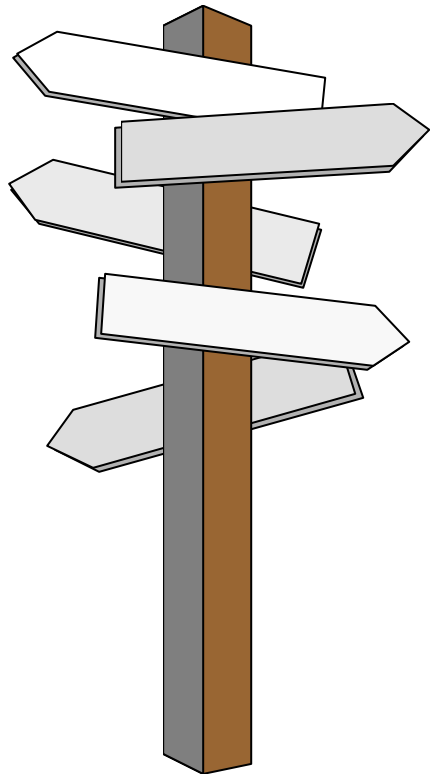
Quality-Driven Code Restructuring

Refactoring

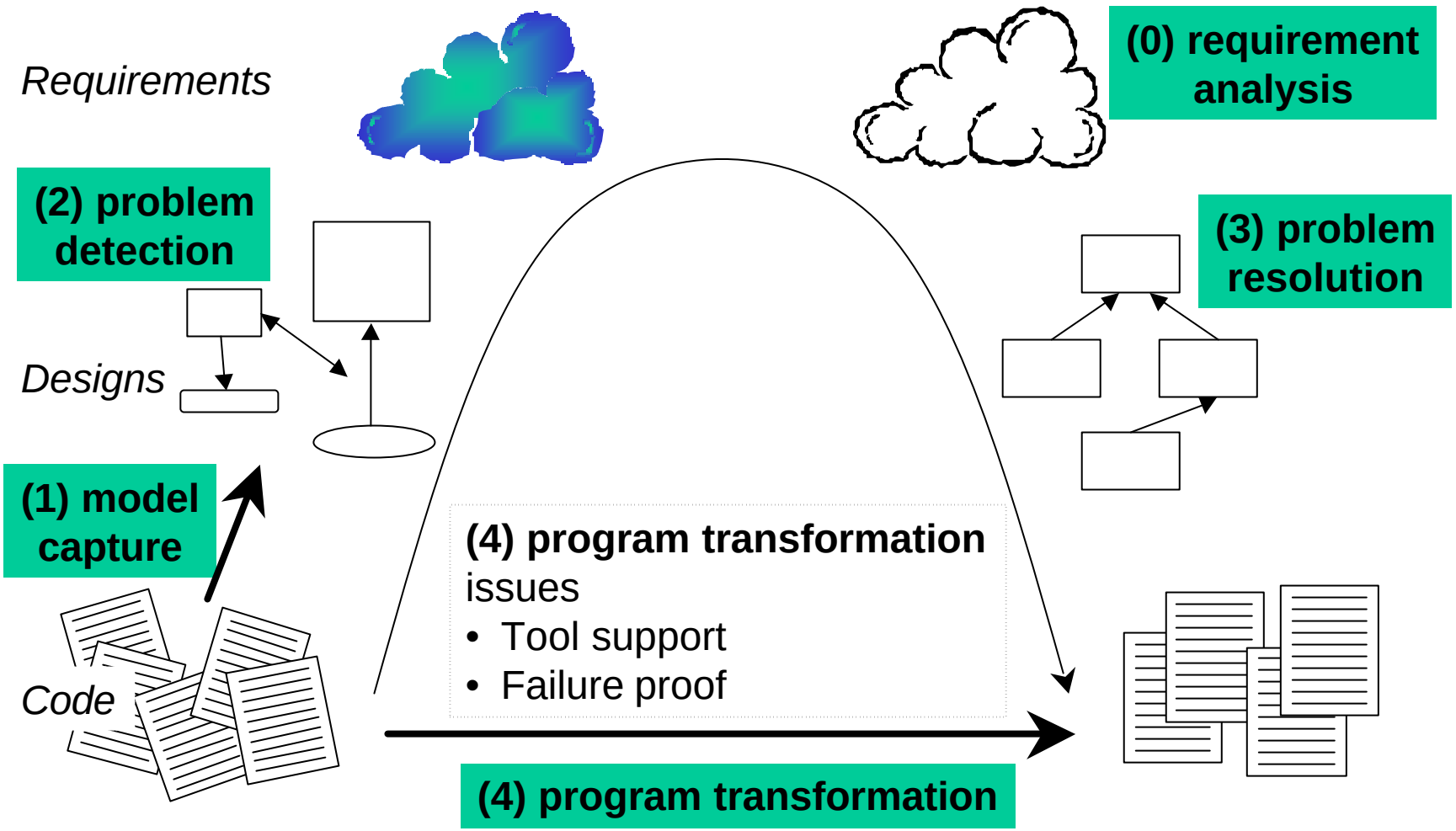
Refactoring
- What is it?
- Why is it necessary?
- Examples
- Tool support

Refactoring Strategy
- Code Smells
- Examples of Cure

Conclusions
- Obstacle-driven Conclusions



The Reengineering Life-Cycle



What is Refactoring?

The process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure [Fowl99a]

A change to the system that leaves its behaviour unchanged, but enhances some non-functional quality - simplicity, flexibility, understandability, ... [Beck99a]

Typical Refactorings

Class Refactorings	Method Refactorings	Attribute Refactorings
add (sub)class to hierarchy	add method to class	add variable to class
rename class	rename method	rename variable
remove class	remove method	remove variable
	push method down	push variable down
	push method up	push variable up
	add parameter to method	create accessors
	move method to component	abstract variable
	extract code in new method	

These simple refactorings can be combined to provide bigger restructurings such as the introduction of design patterns.

Why Refactoring?

> *Grow, don't build software*
> F.P. Brooks jr

Some argue that good design does not lead to code needing refactoring,

But in reality

- Extremely difficult to get the design right the first time
- You cannot fully understand the problem domain
- You cannot understand user requirements, if the user does!
- You cannot really plan how the system will evolve in five years
- Original design is often inadequate
- System becomes difficult to change

Refactoring helps you to

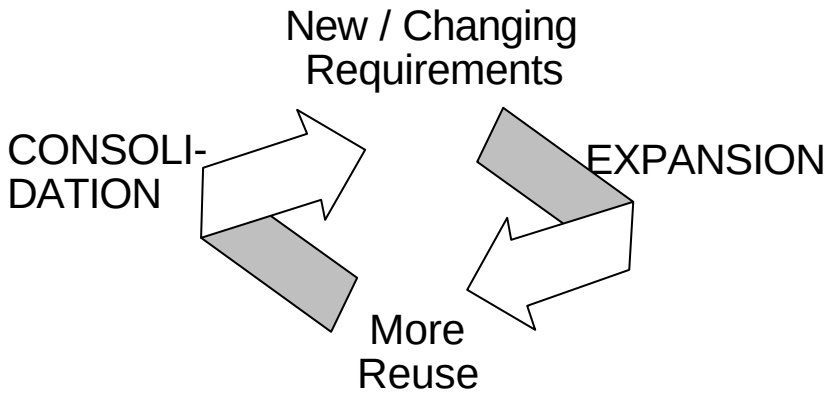
- Manipulate code in a safe environment (behavior preserving)
- Recreate a situation where evolution is possible
- Understand existing code

