Patterns in Software Engineering

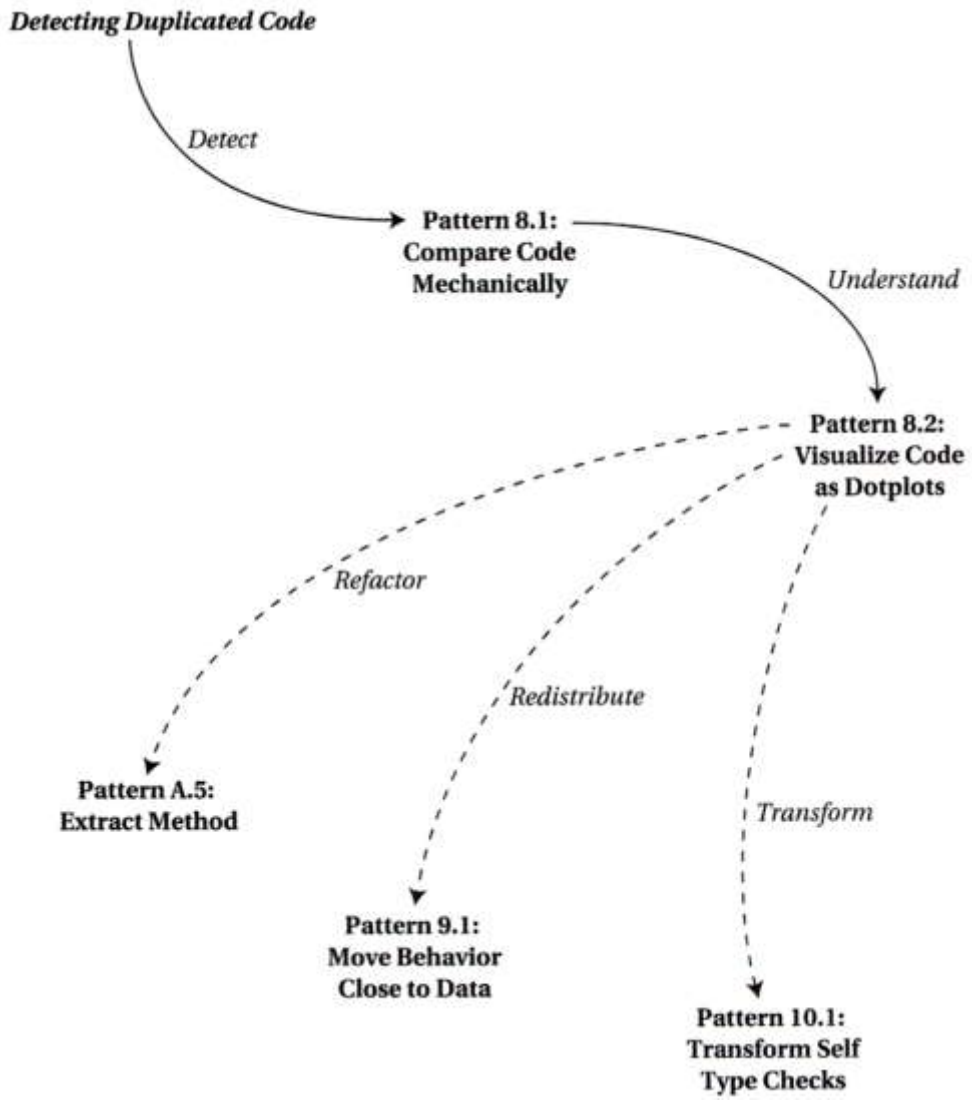
Lecturer: Raman Ramsin

Lecture 14

Reengineering Patterns

Part 2

Reengineering Patterns: Detecting Duplicated Code



Detecting Duplicated Code: Compare Code Mechanically

- **Problem:** How do you discover which parts of an application code have been duplicated?

