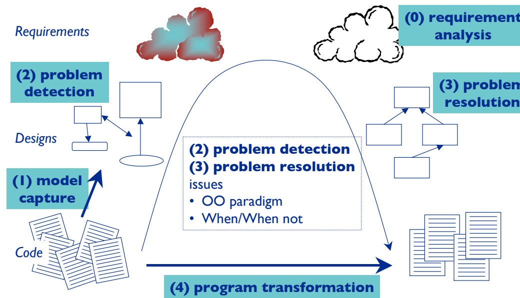
6. Restructuring

- Most common situations
- *Redistribute Responsibilities*
  + Eliminate Navigation Code
  + Move Behaviour Close to Data
  + Split up God Class
- Transform Conditionals to Polymorphism
  + Transform Self Type Checks
  + Transform Provider Type Checks
  + Transform Conditionals in Registration

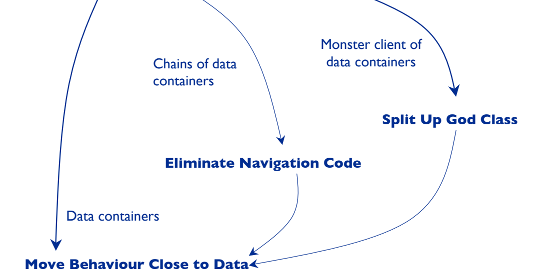
© S. Demeyer, S. Ducasse, O. Nierstrasz

The Reengineering Life-Cycle



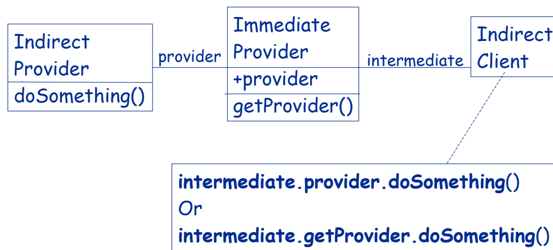
© S. Demeyer, S. Ducasse, O. Nierstrasz

Redistribute Responsibilities



© S. Demeyer, S. Ducasse, O. Nierstrasz

The Core of the Problems



Law of Demeter

© S. Demeyer, S. Ducasse, O. Nierstrasz

Move Behavior Close to Data

Problem: How do you transform a *data* container into a *service* provider

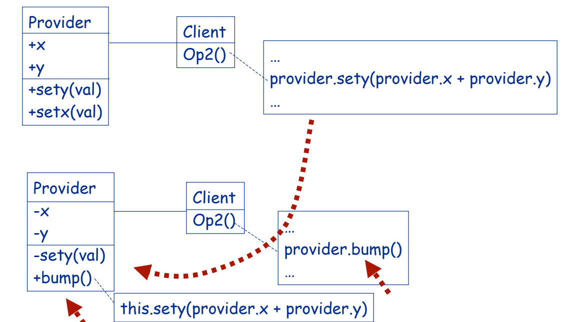
Answer: Move behavior defined by *indirect* clients to the class defining the data they manipulate

…*however*

+ Visitor
+ Difficult to identify client code to be moved in
  - Responsibility of the provider
  - Access attributes of the provider
  - Accessed by multiple clients

© S. Demeyer, S. Ducasse, O. Nierstrasz

Transformation…



© S. Demeyer, S. Ducasse, O. Nierstrasz

Detection

- Look for data containers
  + Classes with only accessors
- Duplicated client code
- Methods using sequence of accessors

© S. Demeyer, S. Ducasse, O. Nierstrasz

Difficulties

- When the moved behavior accessed *client data*, having extra parameters can lead to complex interface
- Certain classes (Set or Stream) are data containers. Move functionality to provider if
  + It represents a provider responsibility
  + It accesses attributes of the provider
  + The same behavior defined in multiple clients

© S. Demeyer, S. Ducasse, O. Nierstrasz

When Legacy Solution is not a Problem

- Visitor typically defines behavior that acts on another class
- Configuration classes (global settings, language dependent information..)
- Mapping classes between objects and UI or databases representation

© S. Demeyer, S. Ducasse, O. Nierstrasz

Eliminate Navigation Code

Problem: How do you *reduce* the *coupling* due to classes that navigate object graph?