Refactoring and OO

Object-Oriented Programming

- emphasize the possibility of changes
- rapid development cycle
- incremental definition

Iterative development

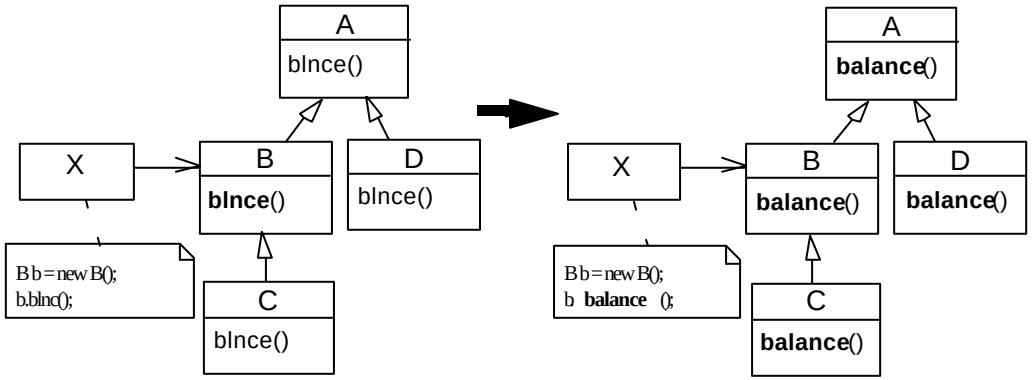


Consolidation is necessary to ensure next expansion success

However software evolves, grows and... dies if not taken care of

=> This is where Refactoring comes in

Rename Method: Do It Yourself



- Do it yourself approach
- Check if a method does not exist in the class and superclass/subclasses with the same “name”
- Browse all the implementers (method definitions)
- Browse all the senders (method invocations)
- Edit and rename all implementers
- Edit and rename all senders
- Remove all implementers
- Test
- Automated refactoring is better !

Rename Method

`Rename Method (method, new_name)`

Preconditions

- no method exists with the signature implied by *new_name* in the inheritance hierarchy that contains *method*
- [Java, C++] *method* is not a constructor

Postconditions

- *method* has *new name*
- relevant methods in the inheritance hierarchy have *new name*
- invocations of changed method are updated to *new name*

Other Considerations

- Statically/Dynamically Typed Languages
  => Scope of the renaming

Which Refactoring Tools?

Change Efficient

Refactoring

- Source-to-source program transformation
- Behaviour preserving

=> improve the program structure

Programming Environment

- Fast edit-compile-run cycles
- Integrated into your environment
- Support small-scale reverse engineering activities

=> convenient for “local” ameliorations

Failure Proof

Regression Testing

- Repeating past tests
- Tests require no user interaction
- Answer per test is yes / no

=> verify if improved structure does not damage previous work

Configuration & Version Management

- keep track of versions that represent project milestones

=> possibility to go back to previous version

Conclusion: Tool Support

Refactoring Philosophy

combine simple refactorings into larger restructuring
=> improved design
=> better understandable
=> ready to add functionality

Do not apply refactoring tools in isolation

	C++	Java
refactoring tools	- (?)	+
rapid edit-compile-run cycles	-	+-
reverse engineering facilities	+-	+-
regression testing	+	+
version & configuration management	+	+

Curing Duplicated Code

> *Say everything exactly once*
>
> Kent Beck

In the same class

- Extract Method

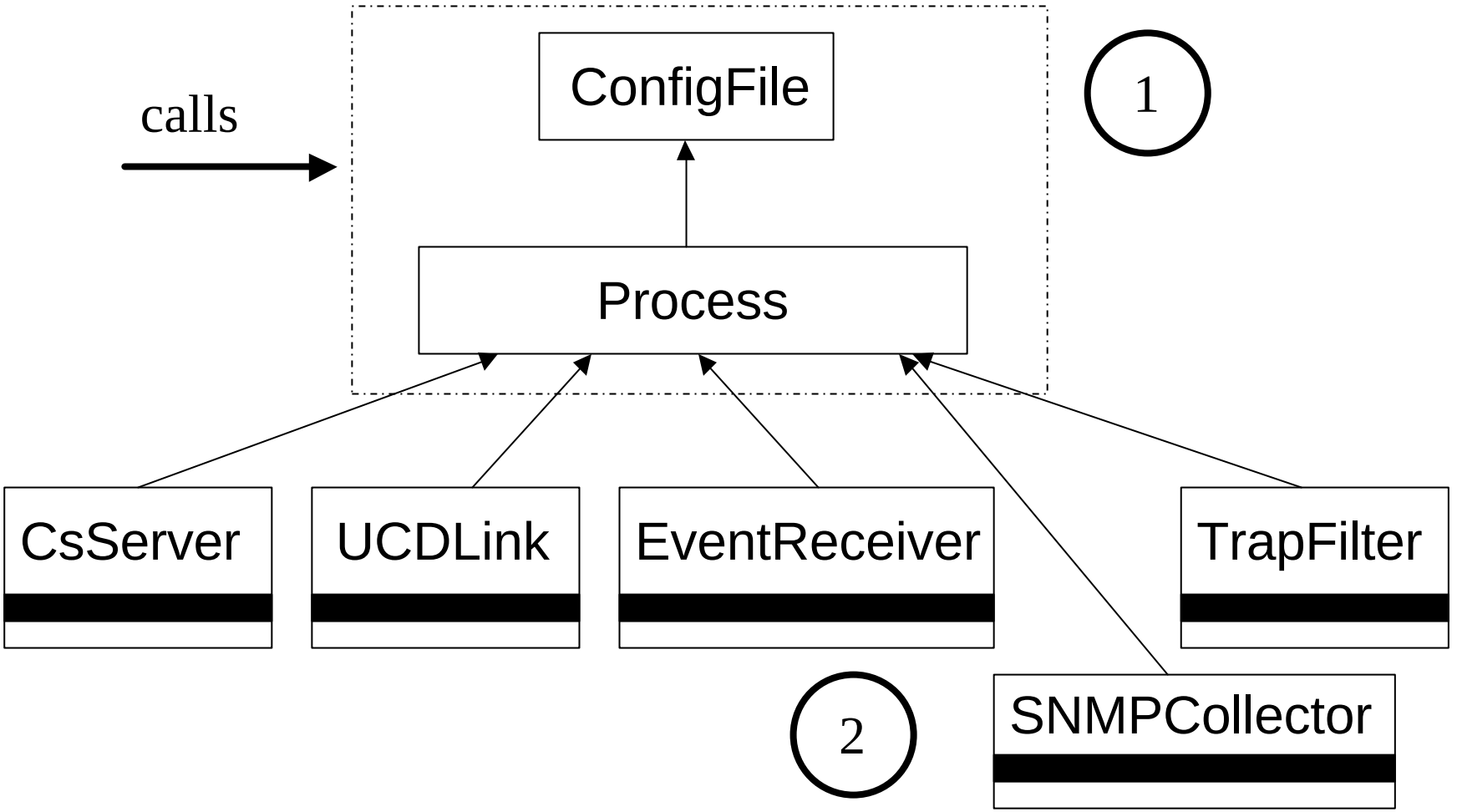
Between two sibling subclasses

- Extract Method
- Push identical methods up to common superclass
- Form Template Method

