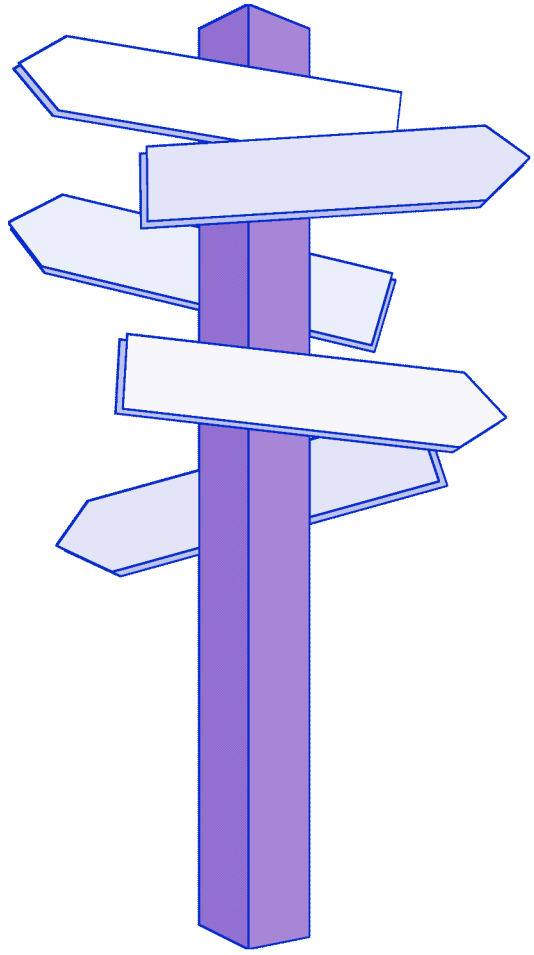
Most Common Reengineering Patterns

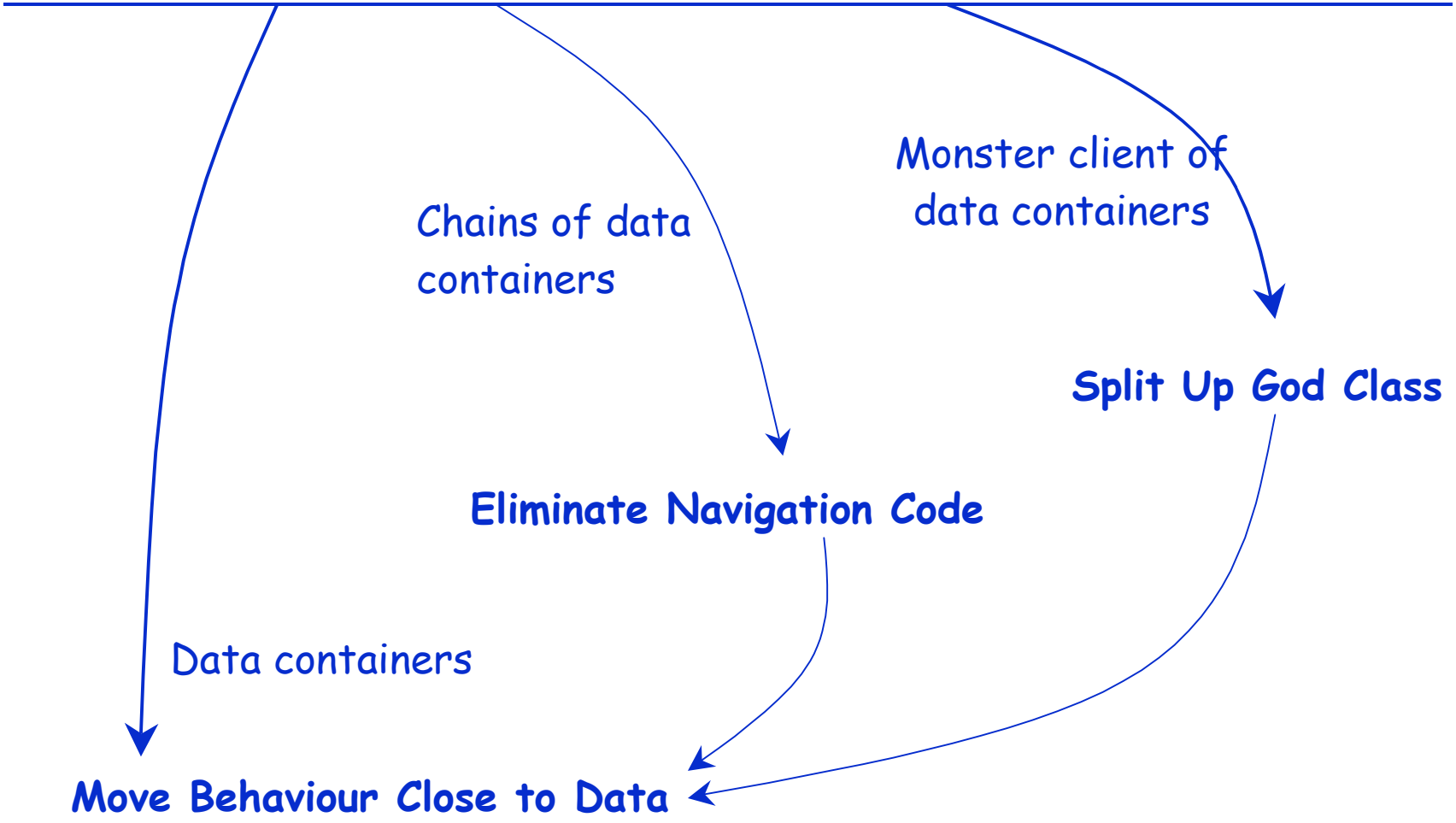
- Most common situations

- Redistribute Responsibilities
  - Eliminate Navigation Code
  - Move Behaviour Close to Data
  - Split up God Class