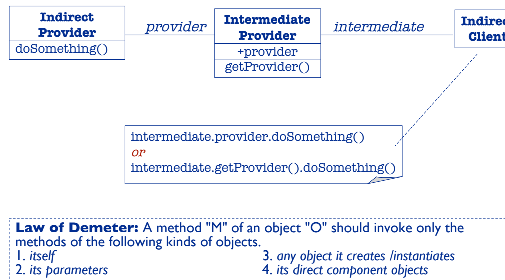
Answer: iteratively move behavior close the data

…however

+ Systematic uses produce large interfaces (shield collections)
+ a.k.a Law of Demeter

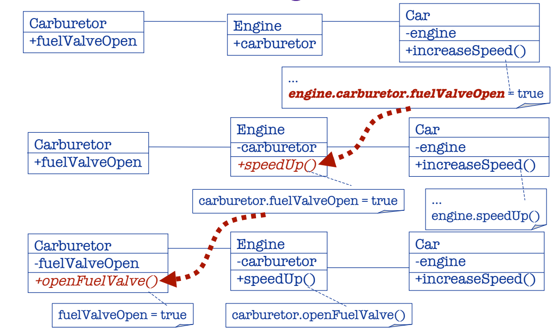
© S. Demeyer, S. Ducasse, O. Nierstrasz

The Law of Demeter



© S. Demeyer, S. Ducasse, O. Nierstrasz

Eliminate Navigation Code



© S. Demeyer, S. Ducasse, O. Nierstrasz

Detection

- Class with lot of accessors few methods
- Each time a class changes, indirect clients get impacted
- a.b.c.d.op() identified by
  + egrep '.*\..*\..*\..' *.java
- anObject.m1().m2().op() identified by
  + egrep '.*\(\).*\(\).*\(\).' *.java

© S. Demeyer, S. Ducasse, O. Nierstrasz

Detection (ii)

- Not a problem
  + (a.isNode()) & (a.isAbstract())
- Disguise Navigation

```
Token token;
token = parseTree.token();
if (token.identifier() != null){…
⇔
if(parseTree.token().identifier() != null){…
```

© S. Demeyer, S. Ducasse, O. Nierstrasz

When the Legacy Solution is the Solution

- User Interfaces or databases may need to have access to indirect providers
- Brokers or object servers are special objects returning objects

© S. Demeyer, S. Ducasse, O. Nierstrasz

Law of Demeter's Dark Side

- Produces large interfaces
- Class A

```
    instVar: myCollection
  A>>do: aBlock
        myCollection do: aBlock
  A>>collect: aBlock
        myCollection collect: aBlock
    …
    …
    …
```

© S. Demeyer, S. Ducasse, O. Nierstrasz

Split Up Good Class

a.k.a: God Class [Riel96]

Problem: How to *break* a class that controls the *complete* system logic?

Answer: Incrementally distribute responsibilities into slave classes

…however it is difficult to

+ Identify abstractions in blob
+ Limit impact of changes on other parts

© S. Demeyer, S. Ducasse, O. Nierstrasz

Detection

- *Huge* and *monolithic* class with no clear and simple responsibility
- "The heart of the system"
- One *single* class contains *all* the logic and control flow
- Classes only serve as passive data holder
- Manager, System, Root, *Controller*,
- Introducing changes always requires to *change* the same class

© S. Demeyer, S. Ducasse, O. Nierstrasz

Transformation

- Difficult because God Class is a usually a huge blob
- Identify cohesive set of attributes and methods
  - Create classes for these sets
- Identify all classes used as data holder and analyze how the god class use them
  - *Move Behavior close to the Data*
- Try to always have a running system before decomposing the God Class
  - Use accessors to hide the transformation
  - Use method delegation from the God Class to the providers
  - Use *Façade* to minimize change in clients

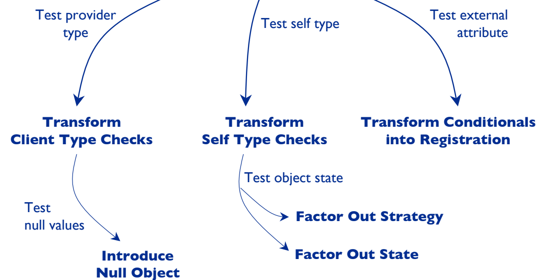
Strategies

- If God Class does not need to be changed *don't* touch it!
- *Wrap* it with different OO views
  - but a God Class usually defines the control flow of the application

Roadmap

- Most common situations
- **Redistribute Responsibilities**
  - Move Behaviour Close to Data
  - Eliminate Navigation Code
  - Split up God Class
- **Transform Conditionals to Polymorphism**
  - Transform Self Type Checks
  - Transform Provider Type Checks
  - Transform Conditionals in Registration

