



Some Design Points

Stéphane Ducasse
Stephane.Ducasse@univ-savoie.fr
<http://www.iam.unibe.ch/~ducasse/>
Stéphane Ducasse --- 2005

The Design in Question

- The Basic Idea behind Frameworks
- Subclassing vs SubTyping
- Coupling
- Design Heuristics
- Design Symptoms



Frameworks

- What is it?
- Principles
- vs. Libraries



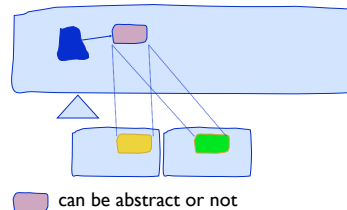
Inheritance as Parameterization

- Subclass customizes hook methods by implementing (abstract) operations in the context of template method
- Any method acts as a parameter of the context
- Methods are unit of reuse
- Abstract class -- one that must be customized before it can be used



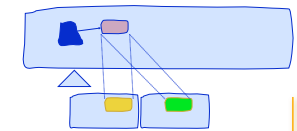
Methods are Unit of Reuse

self sends are plans for reuse



Frameworks vs. Libraries

- Libraries
 - You call them
 - Callback to extend them
- Framework
 - **Hollywood principle:** *Don't call me I will call you*
 - **GreyHound principle:** *Let's drive*



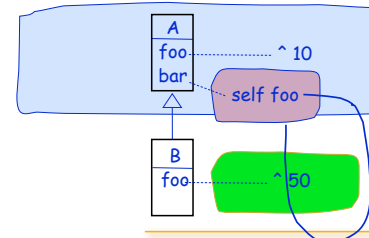
Library vs. Framework

Classes instantiated by the client	Framework instantiated classes, extended by inheritance
Clients invoke library functions	Framework calls the client functions
No predefined flow, predefined interaction, default behavior	Predefined flow, interaction and default behavior



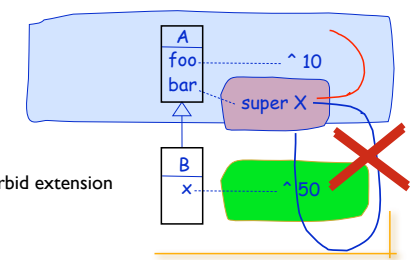
You remember self...

- self is dynamic
- self acts as a hook



You remember super...

- super is static



Frameworks

- A set of collaborating classes that define a context and are reusable by extension in different applications
- A framework is a reusable design expressed as a set of abstract classes and the way their instances collaborate. By definition, a framework is an object-oriented design. It doesn't have to be implemented in an object-oriented language, though it usually is. Large-scale reuse of object-oriented libraries requires frameworks. The framework provides a context for the components in the library to be reused. [Johnson]
- A framework often defines the architecture of a set of applications

S.Ducasse

10



On Frameworks...

- Frameworks design
 - Need at least 3 applications to support the generalization
 - <http://st-www.cs.uiuc.edu/users/droberts/evolve.html>
- Smile if somebody tell that they start implementing a framework
- Framework often rely on whitebox abstractions: ie extended by inheritance
- Others are blackboxes framework: ie extended by composition
- A framework can use design patterns

S.Ducasse

11



SubTyping vs. Subclassing



S.Ducasse

12



How to Implement a Stack?

By subclassing OrderedCollection...

```
Stack>>pop
^ self removeLast
Stack>>push: anObject
self addFirst: anObject
Stack>>top
^ self first
```

Stack>>size, Stack>>includes:
are free, inherited from

S.Ducasse

13



BUT BUT BUT!!!

- What do we do with all the rest of the interface of OrderedCollection?
- a Stack IS NOT an OrderedCollection!
- We cannot substitute an OrderedCollection by a Stack
 - Some messages do not make sense on Stack
 - Stack new addLast: anObject
 - Stack new last
- So we have to block a lot of methods...