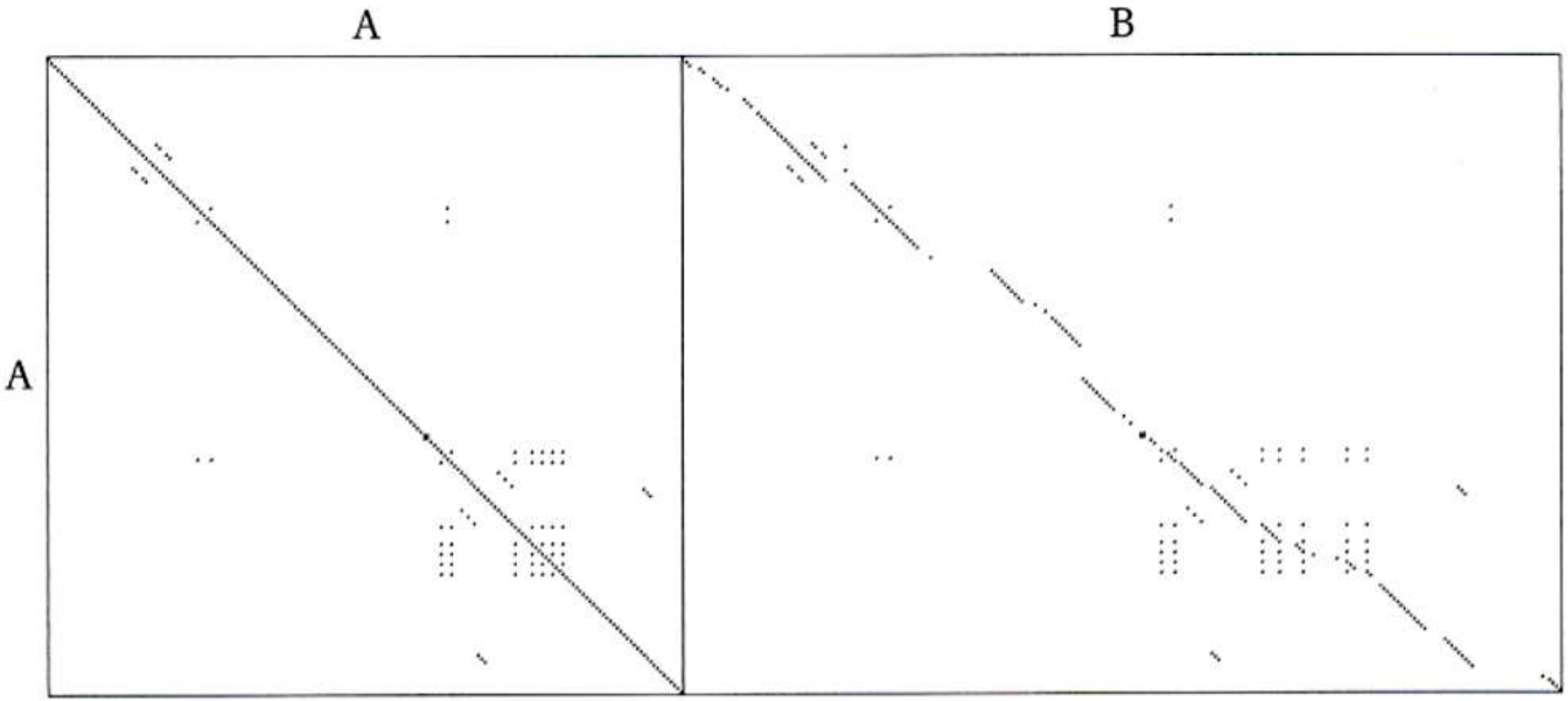
- **Solution:** Textually compare each line of the software source code with all the other lines of code.

Detecting Duplicated Code: Visualize Code as Dotplots

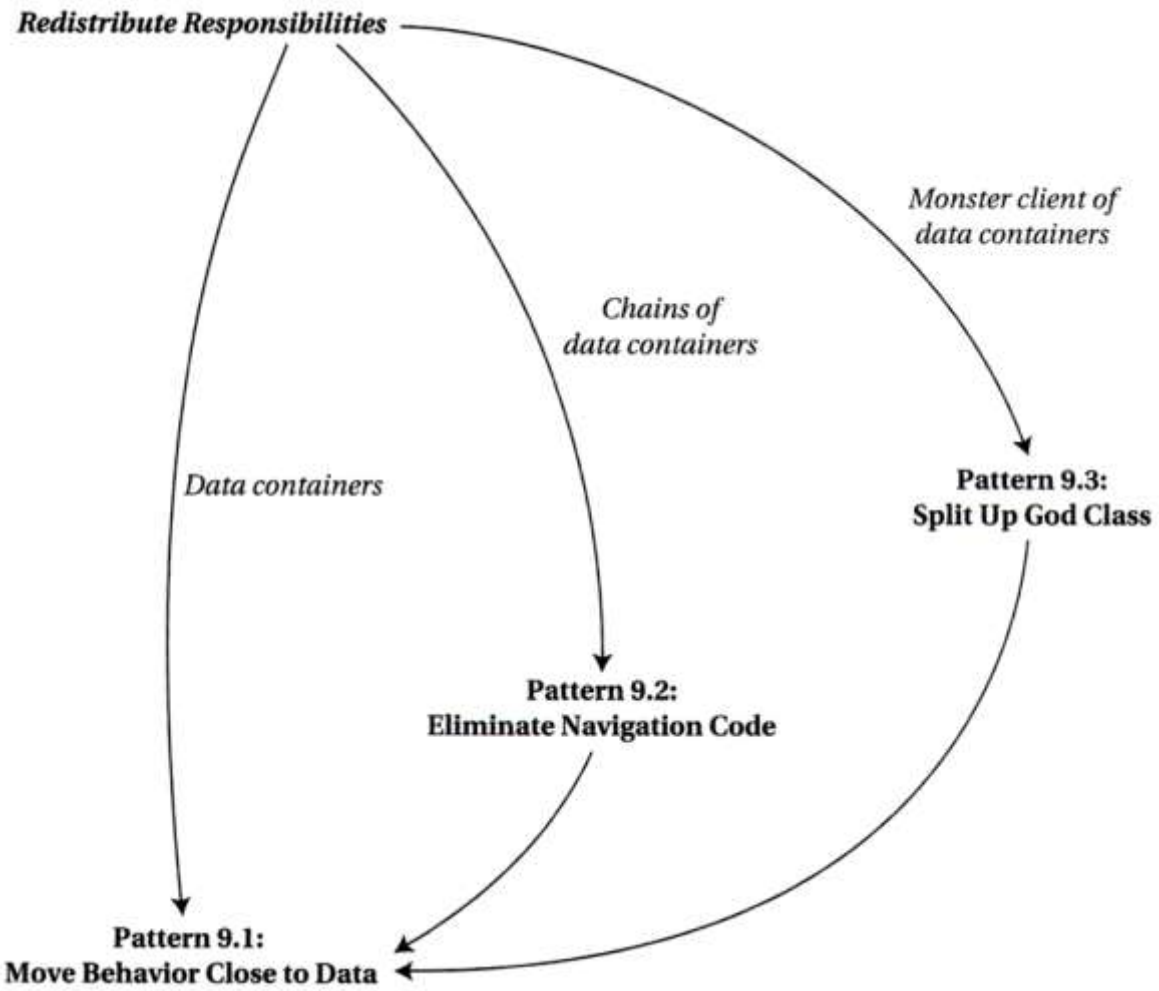
- **Problem:** How can you gain insight into the scope and nature of code duplication in a software system?