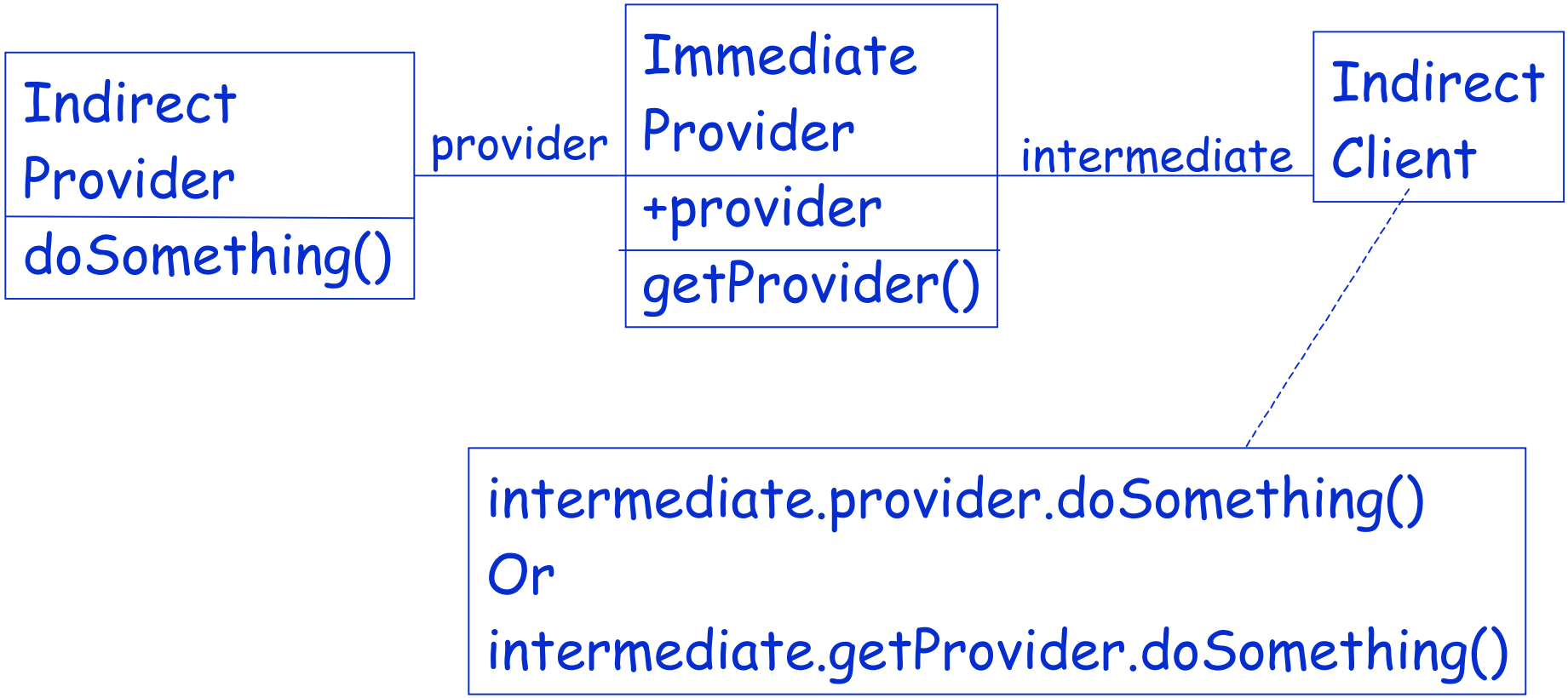
- Transform Conditionals to Polymorphism
  - Transform Self Type Checks
  - Transform Provider Type Checks
  - Transform Conditionals in Registration