S.Ducasse

14



Consequences...

```
Stack>>removeLast
self shouldNotImplement
```

```
Stack>>pop
^ super removeLast
```



S.Ducasse

15



The Problem

- There is not a clean simple relationship between Stack and OrderedCollection
- Stack interface is not an extension or subset of OrderedCollection interface
- Compare with CountingStack a subclass of Stack
- CountingStack is an extension

S.Ducasse

16



Another Approach

By defining the class Stack that uses OrderedCollection

```
Object subclass: Stack
iv: elements
```

```
Stack>>push: anElement
elements addFirst: anElement
Stack>>pop
element isEmpty ifFalse: [^ self removeFirst]
```

S.Ducasse

17



Inheritance and Polymorphism

- Polymorphism works best with standard interfaces
- Inheritance creates families of classes with similar interfaces
- Abstract class describes standard interfaces
- Inheritance helps software reuse by making polymorphism easier

S.Ducasse

18



Specification Inheritance

- Subtyping
- Reuse of specification
 - A program that works with Numbers will work with Fractions.
 - A program that works with Collections will work with Arrays.
- A class is an abstract data type (Data + operations to manipulate it)

S.Ducasse

19



Inheritance for Code Reuse

- Subclassing
- Dictionary is a subclass of Set
- Semaphore is a subclass of LinkedList
- No relationship between the interfaces of the classes
- Subclass reuses code from superclass, but has a different specification. It cannot be used everywhere its superclass is used. Usually overrides a lot of code.
- ShouldNotImplement use is a bad smell...

S.Ducasse

20



Inheritance for Code Reuse

- Inheritance for code reuse is good for
 - rapid prototyping
 - getting application done quickly.
- Bad for:
 - easy to understand systems
 - reusable software
 - application with long life-time.

S.Ducasse

21



Subtyping Essence

- You reuse specification
- You should be able to substitute an instance by one of its subclasses (more or less)
- There is a relationship between the interfaces of the class and its superclass

S.Ducasse

22



How to Choose?

- Favor subtyping
- When you are in a hurry, do what seems easiest.
- Clean up later, make sure classes use "is-a" relationship, not just "is-implemented-like".
- Is-a is a design decision, the compiler only enforces is-implemented-like!!!

S.Ducasse

23



Quiz

- Circle subclass of Point?
- Poem subclass of OrderedCollection?

S.Ducasse

24



Class Design



S.Ducasse

25



Behavior Up and State Down

- Define classes by behavior, not state
- Implement behavior with abstract state: if you need state do it indirectly via messages.
- Do not reference the state variables directly
- Identify message layers: implement class's behavior through a small set of kernel method

S.Ducasse

26



Example

```
Collection>>removeAll: aCollection  
aCollection do: [:each | self remove: each]  
^ aCollection
```

```
Collection>>remove: oldObject  
self remove: oldObject ifAbsent: [self notFoundError]
```

```
Collection>>remove: anObject ifAbsent:  
anExceptionBlock  
self subclassResponsibility
```

S.Ducasse

27



Behavior-Defined Class

When creating a new class, define its public protocol and specify its behavior without regard to data structure (such as instance variables, class variables, and so on).

For example:

Rectangle

Protocol:

area
intersects:
contains:
perimeter

S.Ducasse

28



Implement Behavior with Abstract State

- If state is needed to complete the implementation
- Identify the state by defining a message that returns that state instead of defining a variable.

For example, use

```
Circle>>area  
  ^self radius squared * self pi
```

not

```
Circle>>area  
  ^radius squared * pi.
```

S.Ducasse

29



Identify Message Layers

- How can methods be factored to make the class both efficient and simple to subclass?
- Identify a small subset of the abstract state and behavior methods which all other methods can rely on as kernel methods.

```
Circle>>radius  
Circle>>pi  
Circle>>center  
Circle>>diameter  
  ^self radius * 2  
Circle>>area  
  ^self radius squared * self pi
```

S.Ducasse

30



Good Coding Practices

- Good Coding Practices promote good design
- Encapsulation
- Level of decomposition
- Factoring constants



S.Ducasse

31



The Object Manifesto

- Be lazy and be private
- Never do the job that you can delegate to another one
- Never let someone else plays with your private data

S.Ducasse

32



The Programmer Manifesto

- Say something only once
- Don't ask, tell!