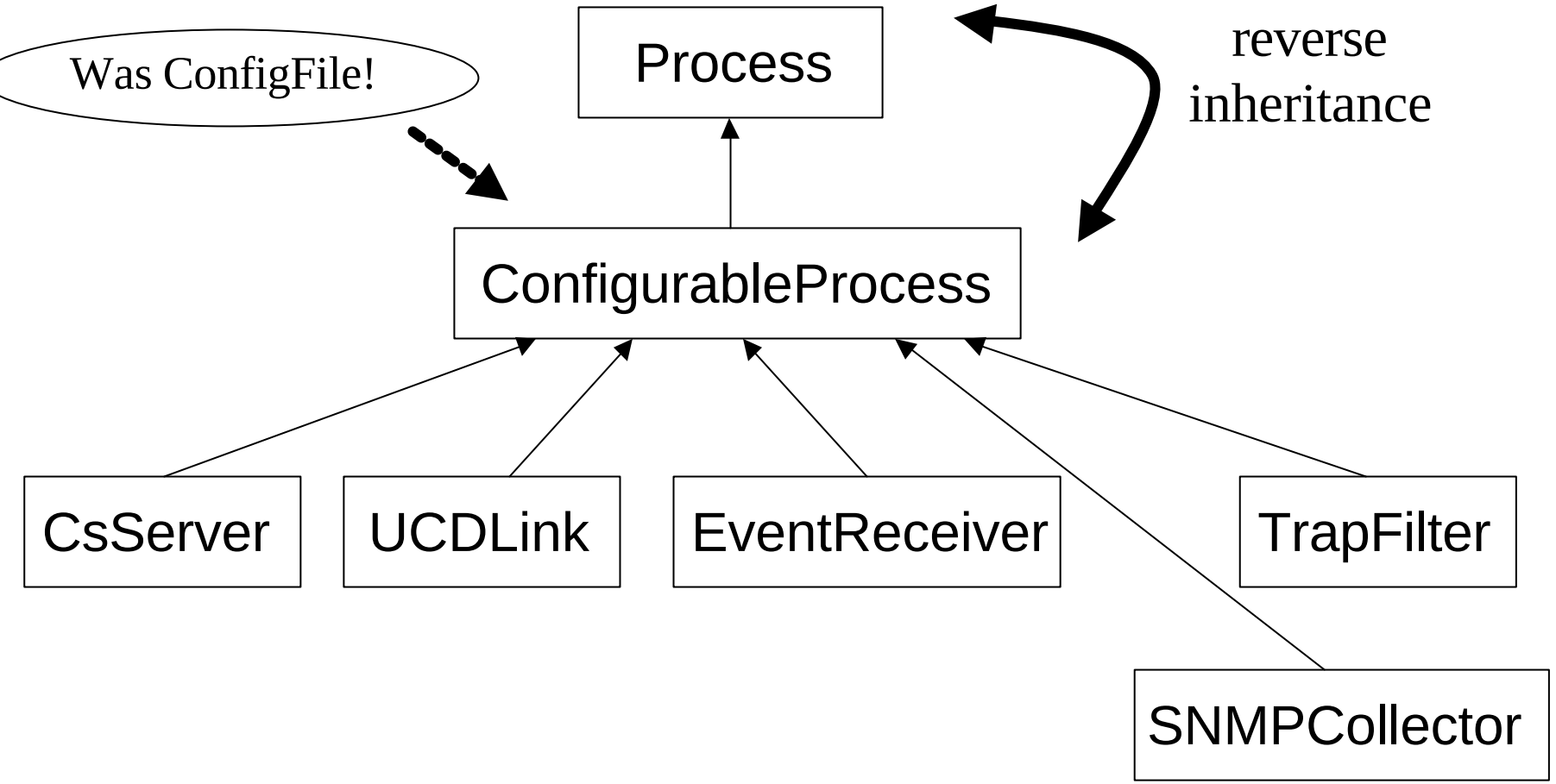
Between unrelated class

- Create common superclass
- Extract Component (e.g., Strategy)