Redistribute Responsibilities



The Core of the Problems

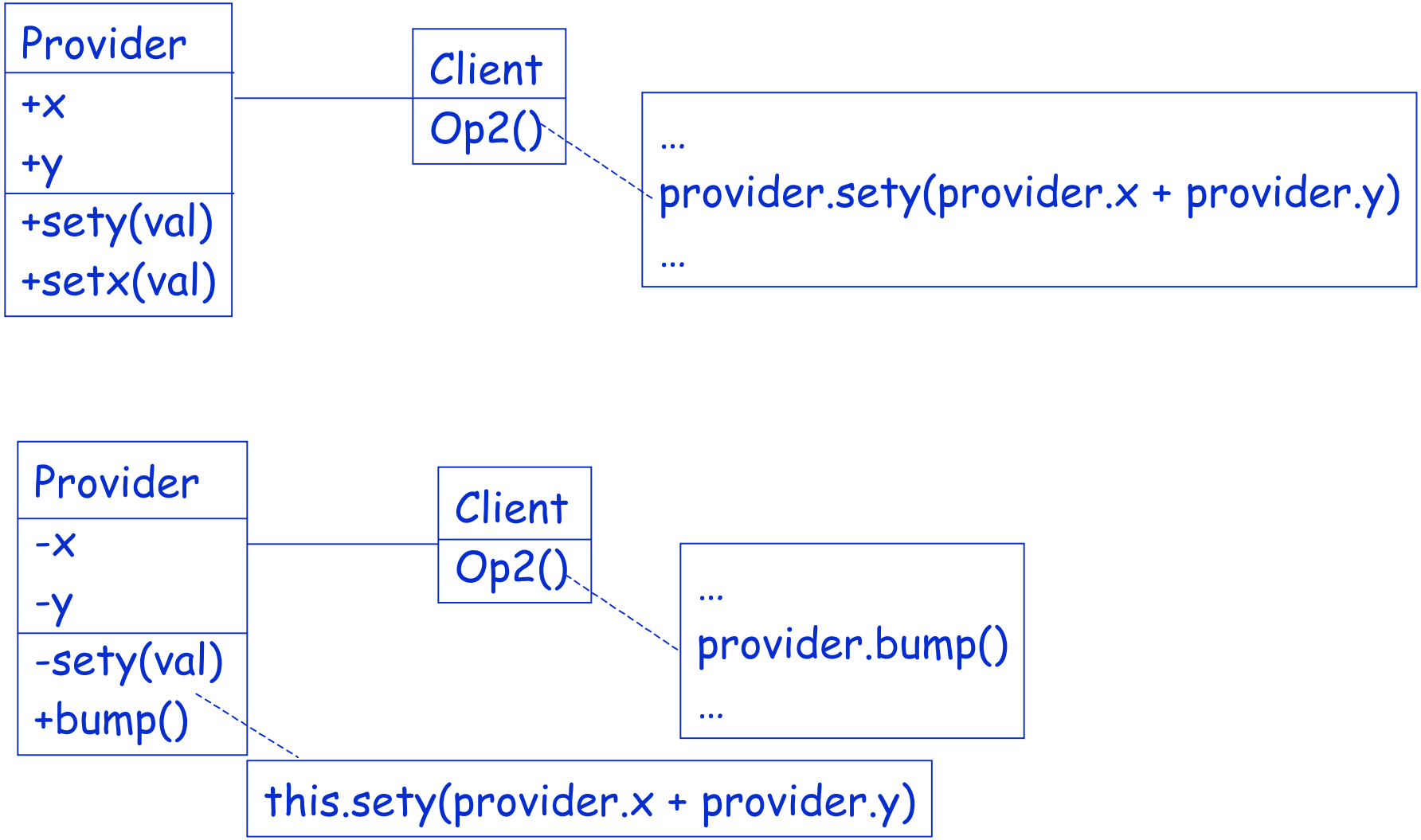


Law of Demeter

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

Transformation...



Detection

- Look for data containers
- Duplicated client code
- Methods using sequence of accessors

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

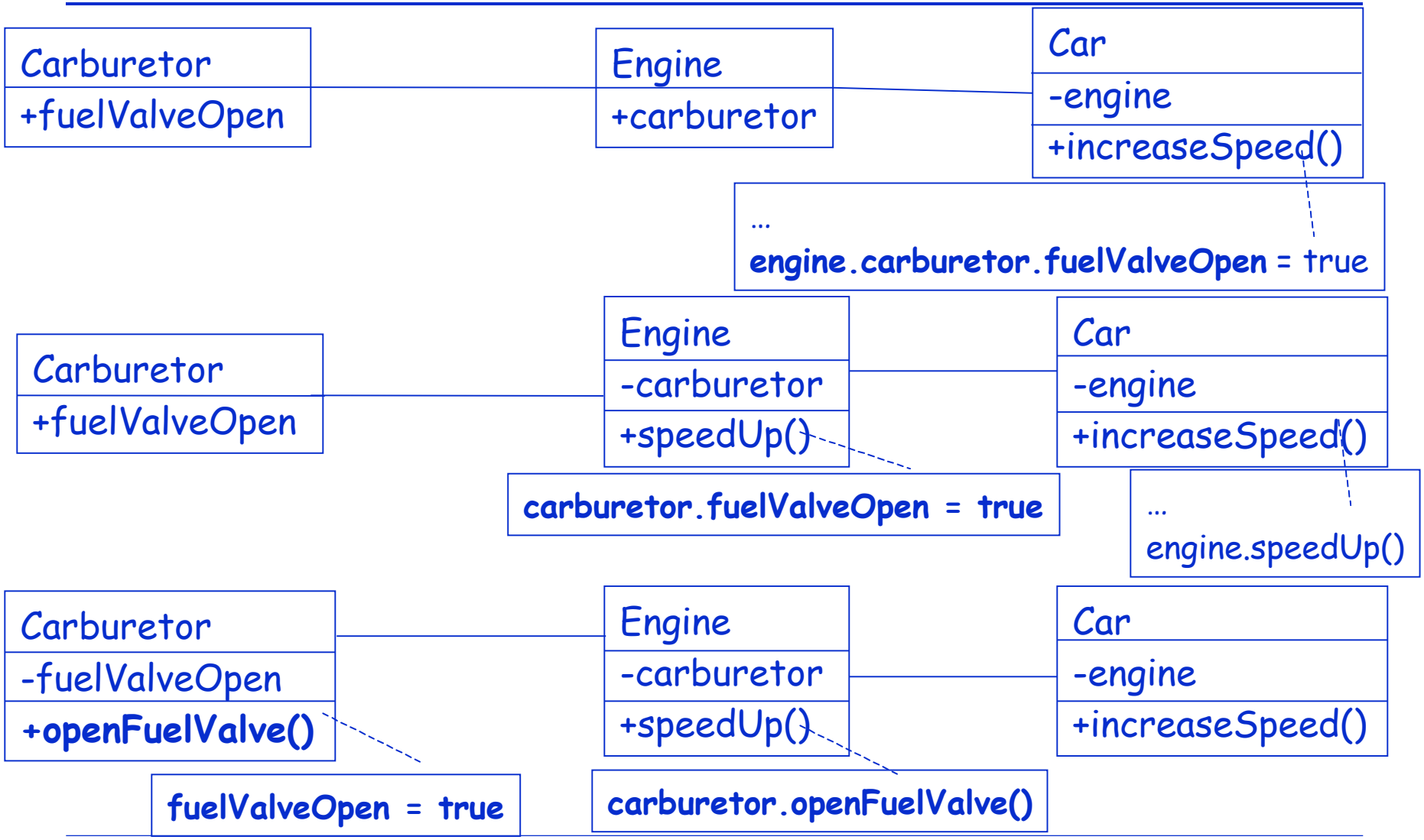
When Legacy Solution is not a Problem

- 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

Eliminate Navigation Code

- a.k.a Law of Demeter
- 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)