- **Solution:** Visualize the code as a matrix in which the two axes represent two source code files (possibly the same file):
  - dots in the matrix occur where source code lines are duplicated.

Visualize Code as Dotplots



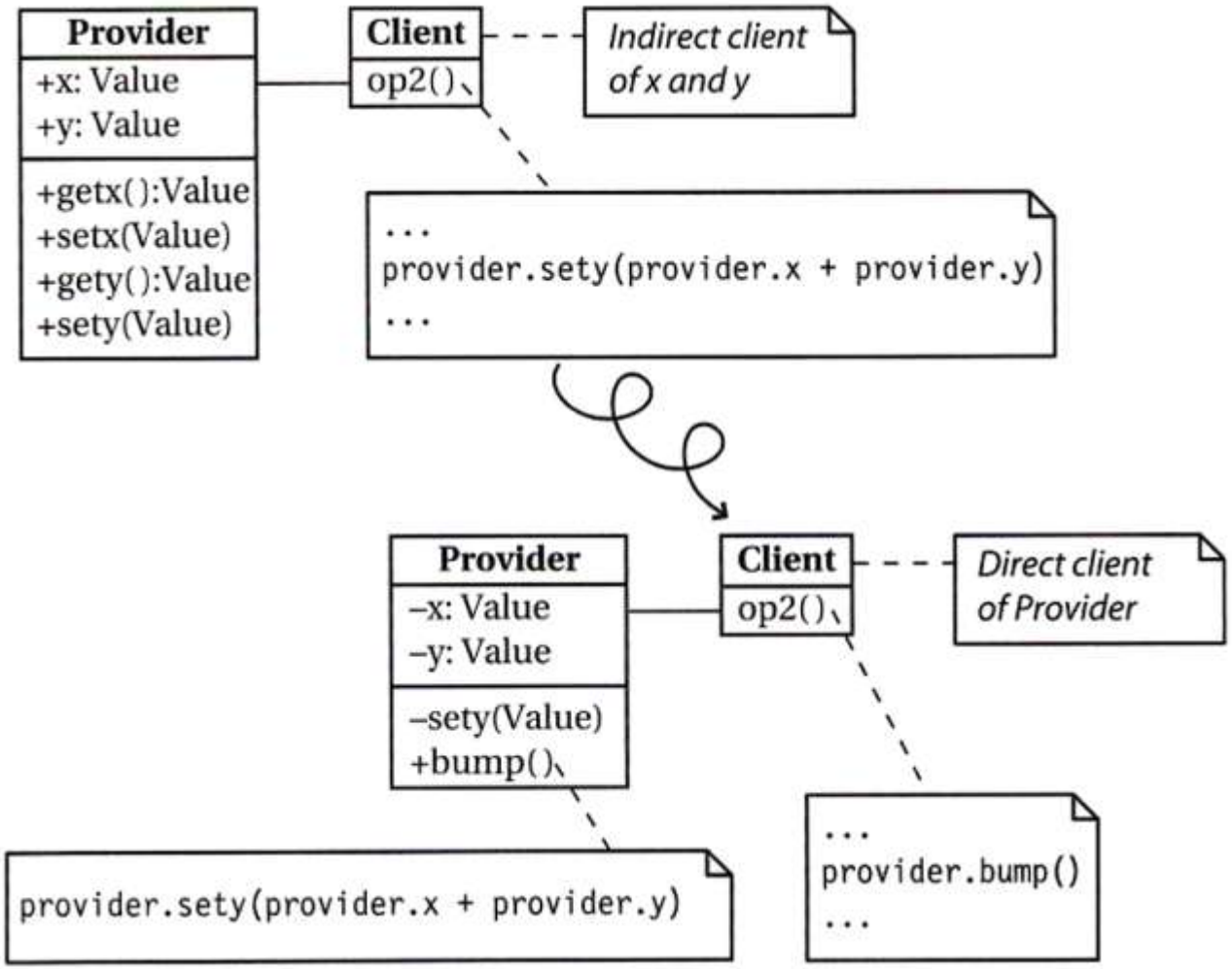
Reengineering Patterns: Redistribute Responsibilities



Redistribute Responsibilities: Move Behavior Close to Data

- **Problem:** How do you transform a class from being a mere data container into a real service provider?
- **Solution:** Move behavior defined by indirect clients to the container of the data on which it operates.