Transform Conditionals to Polymorphism



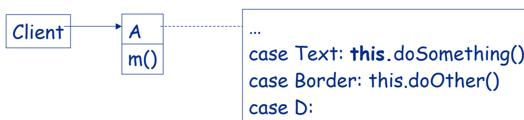
Forces

- Requirements change, so *new classes* and *new methods* will have to be introduced
- Adding new classes may *clutter the namespace*
- *Conditionals* group all the variant in one place but *make changes difficult*
  - Conditionals clutter logic
  - Editing several classes and fixing case statements to introduce a new behavior is error prone

Overview

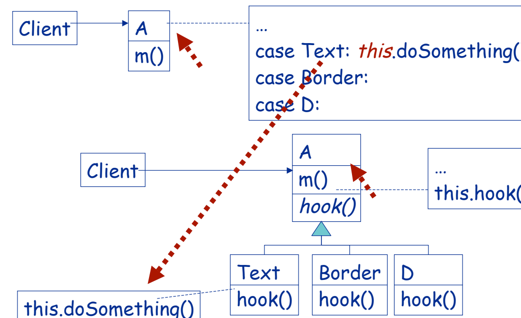
- **Transform Self Type Checks** eliminates conditionals over *type* information in a provider by introducing new subclasses
- **Transform Client Checks** eliminates conditionals over *client* type information by introducing new method to each provider classes
- **Factor out State** (kind of Self Type Check)
- **Factor out Strategy** (kind of Self Type Check)
- **Introduce Null Object** eliminates *null tests* by introducing a Null Object
- **Transform Conditionals into Registration** eliminates conditional by using a registration mechanism

Transform Self Type Checks



- Symptoms
  - Simple extensions require many changes in conditional code
  - Subclassing impossible without duplicating and updating conditional code
  - Adding new case to conditional code

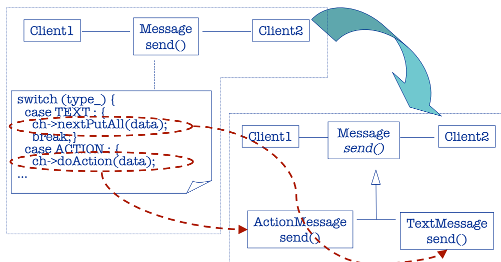
Transformation



Example: Transform *Self* Type Checks

```
class Message {
  private:
   int type_; void* data;
...
void send (Channel* ch) {
   switch (type_) {
     case TEXT : {
       ch->nextPutAll(data);
       break;
     }
     case ACTION : {
       ch->doAction(data); ...
```

Transform Self Type Check



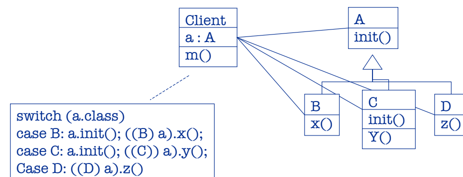
Detection

- Long methods with complex decision logic
  + Look for attribute set in constructors but never changed
  + Attributes to model type or finite set constants
  + Multiple methods switch on the same attribute
  + grep switch 'find . -name "*.cxx" -print'

Pros/Cons/Difficulties

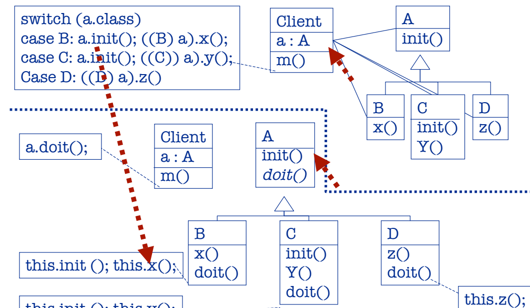
- Pros
  + New behavior are easy to add and to understand: a new class
  + No need to change different method to add a behavior
  + All behaviors share a common interface
- Cons
  + Behavior are dispersed into multiple but related abstractions
  + More classes
- Difficulties
  + Not always one to one mapping between cases and subclasses
  + Clients may be changed to create instance of the right subclass

Transform Client Type Checks



- Symptoms
  + Clients perform explicit type checks / type coercions
  + Adding a new provider (A subclass) ⇒ change all clients
  + *Clients* are defining logic about providers