Transformation



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

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){…
```

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

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

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

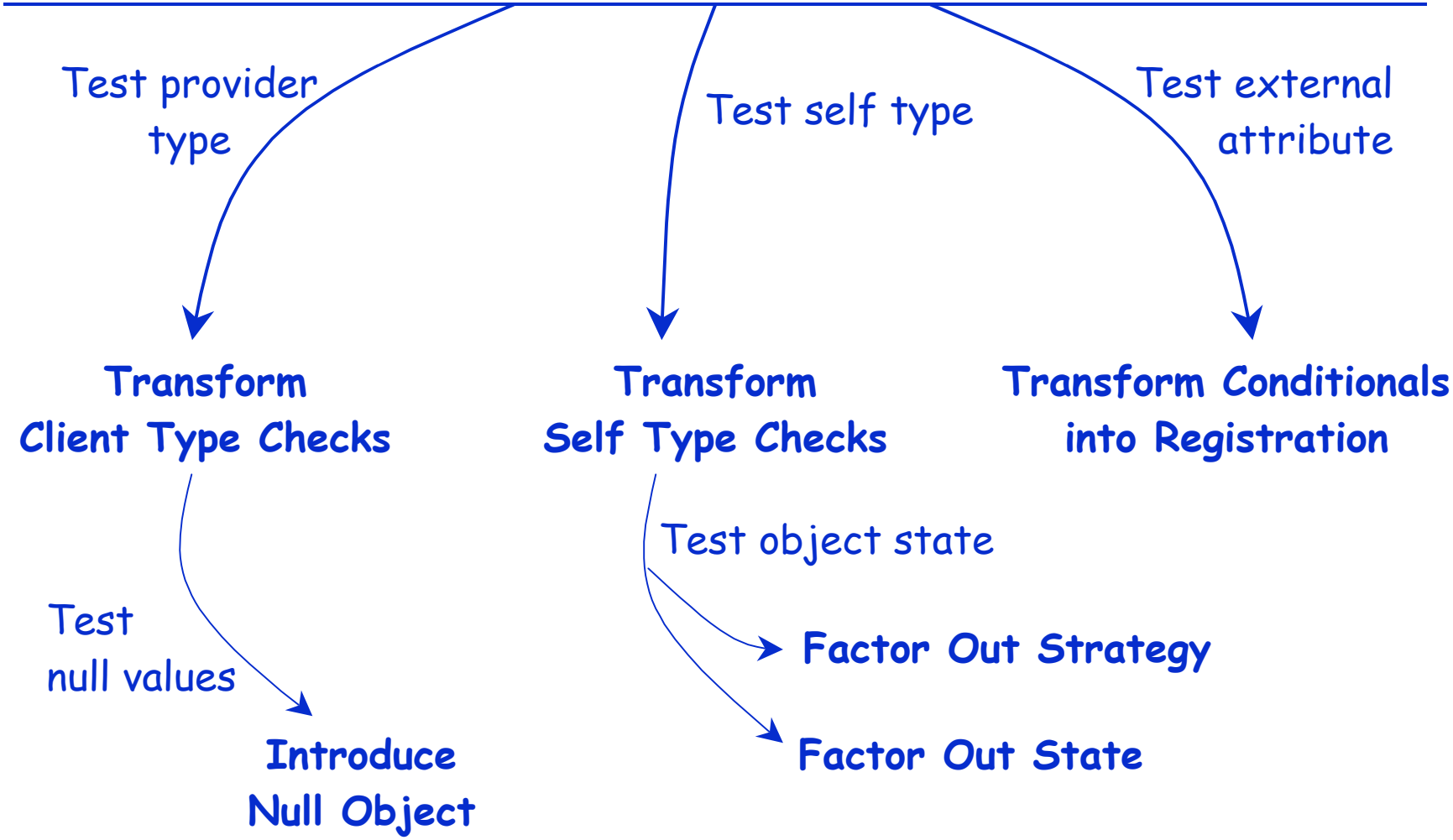
Transformation

- 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 do't touch it!
- Wrap it with different OO views
  - but a God Class usually defines the control flow of the application

Transform Conditionals to Polymorphism



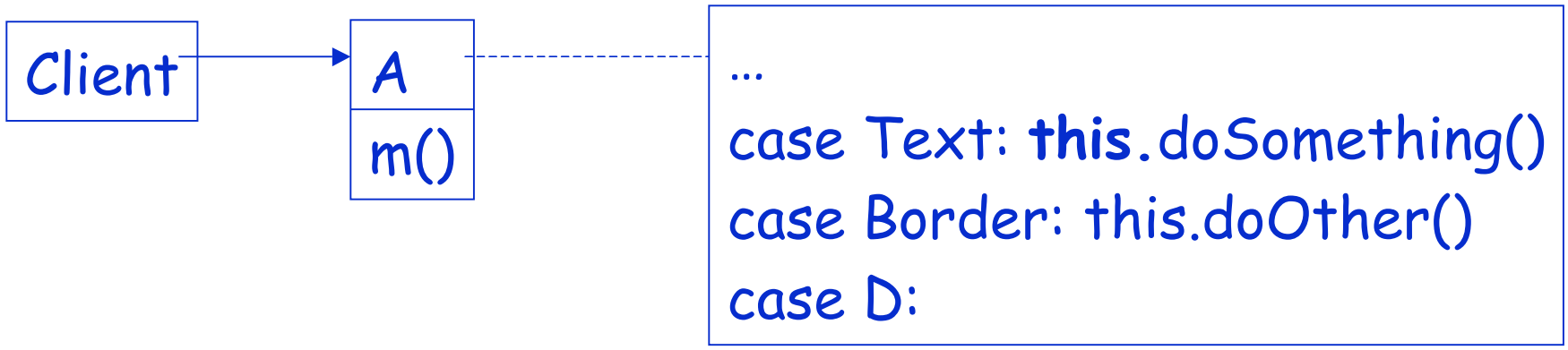
Forces

- Requirements change, so new classes and new method will have to be introduced