Move Behavior Close to Data

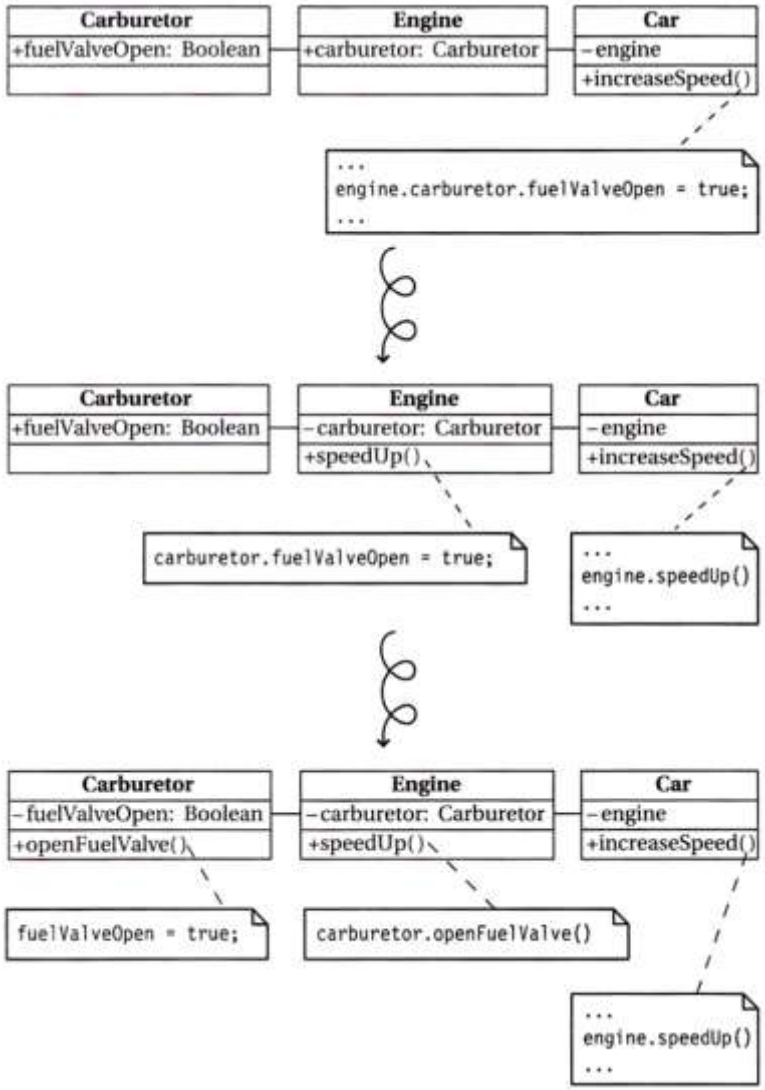


Redistribute Responsibilities: Eliminate Navigation Code

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

- **Solution:** Iteratively move behavior defined by an indirect client to the container of the data on which it operates.
  - The actual reengineering steps are basically the same as those of Move Behavior Close to Data.
    - However, since the manifestation of the problem is rather different, different detection steps apply.