S.Ducasse

33



Tell, Don't Ask!

```
MyWindow>>displayObject: aGrObject  
  aGrObject displayOn: self
```

• And not:

```
MyWindow>>displayObject: aGrObject
```

```
  aGrObject isSquare ifTrue: [...]  
  aGrObject isCircle ifTrue: [...]  
  ...
```

S.Ducasse

Good Signs of OO Thinking

- Short methods
- No dense methods
- No super-intelligent objects
- No manager objects
- Objects with clear responsibilities
 - State the purpose of the class in one sentence
- Not too many instance variables

S.Ducasse

35



Composed Methods

- How do you divide a program into methods?
 - Messages take time
 - Flow of control is difficult with small methods
- But:
 - Reading is improved
 - Performance tuning is simpler (Cache...)
 - Easier to maintain / inheritance impact

S.Ducasse

36



Composed Methods

- Divide your program into methods that perform one identifiable task. Keep all of the operations in a method at the same level of abstraction.
- Controller>>controlActivity
self controlInitialize.
self controlLoop.
self controlTerminate

S.Ducasse

37



Do you See the Problem?

initializeToStandAlone

```
super initializeToStandAlone.  
self borderWidth: 2.  
self borderColor: Color black.  
self color: Color blue muchLighter.  
self extent: self class defaultTileSize * (self columnNumber @ self rowNumber).  
self initializeBots.  
self running.  
area := Matrix rows: self rowNumber columns: self columnNumber.  
area indicesDo: [:row :column | area at: row at: column  
put: OrderedCollection new].  
self fillWorldWithGround.  
self firstArea.  
self installCurrentArea
```

S.Ducasse



Do you See the Problem?

initializeToStandAlone

```
super initializeToStandAlone.  
self initializeBoardLayout.  
self initializeBots.  
self running.  
self initializeArea.  
self fillWorldWithGround.  
self firstArea.  
self installCurrentArea
```

S.Ducasse



With code reuse...

initializeArea

```
area := self matrixClass  
rows: self rowNumber  
columns: self columnNumber.  
area indicesDo: [:row :column | area  
at: row  
at: column  
put: OrderedCollection new]
```

initializeArea can be invoke **several** times

S.Ducasse



About Methods

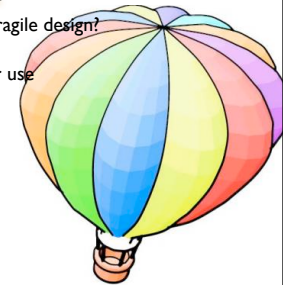
- Avoid long methods
- A method: one task
- Avoid duplicated code
- Reuse Logic

S.Ducasse



About Coupling

- Why coupled classes is fragile design?
- Law of Demeter
- Thoughts about accessor use

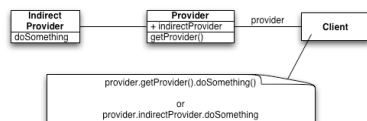


S.Ducasse

42



The Core of the Problem



S.Ducasse



The Law of Demeter

You should only send messages to:

- an argument passed to you
- an object you create
- self, super
- your class

Avoid global variables
Avoid objects returned from message sends other than self

S.Ducasse

44



Correct Messages

```
someMethod: aParameter  
self foo.  
super someMethod: aParameter.  
self class foo.  
self instVarOne foo.  
instVarOne foo.  
self classVarOne foo.  
classVarOne foo.  
aParameter foo.  
thing := Thing new.  
thing foo
```

S.Ducasse

45



Law of Demeter by Example

```

NodeManager>>declareNewNode: aNode
[nodeDescription]
(aNode isValid)           "Ok passed as an
argument to me"
ifTrue: [aNode certified].
nodeDescription := NodeDescription for: aNode.
nodeDescription localTime. "I created it"
self addNodeDescription: nodeDescription.

"I can talk to myself"
nodeDescription data
at: self creatorKey
put: self creator
    
```

"Wrong I should not know"
"that data is a dictionary"

S.Ducasse



In other words

- Only talk to your immediate friends.
- In other words:
 - You can play with yourself. (this.method())
 - You can play with your own toys (but you can't take them apart). (field.method(), field.getX())
 - You can play with toys that were given to you. (arg.method())
 - And you can play with toys you've made yourself. (A a = new A(); a.method())

S.Ducasse

47



Halt!

```

class A {public: void m(); P p(); B b; };
class B {public: C c; };
class C {public: void foo(); };
class P {public: Q q(); };
class Q {public: void bar(); };
void A::m() {
    this.b.c.foo(); this.p.q().bar();}
    
```