- Adding new classes may clutter the namespace
- 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

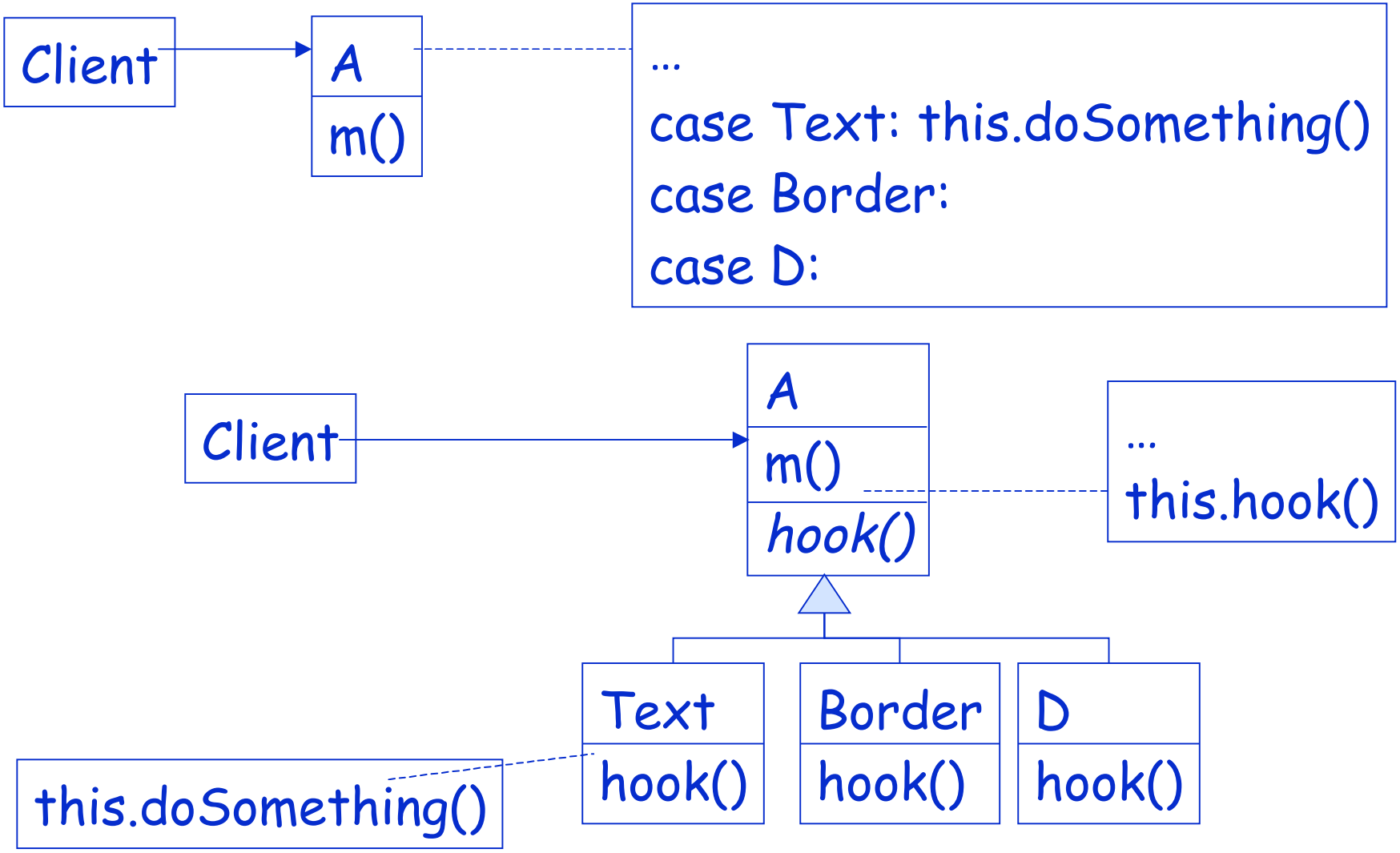
- **Transform Self Type Checks** eliminates conditionals over type information in a provider by introducing new subclasses
- **Transform Client Checks** eliminates conditionals over client type information by introducing new method to each provider classes
- **Factor out State** (kind of Self Type Check)
- **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



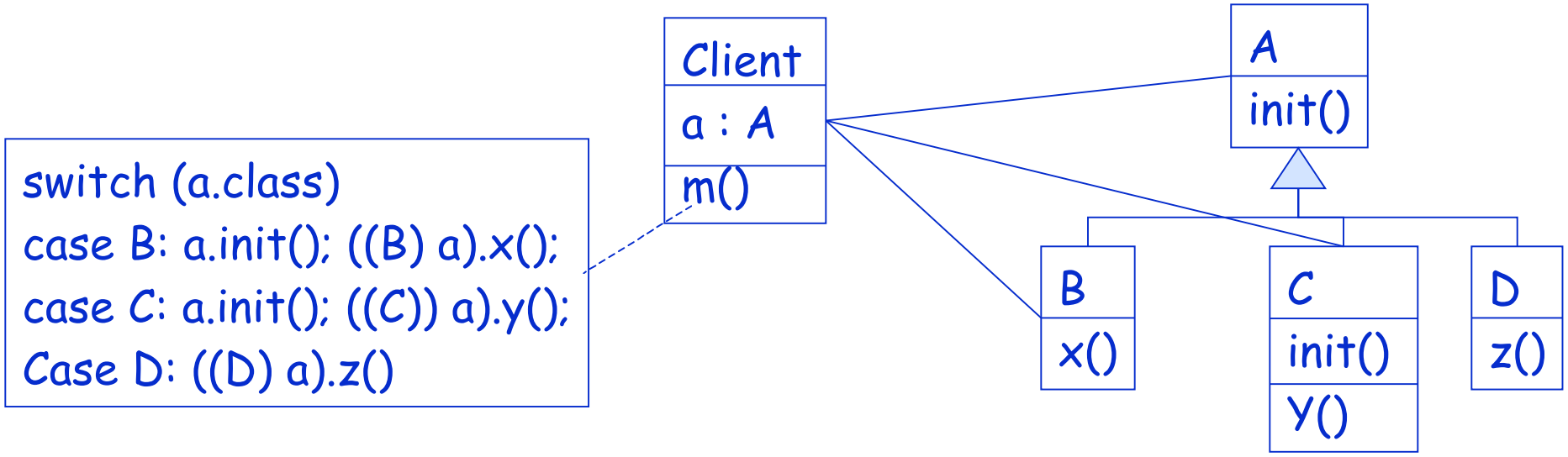
Detection

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

Pros/Cons/Difficulties