Eliminate Navigation Code

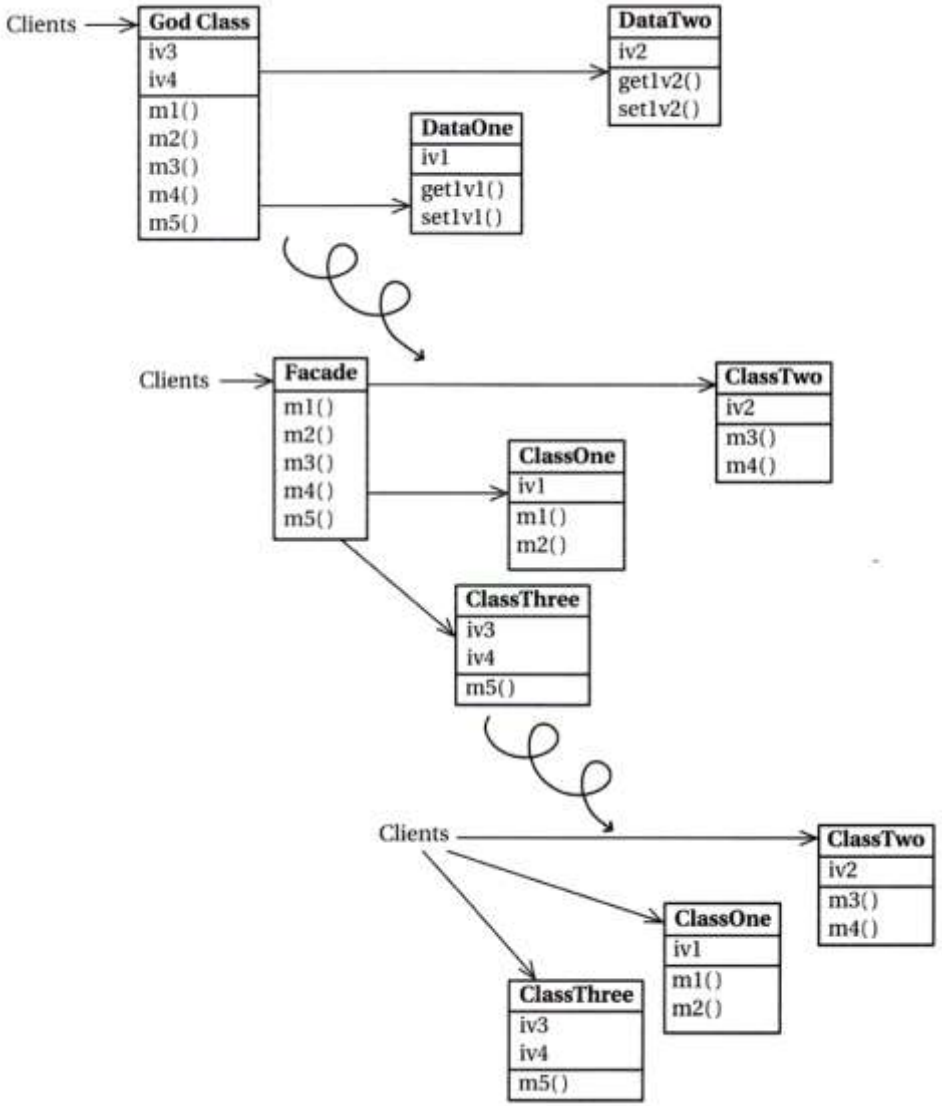


Redistribute Responsibilities: Split Up God Class

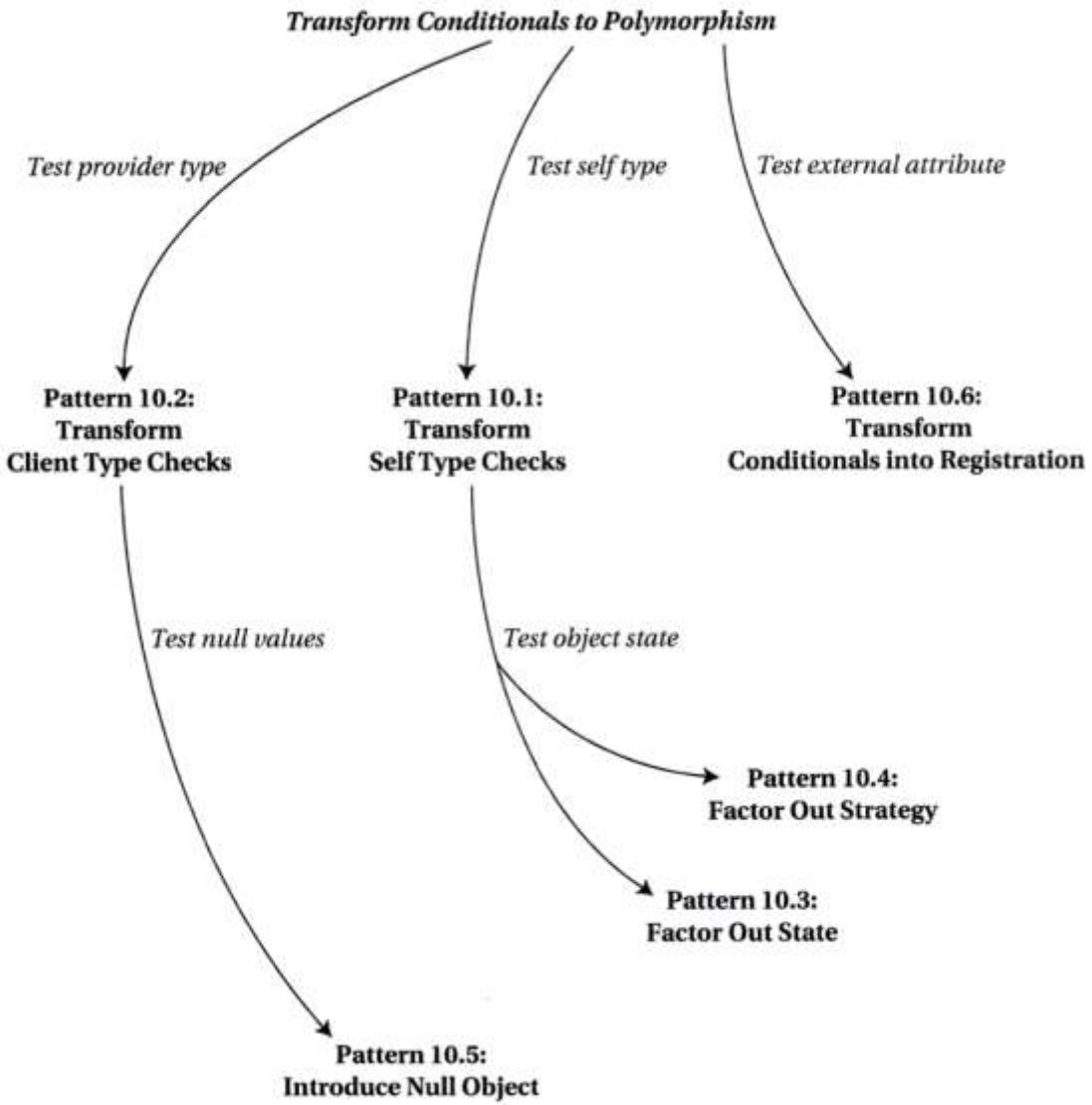
- **Problem:** How do you maintain a class that assumes too many responsibilities?

- **Solution:** Incrementally redistribute the responsibilities of the god class either to its collaborating classes or to new classes that are pulled out of the god class.
  - When there is nothing left of the god class but a facade, remove or deprecate the facade.

