이는 파싱 도중에 정확히 무슨 일이 일어나는지 알지 못할 때 매우 유용하다. 단계를 선택하면 입력의 하위집합이 강조 표시되어 특정 단계에서 파싱된 입력 부분을 확인할 수 있다.

가령, ExpressionGrammar가 작용하는 방식과 어떤 규칙이 어떤 순서로 호출되는지를 확인할 수 있는 것이다. 이것을 그림 18.18에서 설명한다. grey 규칙은 실패한 규칙이다. 이는 주로 choice 파서에 발생하는데, prod 규칙에 대한 예제를 확인할 수 있다 (그 정의가 스크립트 18.47에 제시되어 있다). 파서가 12+3∗4 term을 파싱했을 당시에는 파서가 mul 규칙을 prod에서 첫 번째 옵션으로서 파싱을 시도하였다. 하지만 mul은 별표 문자 '∗'를 위치 2에 필요로 했는데 해당 위치는 존재하지 않았기 때문에 mul가 실패하였고, 대신 12 값을 가진 prim이 파싱되었다.

스크립트 18.47: ExpressionGrammar 내의 prod 규칙

```
ExpressionGrammar>>prod
    ^ mul / prim
ExpressionGrammar>>mul
    ^ prim, $ * asParser trim, prod
```

12+3∗4를 12∗3∗4로 변경할 때 파싱 도중에 어떤 일이 발생하는지 비교해보라. 어떤 규칙이 적용되고, 그 중 어떤 규칙이 실패하는가? 두 번째 디버거 출력을 그림 18.19에서 소개하고 있지만 스스로 시도해보길 바란다.

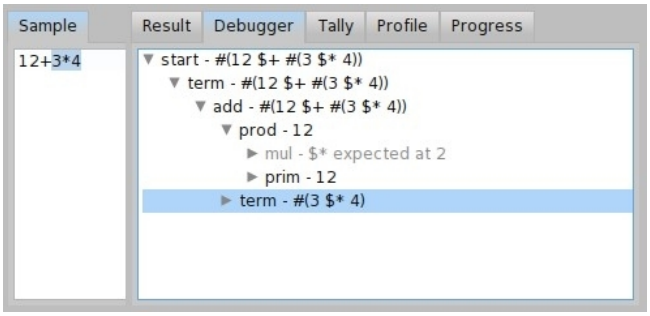


그림 18.18: 입력 12 + 3 * 4에 대한 ExpressionGrammar의 디버거 출력.

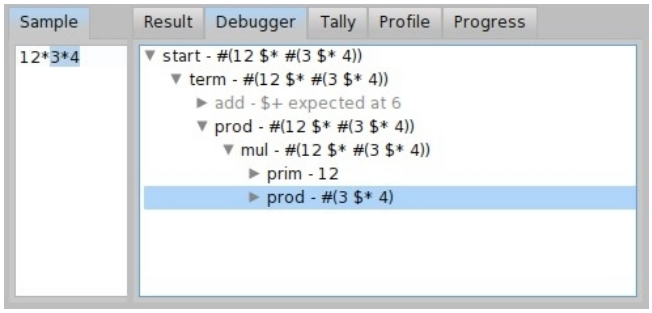


그림 18.19: 입력 12 * 3 * 4에 대한 ExpressionGrammar의 디버거 출력.

Tally 는 특정 파서가 파싱 도중에 몇 번이나 호출되었는지를 보여준다. 퍼센트는 총 호출 횟수에 대한 비율이다. 이는 자신의 파서 성능을 최적화할 때 유용할 것이다 (그림 18.20 참고).

Profile 은 입력의 파싱 도중에 특정 파서에서 소요된 시간을 표시한다. 퍼센트는 총 시간에 대한 비율이다. 이는 자신의 파서 성능을 최적화할 때 유용할 것이다 (그림 18.20 참고).