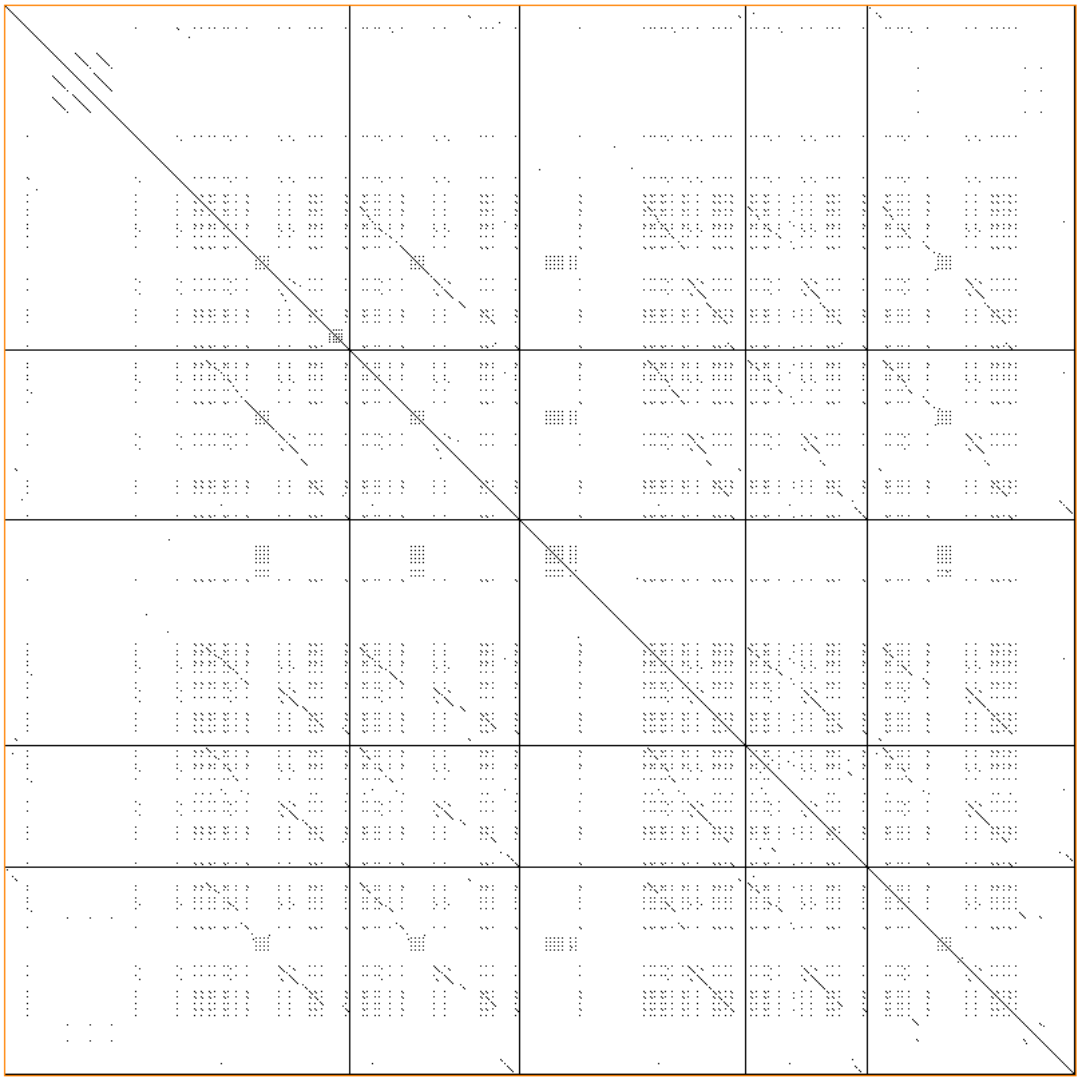
Design



Resolving the Inheritance

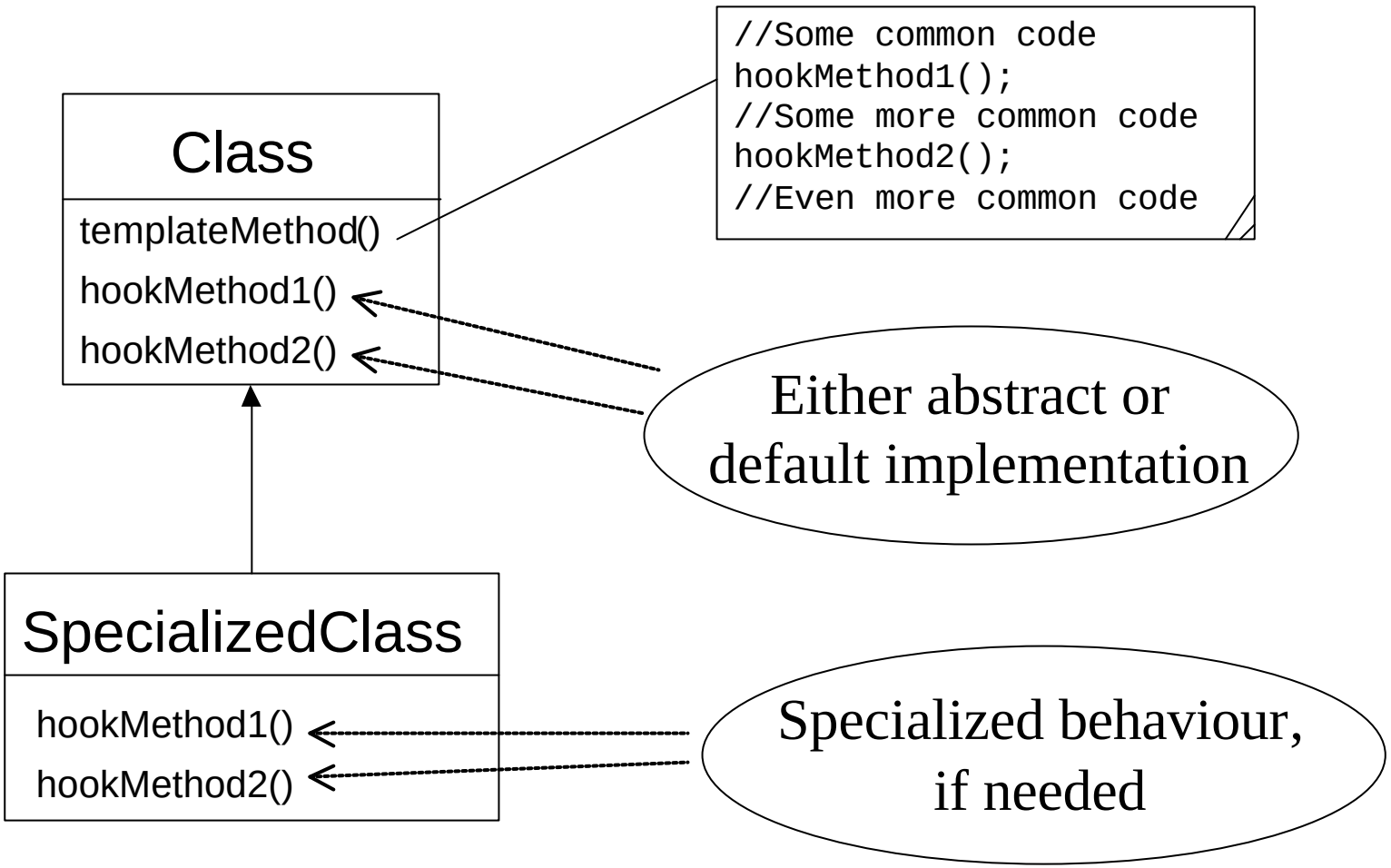


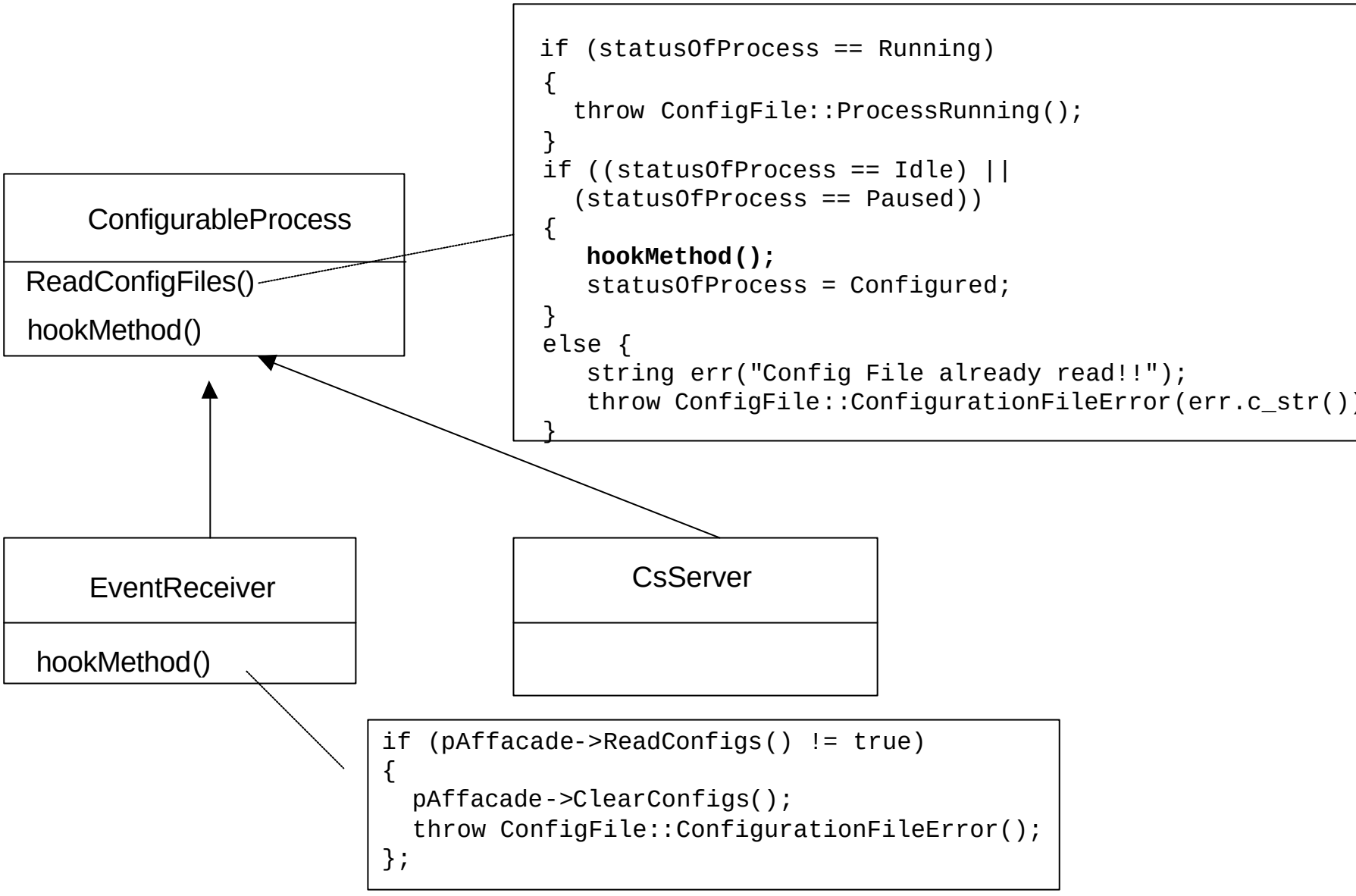
Repeated Functionality



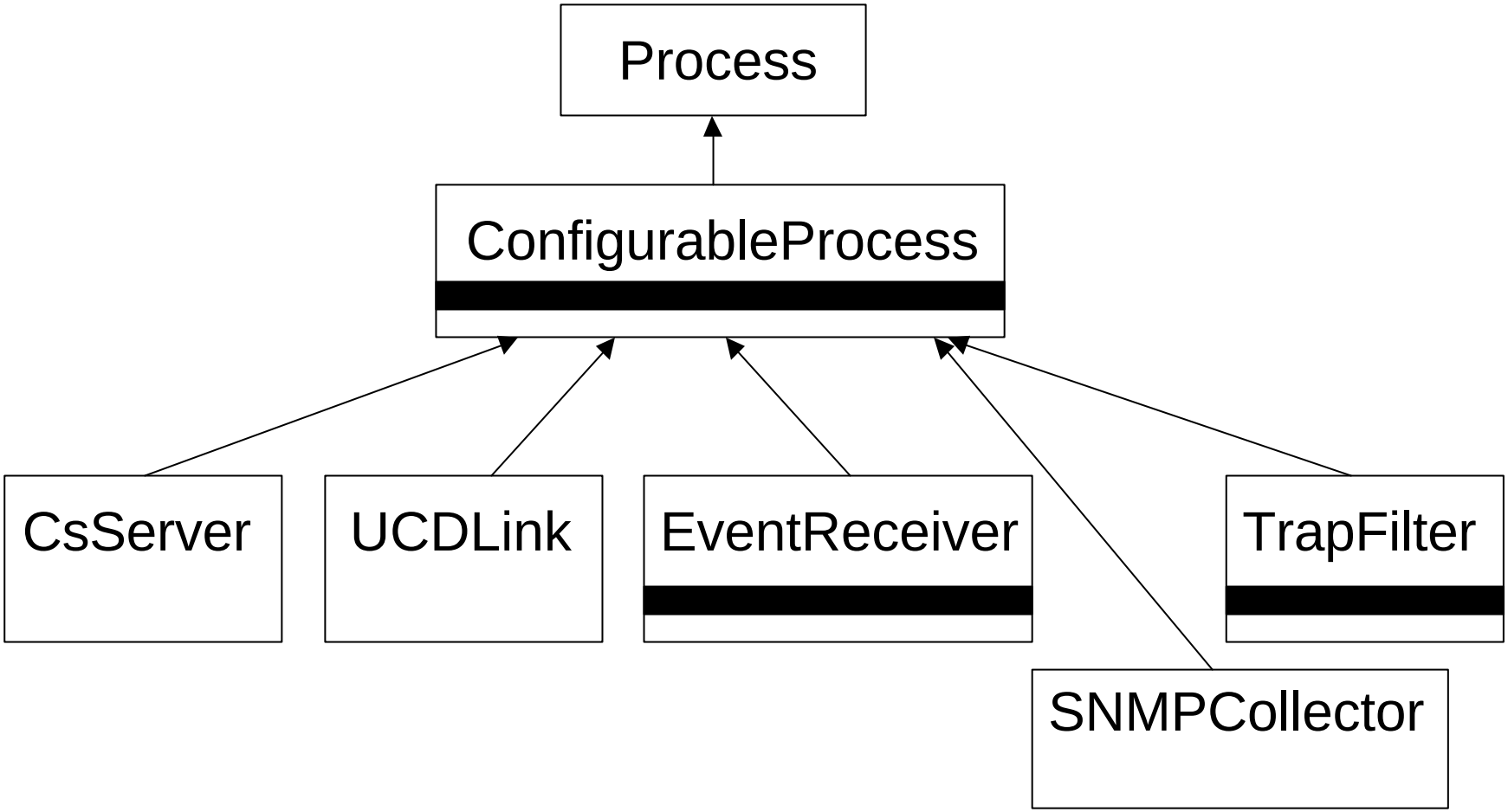
Subclasses of Process all contain repeated Code

Template Method Pattern




ConfigurableProcess
ReadConfigFiles()
hookMethod()
EventReceiver
hookMethod()
CsServer
if (statusOfProcess == Running)
{
throw ConfigFile::ProcessRunning();
}
if ((statusOfProcess == Idle) ||
(statusOfProcess == Paused))
{
hookMethod();
statusOfProcess = Configured;
}
else {
string err("Config File already read!!");
throw ConfigFile::ConfigurationFileError(err.c_str()
}
if (pAffacade->ReadConfigs() != true)
{
pAffacade->ClearConfigs();
throw ConfigFile::ConfigurationFileError();
};

Duplication Resolved



Nested Conditionals

New cases should ideally not require changing existing code

May apply the State / Strategy / NullObject pattern

Use dynamic dispatch

- Define subclasses
- Define abstract method in superclass
- Put every leg into an overriding subclass method

Obstacles to Refactoring

Complexity

- Changing design is hard
- Understanding code is hard

Possibility to introduce errors

- Run tests if possible
- Build tests

Clean first **Then** add new functionality

Cultural Issues

- Producing negative lines of code, what an idea!
  - *"We pay you to add new features, not to improve the code!"*
- If it ain't broke, don't fix it
  - *"We do not have a problem, this is our software!"*

Obstacles to Refactoring

- Performance
  - Refactoring may slow down the execution
  - The secret to write fast software: *Write tunable software first then tune it*
    - Typically only 10% of your system consumes 90% of the resources so just focus on 10 %.
  - Refactorings help to localize the part that need change
  - Refactorings help to concentrate the optimizations

- Development is always under time pressure
  - Refactoring takes time
  - Refactoring better right after a software release

Conclusion: Know-when & Know-how

- **Know-when is as important as know-how**
  - Refactored designs are more complex
  - Use *"code smells"* as symptoms
  - Rule of the thumb: *"Once and Only Once"* (Kent Beck)