Split Up God Class



Reengineering Patterns: Transform Conditionals to Polymorphism

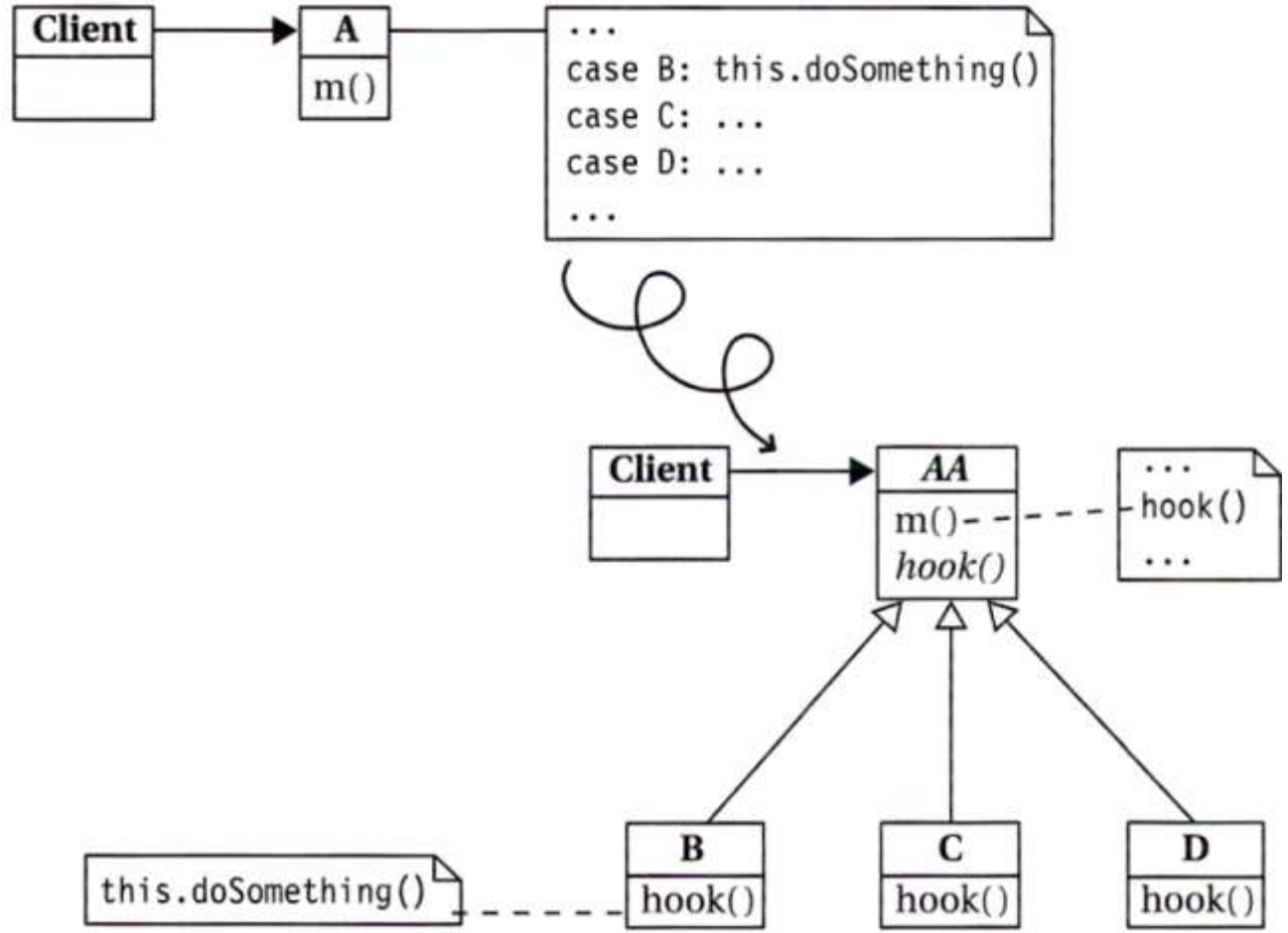


Transform Conditionals to Polymorphism: Transform Self Type Checks

- **Problem:** A class is hard to modify or extend because
  - it bundles multiple possible behaviors in complex conditional statements that test some attribute representing the current "type" of the object.

- **Solution:** Identify the methods with complex conditional branches. In each case:
  - Replace the conditional code with a call to a new hook method.
  - Identify or introduce subclasses corresponding to the cases of the conditional.
  - In each of these subclasses, implement the hook method with the code corresponding to that case in the original case statement.