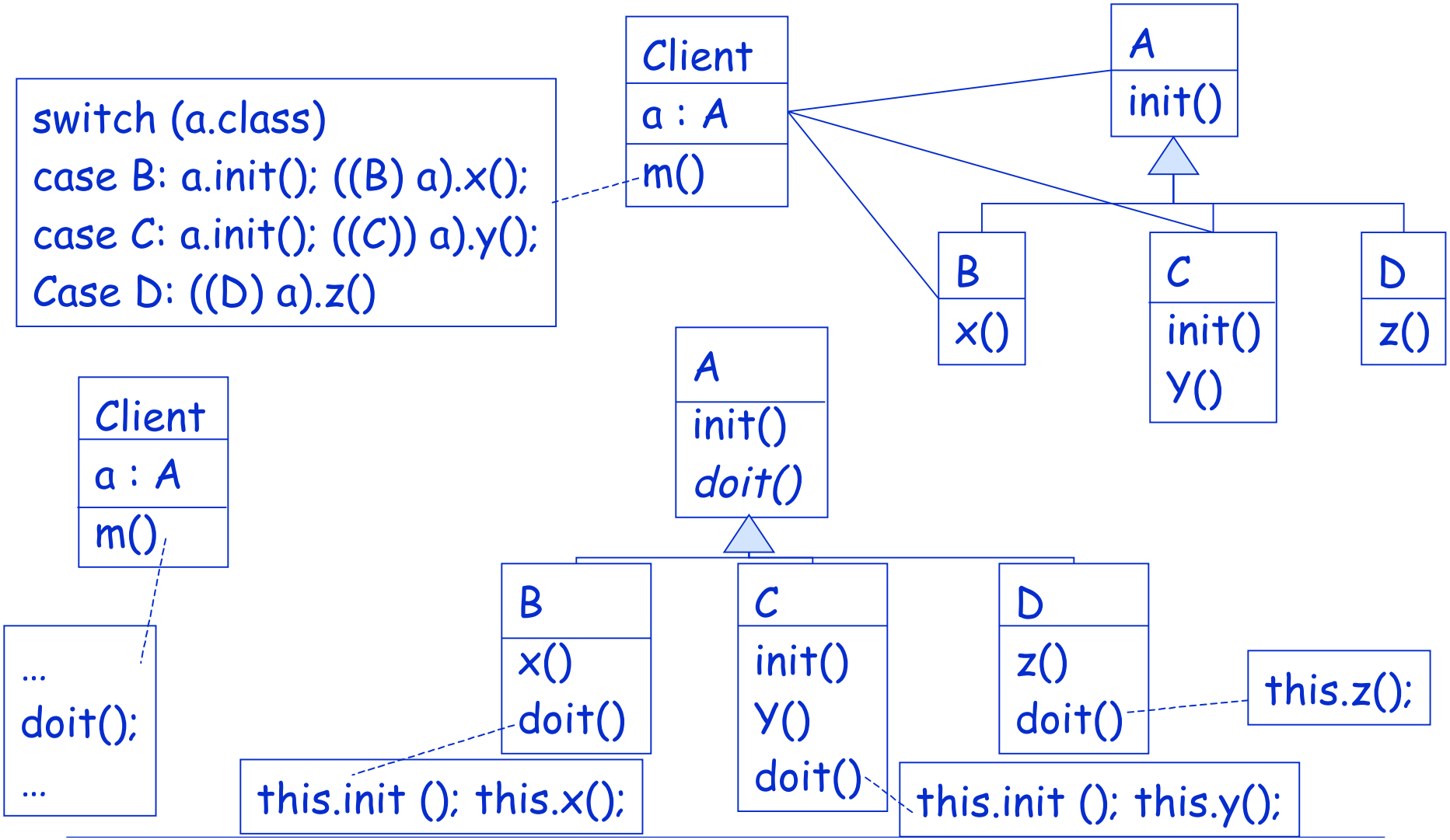
- Pros
  - New behavior are easy to add and to understand: a new class
  - No need to change different method to add a behavior
  - All behaviors share a common interface
- Cons
  - Behavior are dispersed into multiple but related abstractions
  - More classes
- Difficulties
  - Not always one to one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

Transform Client Type Checks



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

Transformation



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

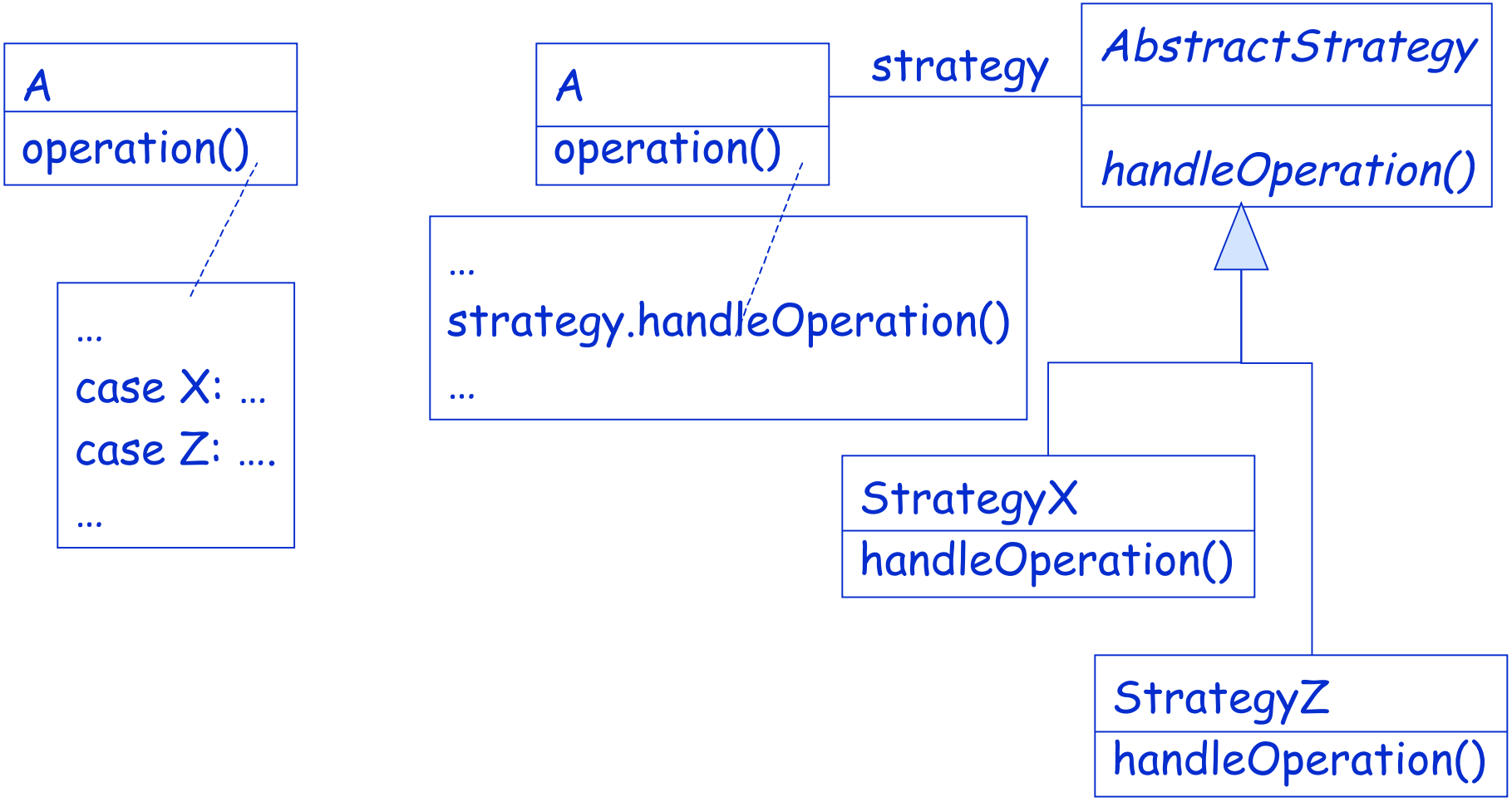
When the Legacy Solution is the Solution

- 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)