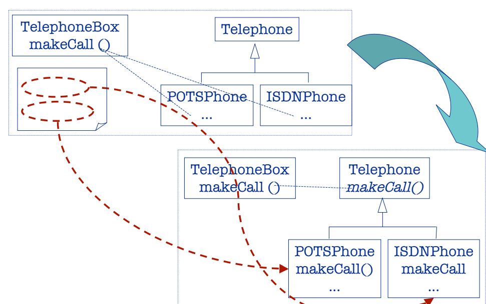
Transformation



Example: Transform *Client* Type Checks

```
void makeCalls (Telephone*
    phoneArray[]) {
for (Telephone *p = phoneArray;
    p; p++) {
    switch (p-> phoneType()) {
    case TELEPHONE::POTS : {
        POTSPhone* potsp =
            (POTSPhone*)p
        potsp->tourne();
        potsp->call();...
    case TELEPHONE::ISDN : {
        ISDNPhone* isdnp =
            (ISDNPhone*)p
        isdnp->initLine();
        isdnp->connect();...
```


Transform Client Type Check



Detection

- Transform Self Type Checks
- Changing clients of method when new case added
- Attribute representing a type
  In Smalltalk: isKindOf:, isMemberOf:
- In Java: instanceof
- x.getClass() == y.getClass()
- x.getClass().getName().equals(….)

Pros/Cons/Difficulties

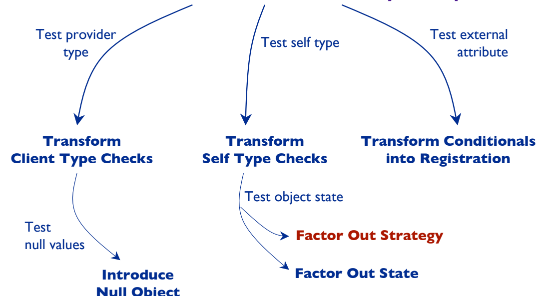
- Pros
  + The provider offers now a polymorphic interface that can be used by other clients
  + A class represent one case
  + Clients are not responsible of provider logic
  + Adding new case does not impact all clients
- Cons
  + Behavior is not group per method but per class
- Difficulties
  + Refactor the clients (Deprecate Obsolete Interfaces)
  + Instance creation should not be a problem

When the Legacy Solution is the Solution

- Abstract Factory may need to check a type variable to know which class to instantiate.
  - + For example streaming objects from a text file requires to know the type of the streamed object to recreate it
- If provider hierarchy is *frozen* (Wrapping the classes could be a good migration strategies)
- Software that interfaces with *non-oo* libraries (switch to simulate polymorphic calls)

© S. Demeyer, S. Ducasse, O. Nierstrasz

Transform Conditionals to Polymorphism



© S. Demeyer, S. Ducasse, O. Nierstrasz

Factor Out Strategy

Problem:

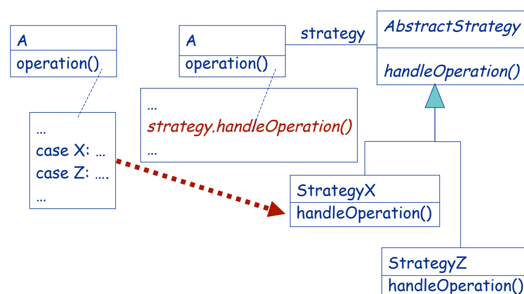
+ How do you make a class whose behavior depends on testing certain value more extensible

Answer:

+ Apply State Pattern
+ Encapsulate the behavior and delegate using a polymorphic call

© S. Demeyer, S. Ducasse, O. Nierstrasz

Transformation



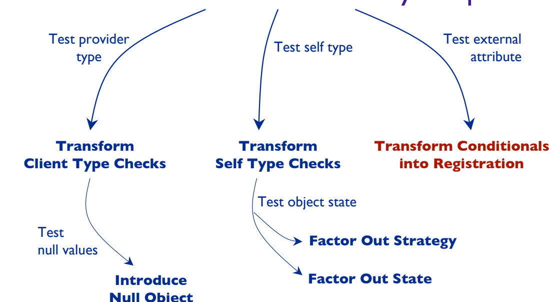
© S. Demeyer, S. Ducasse, O. Nierstrasz

Pros/Cons/Difficulties

- Pros
  - + Behavior extension is well-identified
  - + Behavior using the extension is clearer
  - + Change behavior at *run-time*
- Cons
  - + Namespace get cluterred
  - + Yet another *indirection*
- Difficulties
  - + Behavior can be difficult to convert and encapsulate (passing parameter…)

© S. Demeyer, S. Ducasse, O. Nierstrasz

Transform Conditionals to Polymorphism



© S. Demeyer, S. Ducasse, O. Nierstrasz

Transform Conditional into Registration

Problem:

+ How do you reduce the coupling between *tools* providing services and *clients* so that addition/removal of tools does not change client code?