Transform Self Type Checks

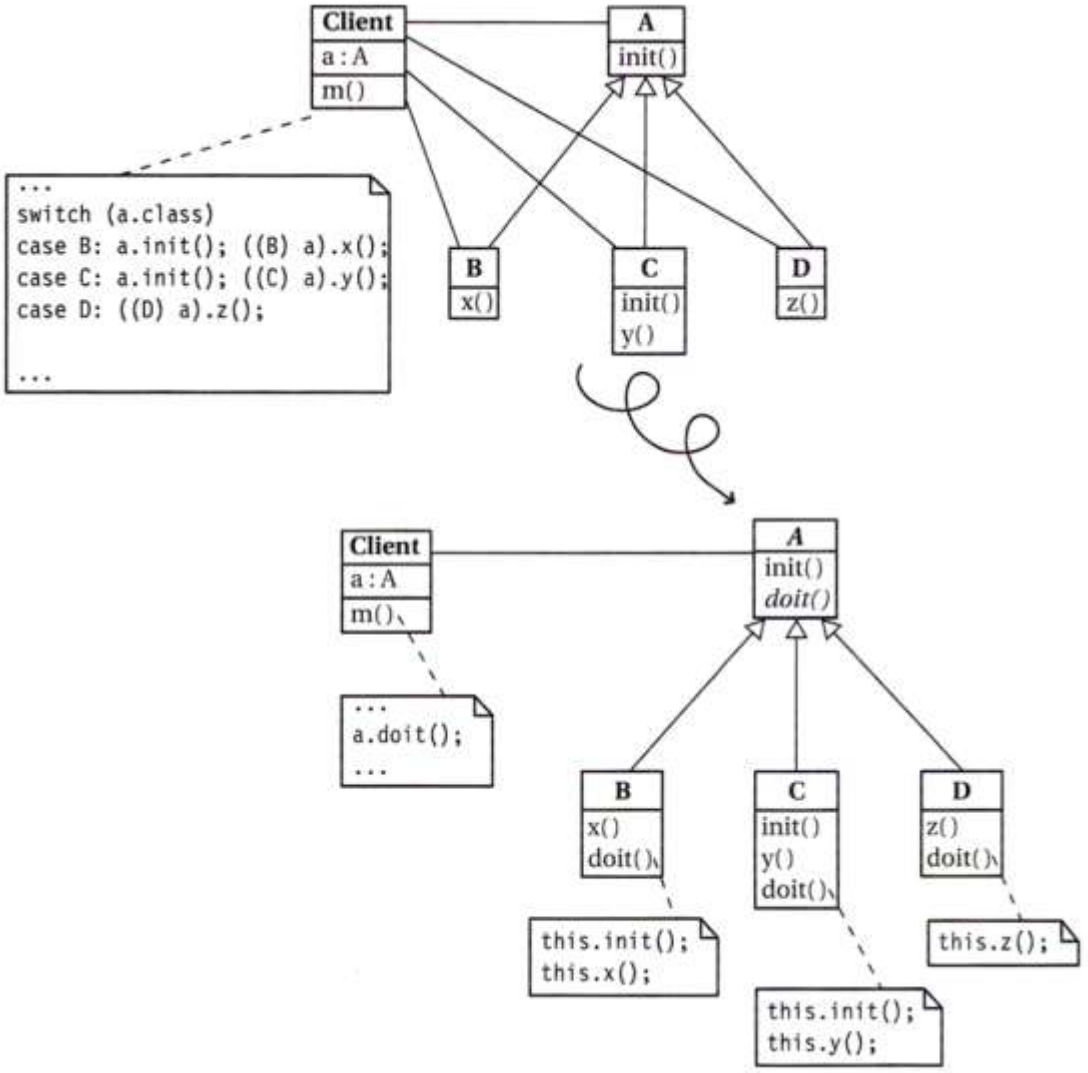


Transform Conditionals to Polymorphism: Transform Client Type Checks

- **Problem:** Reducing the coupling between clients and providers of services, where the clients
  - explicitly test the type of providers, and
  - have the responsibility to compose providers' code.

- **Solution:**
  - Introduce a new method into the provider hierarchy.
  - Implement the new method in each subclass of the provider hierarchy by moving the corresponding case of the client's conditional to that class.
  - Replace the entire conditional in the client by a simple call to the new method.