Progress 는 파서가 입력을 어떻게 소모하는지 시각적으로 표시한다. x 축은 입력 예제에서 얼마나 많은 문자가 읽혔는지 0부터 (좌측 margin) 입력의 문자 개수 (우측 margin) 까지 범위로 나타낸다. y 축은 시간을 나타내는데, 파싱 프로세스의 상단 한계 (top margin) 부터 하단 한계 (bottom margin) 까지 범위다. 좌측 상단부터 우측 하단까지 그어진 직선은 (그림 18.22 참고) 파서가 입력 예제의 각 문자를 한 번만 읽어서 작업을 완료하였음을 나타낸다. 이는 최고의 시나리오로, 파싱은 입력 길이를 따라 선형으로 이루어지는 경운데, 다시 말해 n 개 문자의 입력이 n 개 단계에서 파싱됨을 의미한다.

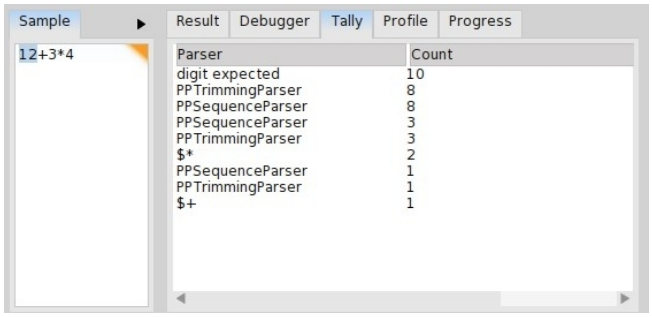


그림 18.20: 입력 12 * 3 * 4에 대한 ExpressionGrammar의 tally.

직선이 여러 개가 보이면 파서가 입력 예제에서 이전에 읽힌 문자로 돌아가 다른 규칙을 시도함을 의미한다. 그림 18.23에서 이를 확인할 수 있다. 이 예제에서 파서는 전체 입력 예제를 올바로 파싱하기 위해 여러 번 돌아가야 했는데, 모든 입력이 n에서 파싱되었기 때문이다! 문법에 대한 backward jump가 여러 개 보인다면 choice 파서의 순서를 재고려하거나, 자신

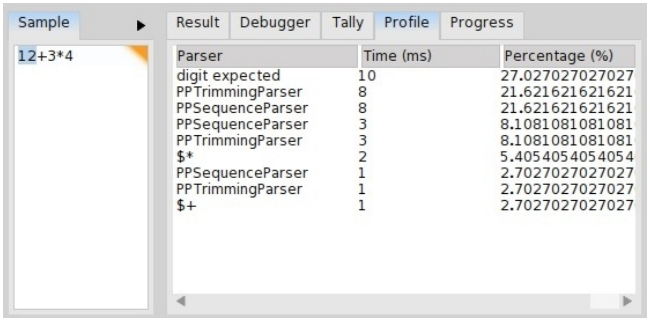


그림 18.21: 입력 12 * 3 * 4에 대한 ExpressionGrammar의 프로파일.

의 문법을 재구성하거나, 보관된(memoized) 파서를 사용해야 한다. 추적 문제는 다음 절에서 상세히 살펴볼 것이다.

그림 18.22: 선형적 단계의 양으로 입력을 파싱하는 Petit Parser의 경과.

그림 18.23: 추적이 많은 Petit Parser의 경과.

디버깅 예제

연습으로 스크립트 18.48의 BacktrackingParser를 개선해보겠다. BacktrackingParser는 정규 표현식 'a∗b'와 'a∗c'에 상응하는 입력을 허용하기 위해 설계되었다. 파서는 올바른 결과를 제공하지만 성능에 문제가 있다. BacktrackingParser가 추적을 너무 많이 실행하기 때문이다.

스크립트 18.48: 너무 많은 추적으로 'a ∗ b'와 'a ∗ c'를 수락하는 파서.

```
PPCompositeParser subclass: #BacktrackingParser
    instanceVariableNames: 'ab ap c p'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PetitTutorial'

BacktrackingParser>>ab
    ^ 'b' asParser /
    ('a' asParser, ab)

BacktrackingParser>>c
    ^ 'c' asParser

BacktrackingParser>>p
    ^ ab / ap / c

BacktrackingParser>>start
    ^ p

BacktrackingParser>>ap
    ^ 'a' asParser, p
```

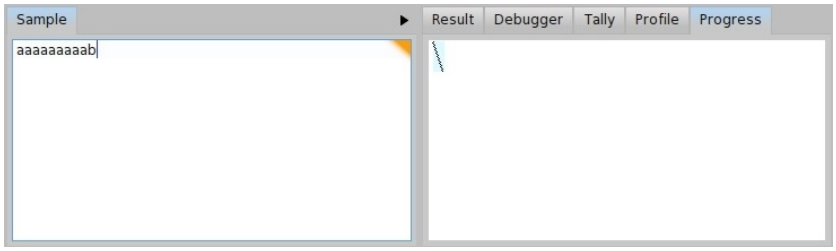


그림 18.24: inputb 에 대한 BacktrackingParser의 경과.