    => a thing stated more than once implies refactoring

Further Information

More about code smells and refactoring

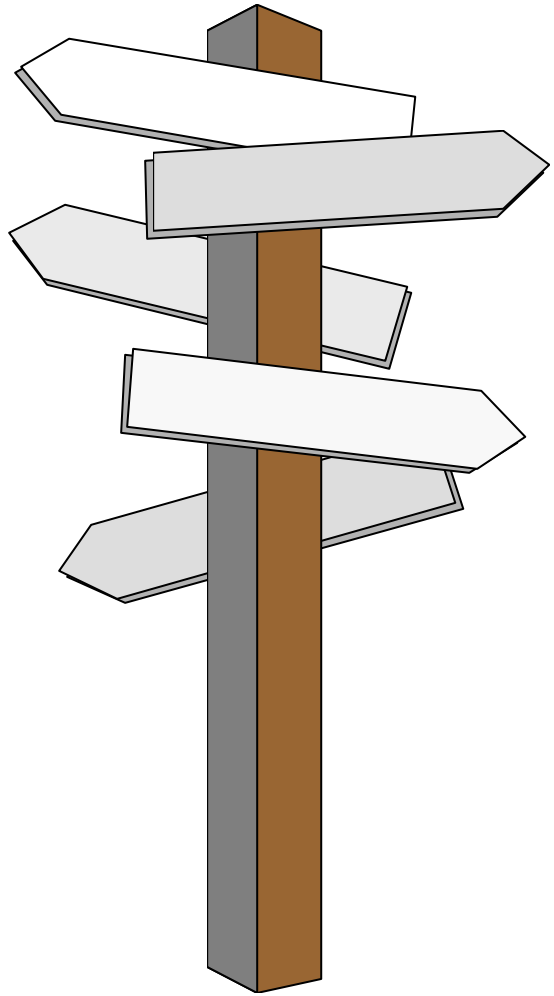
- Book on refactoring [Fowl99a]
  http://www.refactoring.com
- Discussion site on code smells
  http://c2.com/cgi/wiki?CodeSmell

The presented tools

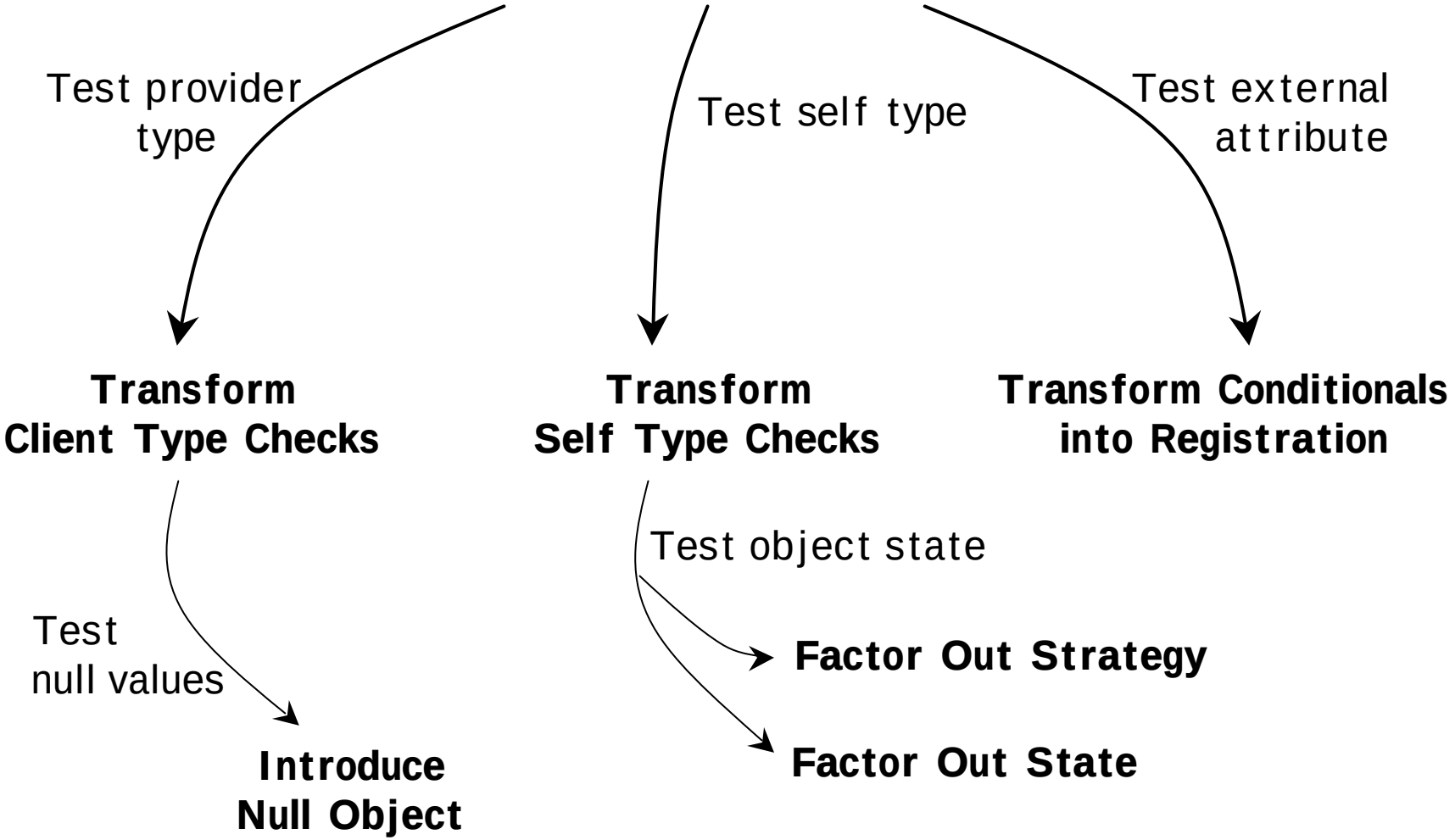
- Refactoring Browser (in VisualWorks Smalltalk)
  http://www.cincom.com/smalltalk
  (also available in several other Smalltalks)
- Java Refactorings in Eclipse
  http://www.eclipse.org
  (also available in other Java environments)

Restructuring

- Most common situations
- Transform Conditionals to Polymorphism
  - Transform Self Type Checks
  - Transform Provider Type Checks