STOP STOP

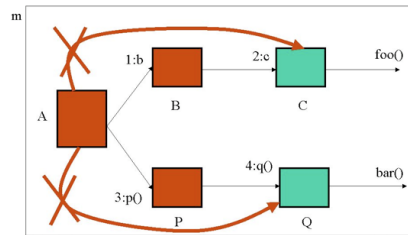
S.Ducasse

48



To not skip your intermediate

Violations: Dataflow Diagram



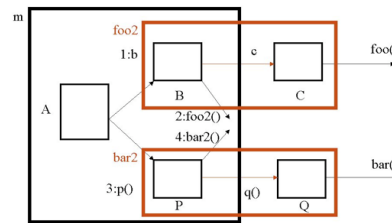
S.Ducasse

49



Solution

OO Following of LoD

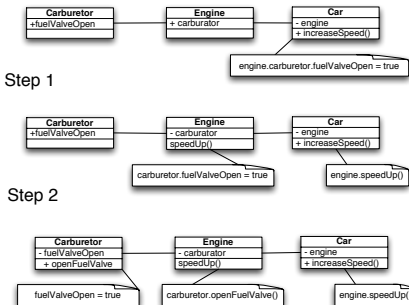


S.Ducasse

50



Transformation



S.Ducasse

51



Law of Demeter's Dark Side

```

Class A
  instVar: myCollection

A>>do: aBlock
  myCollection do: aBlock
A>>collect: aBlock
  ^ myCollection collect: aBlock
A>>select: aBlock
  ^ myCollection select: aBlock
A>>detect: aBlock
  ^ myCollection detect: aBlock
A>>isEmpty
  ^ myCollection isEmpty
    
```

S.Ducasse

52



About the Use of Accessors

Some schools say: "Access instance variables using methods"

But

Be consistent inside a class, do not mix direct access and accessor use

First think accessors as private methods that should not be invoked by clients

Only when necessary put accessors in accessing protocol

S.Ducasse

53



Example

```

Scheduler>>initialize
  self tasks: OrderedCollection new.
    
```

```

Scheduler>>tasks
  ^ tasks
    
```

But now everybody can tweak the tasks!

S.Ducasse

54



Accessors

Accessors are good for lazy initialization

```
Scheduler>>tasks
tasks isNil ifTrue:[task := ...].
^ tasks
```

BUT accessors methods should be PRIVATE by default at least at the beginning

S.Ducasse

55



Accessors open Encapsulation

The fact that accessors are methods doesn't support a good data encapsulation.

You could be tempted to write in a client:

```
ScheduledView>>addTaskButton
...
model tasks add: newTask
```

What's happen if we change the representation of tasks?

S.Ducasse

56



Tasks

If tasks is now an array it will break

Take care about the coupling between your objects and provide a good interface!

```
Schedule>>addTask: aTask
tasks add: aTask
```

```
ScheduledView>>addTaskButton
...
model addTask: newTask
```

S.Ducasse

57



About Copy Accessor

Should I copy the structure?

```
Scheduler>>tasks
^ tasks copy
```

But then the clients can get confused...

Scheduler uniqueInstance tasks removeFirst and nothing happens!

S.Ducasse

58



Use intention revealing names

Better

```
Scheduler>>taskCopy
"returns a copy of the pending tasks"
```

```
^ task copy
```

S.Ducasse

59



Provide a Complete Interface

```
Workstation>>accept: aPacket
aPacket addressee = self name
...
```

It is the responsibility of an object to propose a complete interface that protects itself from client intrusion.

Shift the responsibility to the Packet object

```
Packet>>isAddressedTo: aNode
^ addressee = aNode name
Workstation>>accept: aPacket
(aPacket isAddressedTo: self)
```

S.Ducasse

60



Open-Close



- Software entities (classes, modules, functions, etc.) should be **open** for extension, but **closed** for modification.

S.Ducasse

61



The open-closed principle

- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.
- Existing code should not be changed – new features can be added using inheritance or composition.