무엇이 일어나는지 이해를 돕기 위해 개요를 설명하겠다. 제일 먼저 inputb = 'aaaaaaaaab' 과 inputc = 'aaaaaaaaac'. 를 파싱해보라. 그림 18.24에 나타난 경과에서 볼 수 있듯이 inputb 는 다소 선형적인 시간으로 파싱되고, 추적이 전혀 없다. 하지만 그림 18.25에 설명된 경과는 상태가 좋아 보이지 않는다. 많은 추적과 함께 파싱되고 시간이 훨씬 더 많이 소요된다. inputb 와 inputc 모두 tally 출력을 비교할 수도 있다 (그림 18.26과 그림 18.27 참고). inputb의 경우 파서 b의 총 호출 횟수는 19회, 파서 a의 호출 횟수는 9회다. 이는 inpubc 의 경우 파서 b의 110 회 호출, 파서 a의 55회 호출에 비하면 엄청 적다.

inpubc에 문제가 있음을 확인할 수 있다. 문제가 무엇인지 아직도 모르겠다면 디버거 창이 힌트를 제공해줄 것이다. 그림 18.28과 같이 inputb에 대한 디버거 창을 살펴보자. 각 단계에서 하나의 'a'가 소모되고 파서 ab는 'b'로 도달할 때까지 호출됨을 확인할 수 있다. inpubc 에 대한 디버거 창은 그림 18.29과 같이 약간 다르다. p->ab->ap->p 루프 내에서 경과가 있지만 파서 ab는 루프의 반복마다 실패한다. 파서 ab는 모든 문자열을 끝까지 읽고 'b' 대신 'c' 가 보인 이후 실패하므로, 추적의 원인을 로컬화하였다. 이제 문제를 알았으니 무엇을 할 수 있을까? BacktrackingParser를 업데이트하여 'a∗c' 문자열이 'a∗b' 문자열과 비슷한 방식으로 파싱되도록 시도할 수 있다. 그러한 수정은 스크립트 18.49에서 확인할 수 있다.

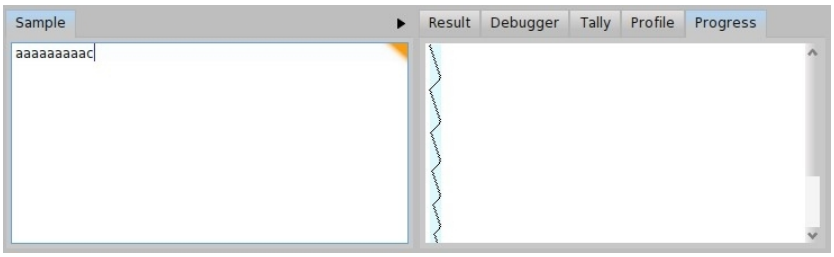


그림 18.25: inpubc에 대한 BacktrackingParser의 경과.

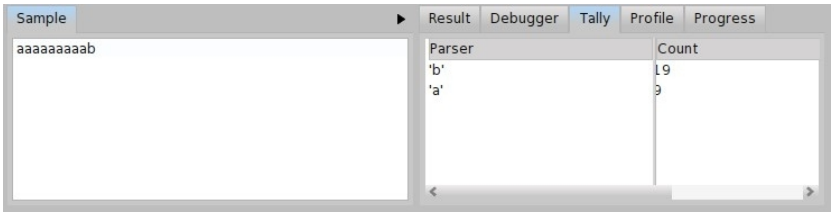


그림 18.26: inputb에 대한 BacktrackingParser의 tally 출력.