Transform Conditionals to Polymorphism



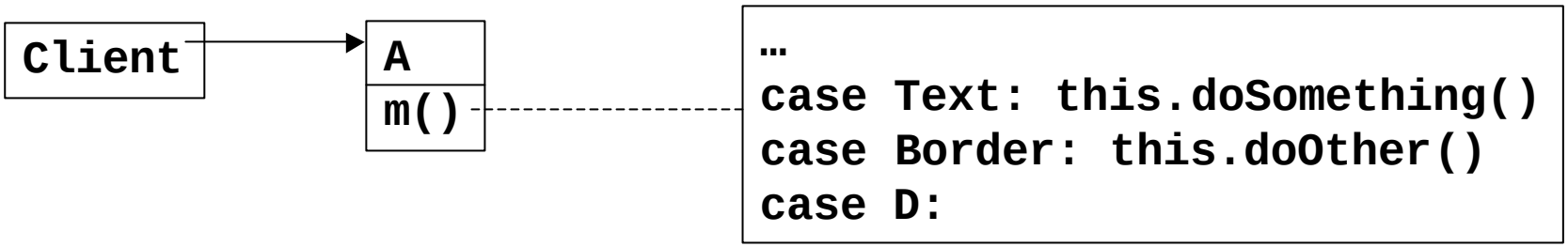
Forces

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

Overview

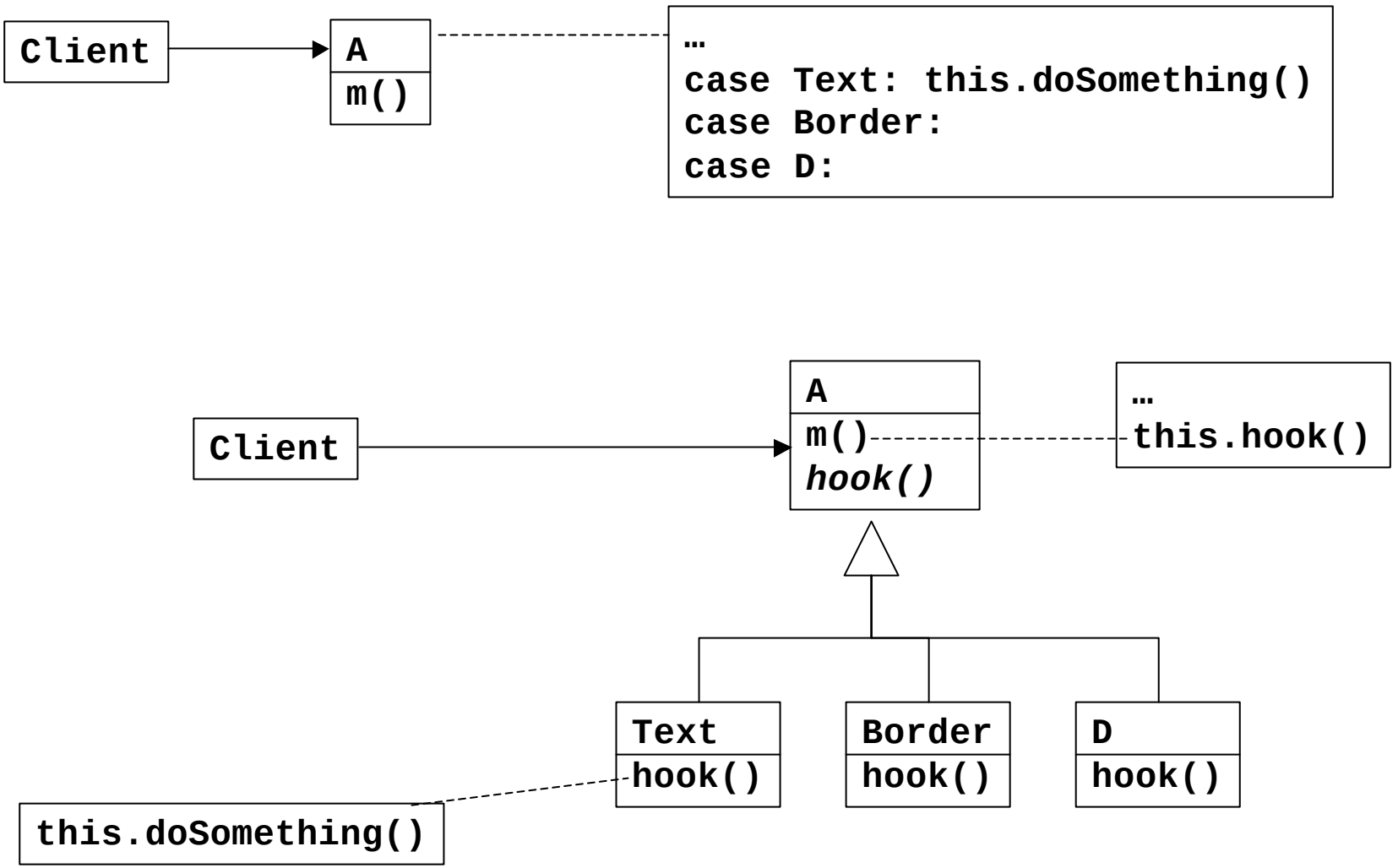
- **Transform Self Type Checks**
  - eliminates conditionals over type information in a provider by introducing new subclasses
- **Transform Client Checks**
  - eliminates conditionals over client type information by introducing new method to each provider classes
- **Factor out State**
  - kind of Self Type Check
- **Factor out Strategy**
  - kind of Self Type Check
- **Introduce Null Object**
  - eliminates null test 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



Detection

- Long methods with complex decision logic
- Look for attribute set in constructors but never changed
- Attributes to model type or finite set constants
  - look for "enums" and attributes with type-related names
- Multiple methods switch on the same attribute
- grep switch ‘find . -name “*.cxx” -print’