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…)

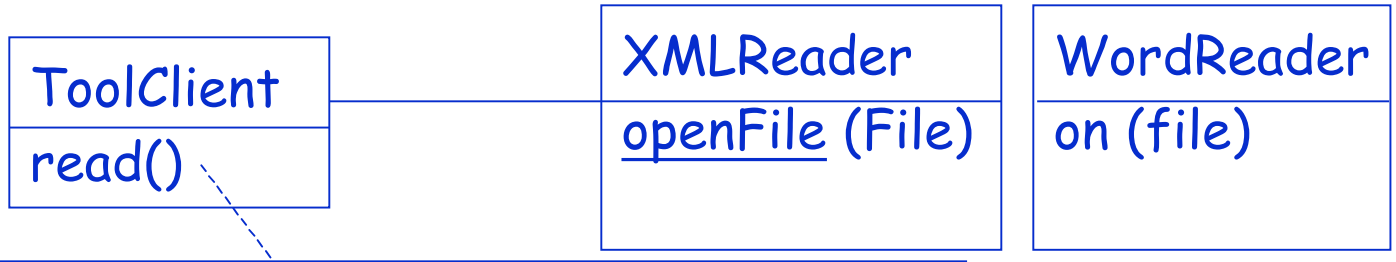
Transform Conditional into Registration

- Problem: How do you reduce the coupling between *tools* providing services and *clients* so that addition/removal of tools *does not change client* code?
- Answer: Introduce a registration mechanism
  - Tools register/unregister
  - Clients query them via the registration repository

Detection

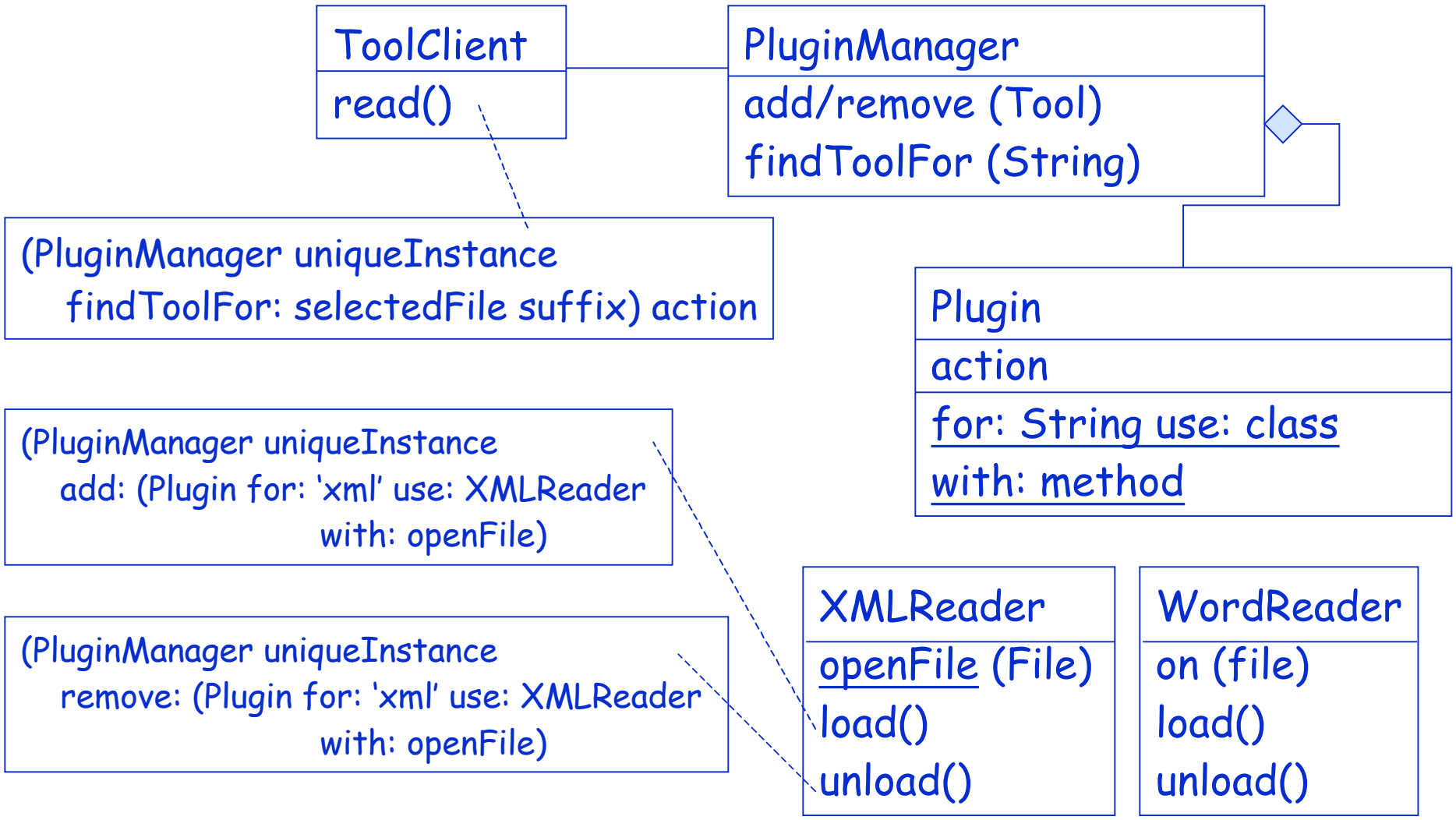
- Long method in clients checking which tools to invoke based
- Removing or adding a tool force to change client code
- Difficulty to have run-time tool loading/unloading

Transformation (i)