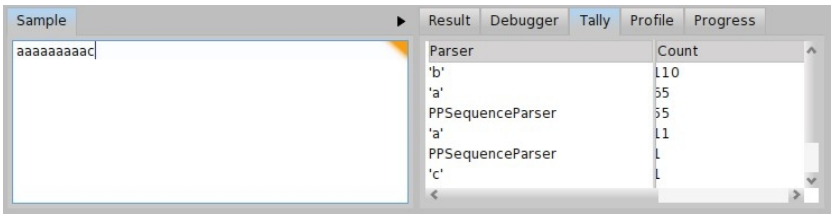


그림 18.27: inputc에 대한 BacktrackingParser의 tally 출력.

스크립트 18.49: 'a * b'와 'a * c'를 허용하는 더 나은 파서.

```
PPCompositeParser subclass: #BacktrackingParser
    instanceVariableNames: 'ab ac'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PetitTutorial'

BacktrackingParser>>ab
    ^ 'b' asParser /
    ('a' asParser, ab)

BacktrackingParser>>ac
    ^ 'b' asParser /
    ('c' asParser, ab)

BacktrackingParser>>start
    ^ ab / ac
```

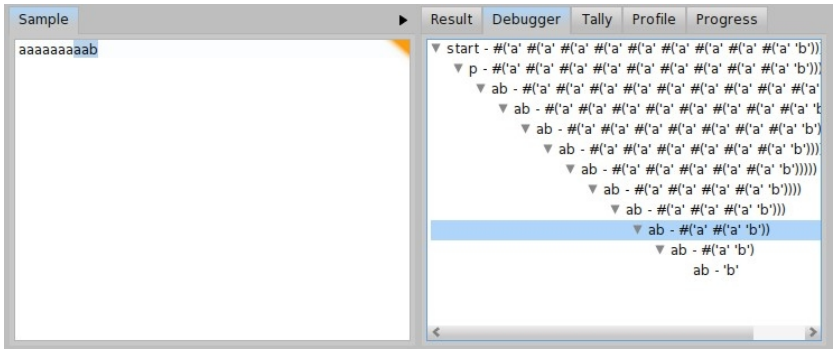


그림 18.28: inputb에 대한 BacktrackingParser의 출력 디버깅.

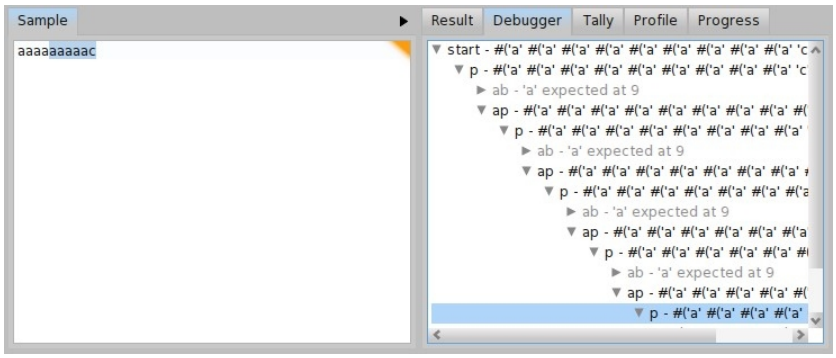


그림 18.29: inputc에 대한 BacktrackingParser의 출력 디버깅.

그림 18.30과 그림 18.31에서 inputc에 대한 새 측정(metrics)을 확인할 수 있다. 크게 개선된 점이 보인다. inputc의 경우 tally를 통해 파서 a가 20번만 호출되고 파서 a가 9번 호출됨을 확인할 수 있다. 이는 파서 b가 110회 호출되고 파서 a가 55회 호출되던 기존 BacktrackingParser(그림 18.27 참고)에 비하면 엄청나게 향상된 것이다.

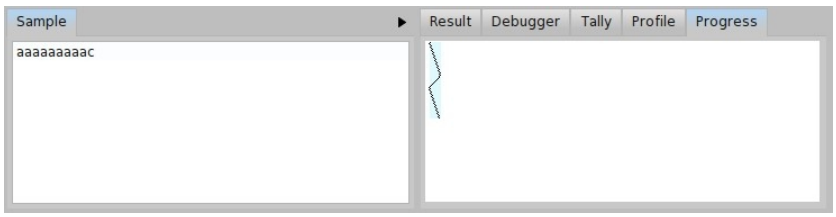


그림 18.30: 첫 번째 업데이트 후 inputc에 대한 BacktrackingParser의 경과.