Solution: Introduce a registration mechanism

+ Tools register/unregister
+ Clients query them via the registration repository

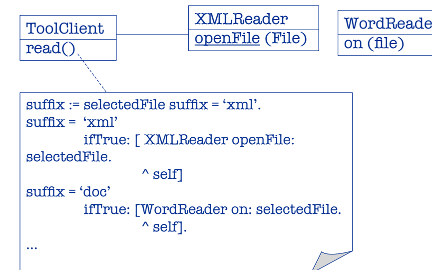
© S. Demeyer, S. Ducasse, O. Nierstrasz

Symptoms

- Long method in clients checking which tools to invoke based *on external* properties e.g., file extension
- Removing or adding a tool *force to change client code*
- Difficulty to have *run-time* tool loading / unloading

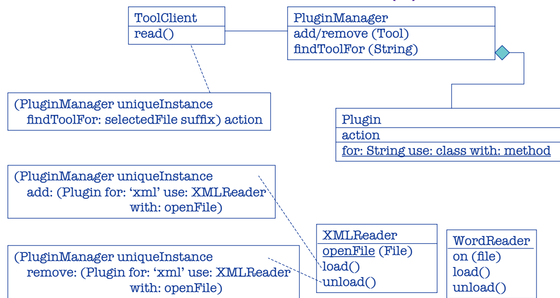
© S. Demeyer, S. Ducasse, O. Nierstrasz

Transformation (i)



© S. Demeyer, S. Ducasse, O. Nierstrasz

Transformation (ii)



© S. Demeyer, S. Ducasse, O. Nierstrasz

Pros/Cons/Difficulties

Pros

- + New tools can be added without impacting clients
- + Interaction between tools and clients is normalized
- + Reduce coupling and support modular design

Cons

- + Every tool should register and unregister

Difficulties

- + Action should be defined on the tool and not the client anymore, information should be passed from the client to the tool
- + Client knew statically the tools, now it is dynamic so more effort for UI (i.e., consistent menu ordering)

© S. Demeyer, S. Ducasse, O. Nierstrasz

Introduce NullObject

Problem:

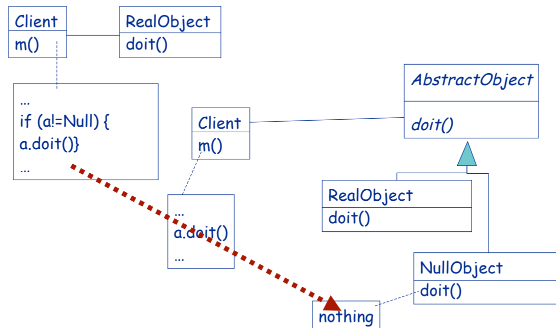
- + How can you avoid repeated tests for null values?

Answer:

- + Encapsulate the null behavior as a separate class that is polymorphic to the provider

© S. Demeyer, S. Ducasse, O. Nierstrasz

Transformation



© S. Demeyer, S. Ducasse, O. Nierstrasz

Pros/Cons/Discussions

- Pros
  - + Clients do not need to test for null values
- Difficulties
  - + Different clients may have *different* null behavior
  - + In strongly typed languages, you have to introduce Null interface
- Discussions
  - + The NullObject does not have to be a subclass of RealObject superclass as soon as it implements RealObject's null interface (in Java and Smalltalk)
- Do not apply when
  - + Very little code uses direct variable access
  - + Code that checks is *well-encapsulated in a single* place

© S. Demeyer, S. Ducasse, O. Nierstrasz

Conclusion

- Navigation Code & Complex Conditionals
  - + Most common lack of OO use
- Polymorphism is key abstraction mechanism
  - + adds flexibility ⇒ reduces maintenance cost
- Avoid Risk
  - + Only refactor when inferior code must be changed (cf. God Class)
- Performance?
  - + Small methods with less navigation code are easier to optimise
  - + Deeply nested if-statements *cost more* than virtual calls
    Long case statements cost *as much* as virtual calls

© S. Demeyer, S. Ducasse, O. Nierstrasz