```
suffix := selectedFile suffix = 'xml'.
suffix =  'xml'
          ifTrue: [ XMLReader openFile: selectedFile.
                    ^ self]
suffix = 'doc'
          ifTrue: [WordReader on: selectedFile.
                    ^ self].
...
```

Transformation (ii)



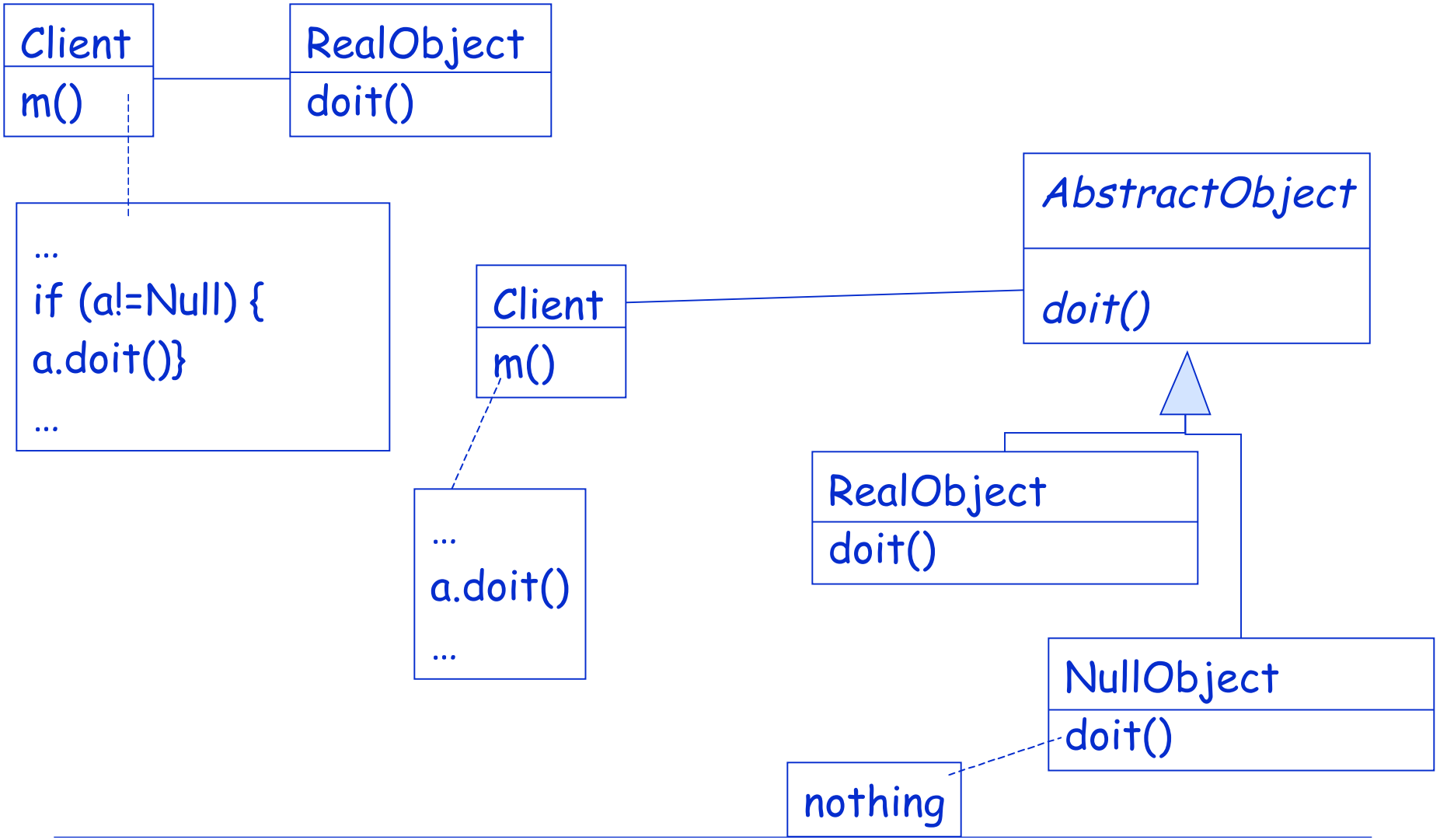
Pros/Cons/Difficulties

- Pros
  - New tools can be added without impacting clients
  - Clients no longer are responsible of the
  - 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 this knowledge is dynamic so more effort for user interface consistency (i.e., consistent menu ordering) is necessary

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

Transformation



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

Conclusion

- Most common lacks of OO use
- Late binding is powerful and flexible
- Long case statements are more costly than virtual calls