여기서 끝날 것이 아니라 이보다 더 개선할 수도 있다. inputc에 여전히 한 번의 추적이 발생한다. 이는 파서 ab가 'a∗b' 입력의 인식을 시도하고 실패(그리고 추적)하여 파서 ac가 'a∗c' 입력을 인식할 수 있을 때 발생한다. 그렇다면 모든 'a'들을 소모한 후 젤 마지막에 'b'와 'c'

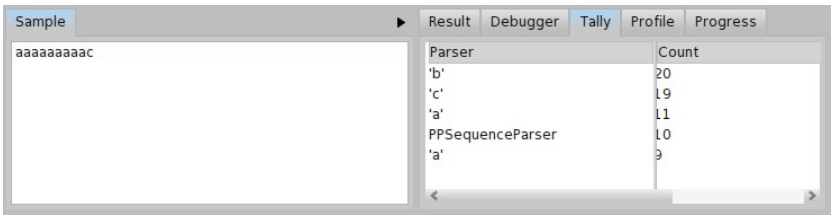


그림 18.31: 첫 번째 업데이트 후 inputc에 대한 BackTrackingParser의 Tally.

중에서 선택하는 경우는 어떨까? BacktrackingParser의 그러한 수정은 스크립트 18.50에서 확인할 수 있다. 이런 경우 그림 18.32에 묘사된 것처럼 inputc에 대한 추적이 없이도 경과를 확인할 수 있다.

반대로 inputb에 대한 파서 호출 횟수는 18만큼 증가했다 (Table 18.4에 Backtracking-Parser의 각 버전마다 총 호출 횟수가 실려 있다). 자신의 요구에 맞는 문법을 결정하는 일은 프로그래머에게 달려 있다. 'a∗b'와 'a∗c'가 입력에 동일한 가능성으로 발생할 경우 두 번째로 개선된 버전을 사용하는 편이 낫다. 입력에서 더 많은 'a∗b' 문자열이 예상된다면 첫 번째 버전이 낫다.

스크립트 18.50: 'a∗b'와 'a∗c'를 허용하는 더 나은 파서.

```
PPCompositeParser subclass: #BacktrackingParser
    instanceVariableNames: 'abc'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PetitTutorial'

BacktrackingParser>>abc
    ^ ('b' asParser / 'c' asParser) /
    ('a' asParser, abc)

BacktrackingParser>>start
    ^ abc
```

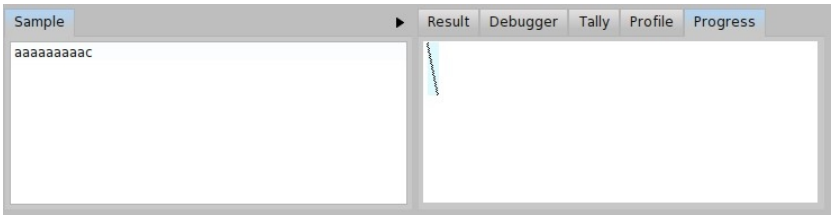


그림 18.32: inputc에 대한 두 번째 업데이트 이후 BacktrackingParser의 경과.

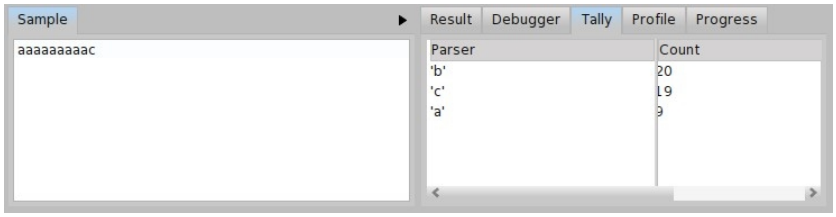


그림 18.33: inputc에 대한 두 번째 업데이트 이후 BacktrackingParser의 Tally.

표 18.4: BacktrackingParser의 버전에 따른 inputb와 inputc에 대한 파서 호출 횟수.