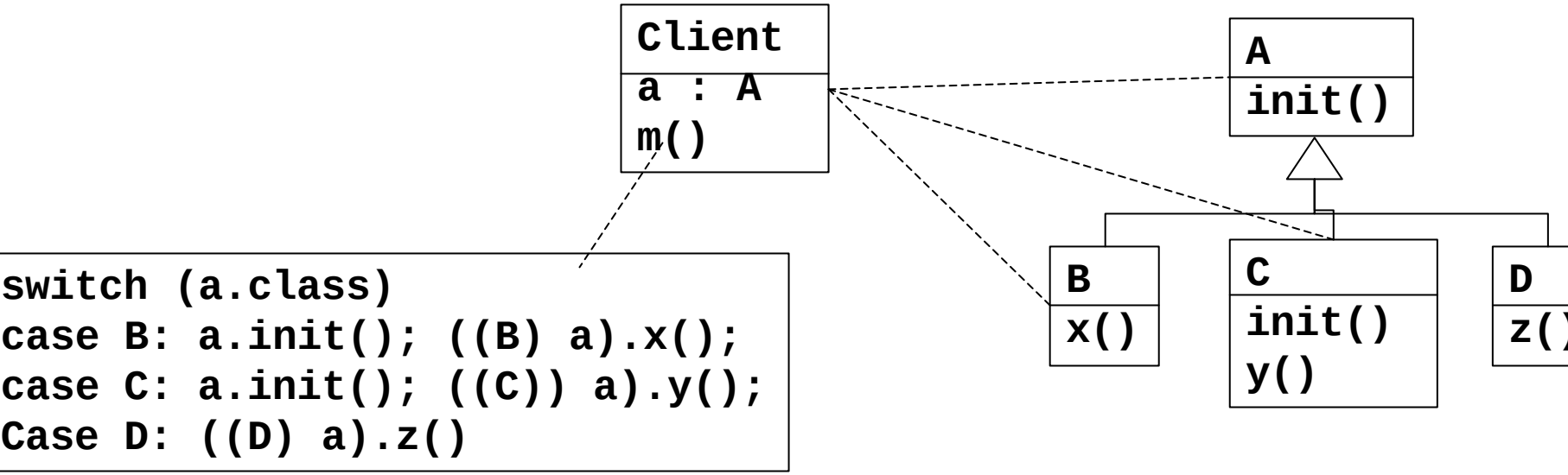
Pros/Cons/Difficulties

- Pros
  - New behavior is 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

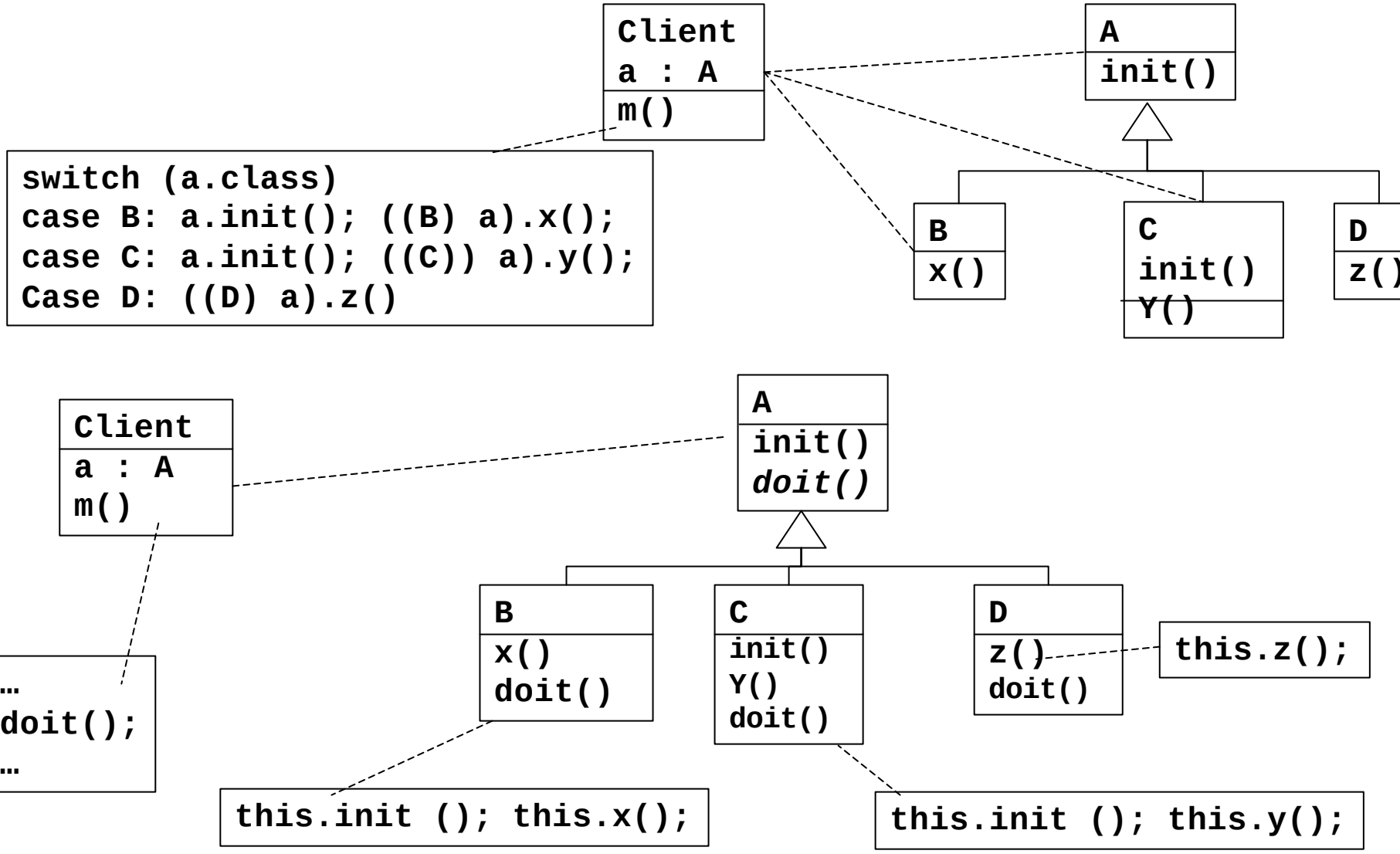
- Difficulties
  - Not always one to one mapping between cases and subclasses
  - Clients may be changed to create instance of the right subclass
    - ...buy creational patterns might help

Transform Client Type Checks



- Clients explicit type checks
- Adding a new provider requires to change all the clients
- Clients are defining logic about providers

Transformation



Detection

- Changing clients of method when new case added
- Attribute representing a type
  - 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 grouped per method but per class
- Difficulties
  - Refactor the clients (Deprecate Obsolete Interfaces)
  - Instance creation should not be a problem

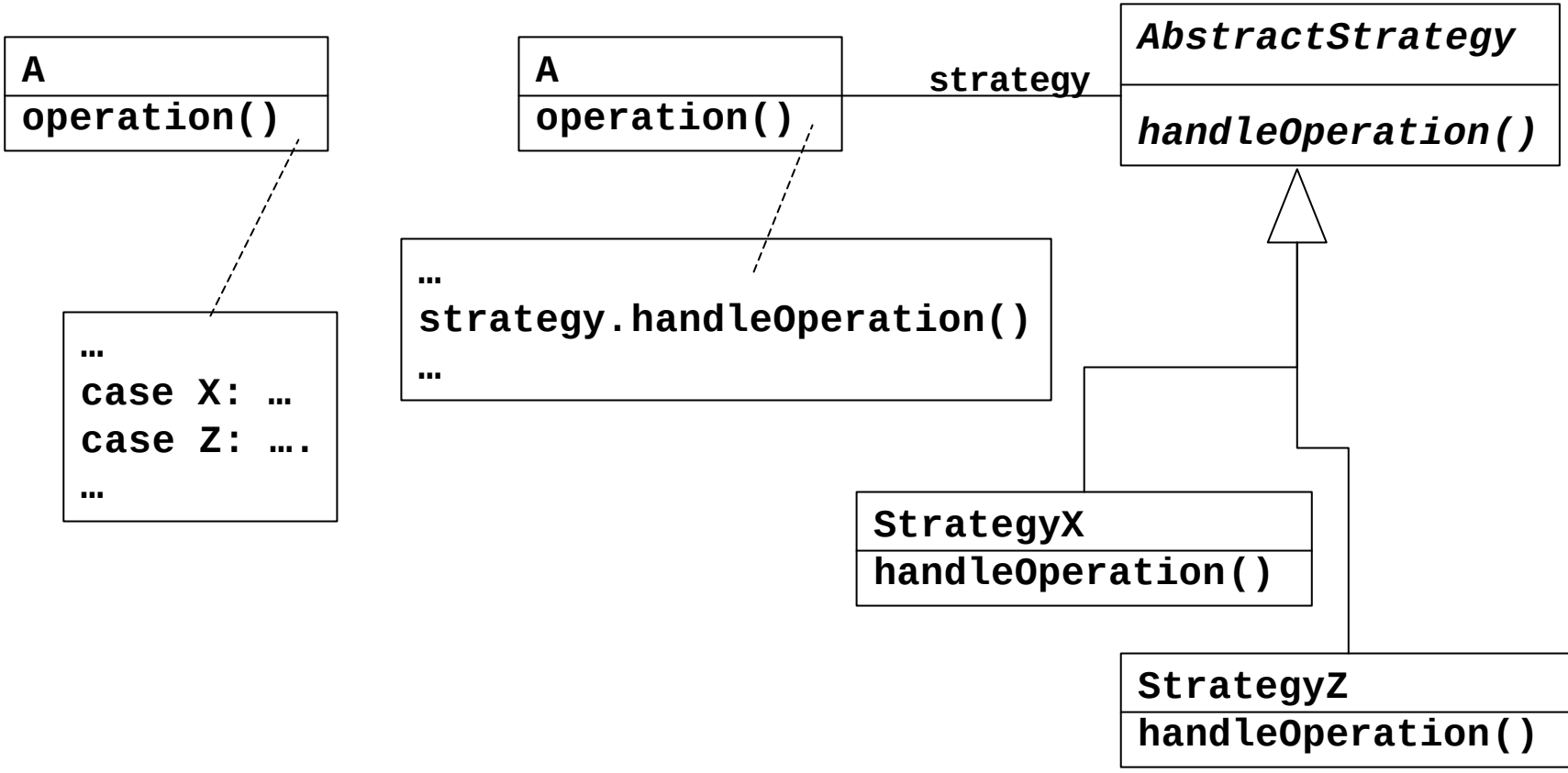
When the Legacy Solution is the Solution

- 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

Factor Out Strategy

- Problem: How do you make a class whose behavior depends on testing certain value more extensible
- Apply State Pattern
  - Encapsulate the behavior and delegate using a polymorphic call

Transformation



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...)