S.Ducasse

62



One kind of application

```
enum ShapeType {circle, square};
struct Shape {
  ShapeType _type;
};
struct Circle {
  ShapeType _type;
  double _radius;
  Point _center;
};
struct Square {
  ShapeType _type;
  double _side;
  Point _topLeft;
};
void DrawSquare (struct Square*)
void DrawCircle (struct Circle*);
```

S.Ducasse

63



Example (II)

```
void DrawAllShapes(struct Shape* list[], int n) {
    int i;
    for (i=0; i<n; i++) {
        struct Shape* s = list[i];
        switch (s->_type) {
            case square: DrawSquare((struct Square*)s); break;
            case circle: DrawCircle((struct Circle*)s); break;
        }
    }
}
```

Adding a new shape requires adding new code to this method.

S.Ducasse

64



Correct Form

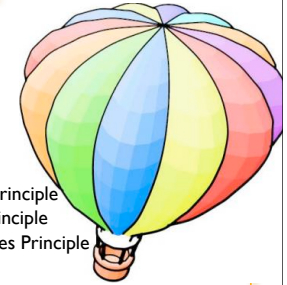
```
class Shape {
public: virtual void Draw() const = 0;
};
class Square : public Shape {
public: virtual void Draw() const;
};
class Circle : public Shape {
public: virtual void Draw() const;
};
void DrawAllShapes(Set<Shape*> &list) {
    for (Iterator<Shape*> i(list); i; i++)
        (*i)->Draw();
}
```

S.Ducasse

65



Some Principles



- Dependency Inversion Principle
- Interface Segregation Principle
- The Acyclic Dependencies Principle

S.Ducasse

66



Dependency Inversion Principle

- High level modules should not depend upon low level modules. Both should depend upon abstractions.
- Abstractions should not depend upon details. Details should depend upon abstractions.

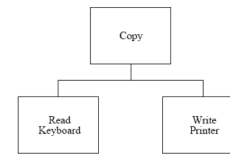
S.Ducasse

67



Example

```
void Copy() {
    int c;
    while ((c = ReadKeyboard()) != EOF)
        WritePrinter(c);
}
```



S.Ducasse

68



Cont...

Now we have a second writing device – disk

```
enum OutputDevice {printer, disk};
```

```
void Copy(outputDevice dev) {
    int c;
    while ((c = ReadKeyboard()) != EOF)
        if (dev == printer)
            WritePrinter(c);
        else
            WriteDisk(c);
}
```

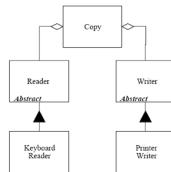
S.Ducasse

69



Solution

```
class Reader {
public:
    virtual int Read() = 0;
};
class Writer {
public:
    virtual void Write(char) = 0;
};
void Copy(Reader& r,
         Writer& w) {
    int c;
    while ((c=r.Read()) != EOF)
        w.Write(c);
}
```



S.Ducasse

70



Some Principle

-
-
-
-
-
-
-



S.Ducasse

71



Interface Segregation Principle

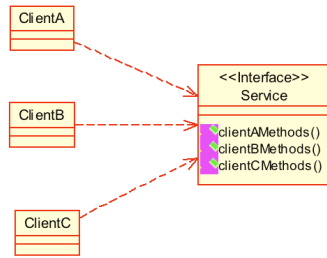
- The dependency of one class to another one should depend on the smallest possible interface.
- Avoid “fat” interfaces

S.Ducasse

72



Examples



S.Ducasse

73



Solutions

- One class one responsibility
- Composition?

- Design is not simple

S.Ducasse

74



The Acyclic Dependency Principle

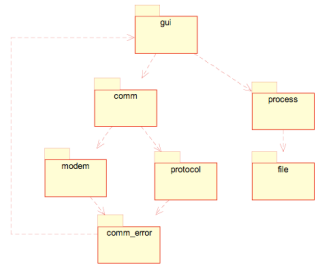
- The dependency structure between packages must not contain cyclic dependencies.

S.Ducasse

75



Example...Ez



S.Ducasse

76



Solutions

- Layering?
- Separation of domain/application/UI

S.Ducasse

77



Packages, Modules and other

- The Common Closure Principle
 - Classes within a released component should share common closure. That is, if one needs to be changed, they all are likely to need to be changed.
- The Common Reuse Principle
 - The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.

S.Ducasse

78



Summary

Build your own taste
Analyze what you write and how?

S.Ducasse

79