버전	inputb 호출 횟수	inputc 호출 횟수
원본	28	233
첫 번째 개선	28	70
두 번째 개선	46	48

Packrat 파서

본 장을 시작할 무렵 네 가지 파서 방법을 언급한 바 있는데, 그 중에 Packrat Parsers가 있었다. packrat 파싱은 선형적 파스 시간을 제공한다고 주장하는 바이다. 하지만 디버깅 예제에서 우리는 BacktrackingParser의 원본 버전이 233개 단계에서 10 길이로 inputc를 파싱하였음을 보았다. 그리고 longinputc = 'aaaaaaaaaaaaaaaaaaaac' (길이 20)의 파싱을 시도할 경우 원본 파서는 969개 단계가 필요함을 확인할 것이다. 사실 경과는 선형적이지 않다.

PetitParser 프레임워크는 기본적으로 packrat 파싱을 사용하지 않는다. packrat 파싱을 활성화하기 위해서는 보관된 메시지를 전송할 필요가 있다. 보관된 파서는 입력과 특정 파서에서 특정 위치에 대한 파싱이 한 번만 실행되고 추후 사용을 위해 결과가 dictionary에 기억되도록 보장한다. 파서가 입력을 두 번째로 파싱하길 원하면 dictionary에서 결과가 검색될 것이다. 이런 방식을 통해 불필요한 파싱을 대다수 피할 수 있다. 단점은 PetitParser가 가능한 모든 위치에서 가능한 모든 파서의 결과를 모두 기억해야 하기 때문에 훨씬 많은 메모리를 필요로 한다는 데에 있다.

packrat 파서의 예제를 제공하기 위해 BacktrackingParser를 다시 살펴보자 (스크립트 18.48). 앞서 분석했듯이 문제는 파서 ab에 있었는데, 이것이 p -> ab -> ap -> p 루프에서 지속적으로 실패했다는 점이다. 이제 트릭을 써서 스크립트 18.51에서와 같이 메서드 ab를 업데이트하여 파서 ab를 보관할 수 있다. 보관(memoization)이 적용되면 그림 18.34에서와 같이 inputc에 대해 총 63회 호출과 longinputc에 대해 129회 호출로 된 경과를 얻을 것이다. BacktrackingParser를 약간만 수정하여 대략적으로 6을 지수(factor)로 하는 선형적 파싱 시간을 (입력의 길이와 연관) 얻었다.

스크립트 18.51: 파서 ab의 보관된 버전.

```
BacktrackingParser>>ab
    ^ ( 'b' asParser /
        ('a' asParser, ab)
    ) memoized‘
```

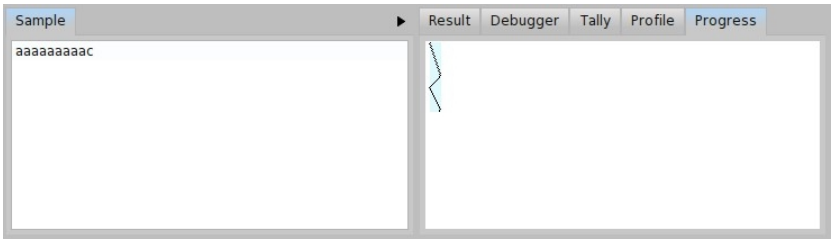


그림 18.34: BacktrackingParser의 보관된 버전의 경과.

요약

PetitParser에 대한 지침을 마무리하겠다. 지금까지 아래를 살펴보았다.

- 파서는 combinator와 결합된 다수의 작은 파서들을 구성 요소로 한다.
- 문자열을 파싱하기 위해서는 parse: 메서드를 사용한다.
- 문자열이 문법에 매치하는지 알기 위해서는 matches: 메서드를 사용한다.
- flatten 메서드는 파싱 결과로부터 String을 리턴한다.
- ==> 메서드는 매개변수에 주어진 블록에 주어진 변형을 실행한다.
- PPCompositeParser를 서브클래싱함으로써 파서를 구성(하고 문법을 생성)한다.
- PPCompositeParserTest를 서브클래싱하여 자신의 파서를 테스트한다.

PetitParser와 그 개념, 구현을 광범위하게 살펴보려면 Moose book[4]과 Lukas Renggli의 PhD[5]에 그와 관련된 장을 참고하길 바란다.

[4] http://www.themoosebook.org/book/internals/petit-parser
[5] http://scg.unibe.ch/archive/phd/renggli-phd.pdf

제 19 장

저자 이력

Alexandre Bergel[1]은 산티아고 칠레 대학(University of Chile)의 Pleiad Laboratory 컴퓨터 과학과에서 교수로 재직 중이다. Alexandre는 2005년 스위스 베른(Berne) 대학에서 박사 학위를 수료하였다. 그의 박사 논문은 2006년 저명한 Ernst-Denert 상을 수여받았다. 박사 과정을 아일랜드의 Lero & Trinity College Dublin에서 첫 박사 후 과정을 완료하였고, 독일의 Hasso-Plattner Institute에서 두 번째 박사 후 과정을 완료하였다. Alexandre와 동료들은 소프트웨어 공학, 소프트웨어 품질, 좀 더 구체적으로 말해 코드 프로파일링, 테스팅, 데이터 시각화를 대상으로 연구를 실행한다. 국제 및 상호 검사 과학 포럼에 출판된 Alexandre의 글은 60편이 넘으며, 그 중에는 소프트웨어 공학 분야에서 가장 경쟁력 있는 회의나 저널도 포함되어 있다. 뿐만 아니라 Alexandre는 연구 결과를 산업으로 적용시키는 데에도 관심이 크다. 그의 연구 프로토타입 중 몇 가지는 상품화되었다.

Damien Cassou[2]는 프랑스 Lille 1 대학에서 부교수(maitre de conferences)이자 RmoD 연구 단체(Inria, LIFL)의 회원이다. 그의 연구의 주요 목표는 복잡한 소스의 브라우징부터 거대한 양의 commits를 반자동으로 분해하는 것까지 개발자들이 매일 직면하는 문제들을 해결하는 데에 있다. Damien Cassou는 RMoD에서 작업하기 전에 Bordeaux I 대학에서 컴퓨터 과학에 박사 학위를 수여받았는데, 그의 논문은 도메인 특정적 아키텍처 기술 언어와 프로그래밍 프레임워크 생성기(generator)를 통해 일반 용도의 프로그래밍 툴을 전용 도메인으로 가져오는 것에 관한 논문이었다. 그는 Pharo의 개발자 중 한 명이며, Pharo by Example book 에도 참여하였다.

[1] http://bergel.eu
[2] http://daminencassou.seasidehosting.st

Stephane Ducasse[3]는 Inria에서 연구소장이다. 2011년 이후 그는 Inria Lille Nord Europe 연구 센터의 과학 대표인으로서 그곳에서 RMod 팀을 이끌고 있다 (http://rmod.lille.inria.fr). 그의 전문 분야는 두 가지로, 객체 지향 언어 디자인과 재공학이 그것이다. 그는 traits, 구성 가능한 메서드 그룹을 작업하였고, 그의 작업은 어느 정도 영향력을 가졌다. traits는 AmbiantTalk, Racket, Squeak/Pharo, Perl, PHP, 그리고 변형되어 Scala, Fortress of SUN Microsystems에도 도입되었다. 그는 또한 소프트웨어 품질, 프로그램 이해, 프로그램 시각화, 재공학, 메타모델링에 있어서도 전문가다. 오픈소스 소프트웨어 분석 플랫폼 (http://www.moosetechnology.org) Moose 의 개발자들 중 한 명이기도 하다. Stethane 은 고급 소프트웨어 분석 전용 툴을 빌드하는 기업인 Synectique(http://www.synectique.eu)과 일한다.

Jannik Laval[4]은 2012년부터 프랑스 Mines Douai에 위치한 MinesTelecom Institute에서 부교수로 재직 중이다. 그는 2011년 6월 프랑스 University Lille 1의 컴퓨터 과학학부에서 박사 학위를 수여했다. 그의 논문의 주제는 소프트웨어 품질, 시각화, 재공학이었다. 그는 그의 모든 소프트웨어 분석에 Moose를 사용한다 (http://www.moosetechnology.org). Mines Douai에서 그는 내장 시스템에 관한 소프트웨어 공학을 작업하였는데, 좀 더 구체적으로 말해 멀티로봇 시스템을 위한 모듈성과 툴을 작업하였다. 그는 Pharo에서 적용된 시각적 프로그래밍 언어인 Phratch의 주 개발자다 (http://car.mines-douai.fr/category/phratch/). 그는 Phratch를 이용해 공학 학생들에게 로봇 소프트웨어 공학을 가르친다.

[3] http://stephane.ducasse.free.fr
[4] http://www.jannik-laval.eu