Transform Client Type Checks

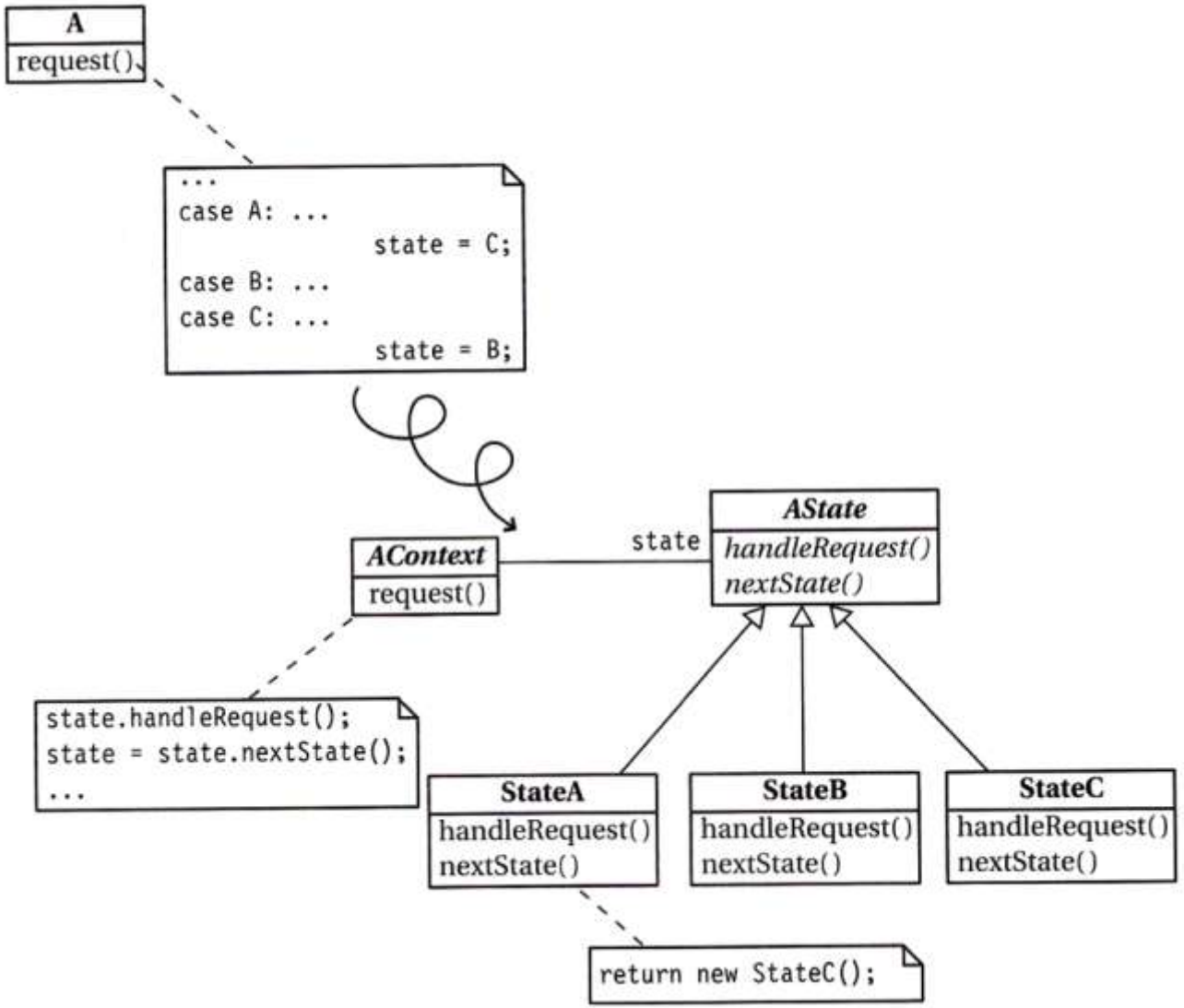


Transform Conditionals to Polymorphism: Factor Out State

- **Problem:** How do you make a class whose behavior depends on a complex evaluation of its current state more *extensible?*

- **Solution:** Apply the *State* pattern, that is,
  - encapsulate the state-dependent behavior into separate objects,
  - delegate calls to these objects, and
  - keep the state of the object consistent by referring to the right instance of these state objects.

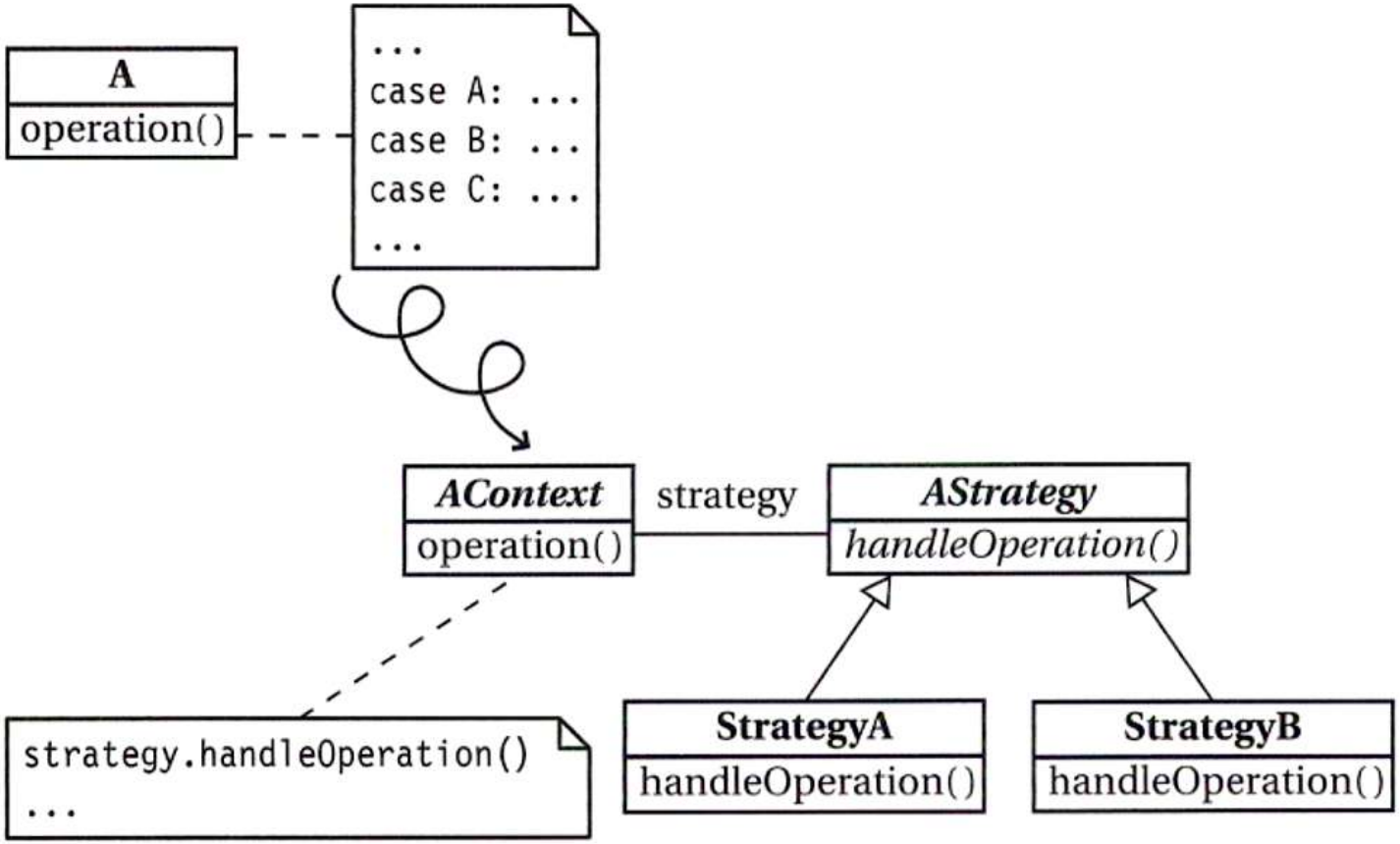
Factor Out State



Transform Conditionals to Polymorphism: Factor Out Strategy

- **Problem:** How do you make a class whose behavior depends on testing the value of some variable more *extensible?*
- **Solution:** Apply the *Strategy* pattern, that is,
  - encapsulate the algorithmic dependent behavior into separate objects with polymorphic interfaces, and
  - delegate calls to these objects.

Factor Out Strategy

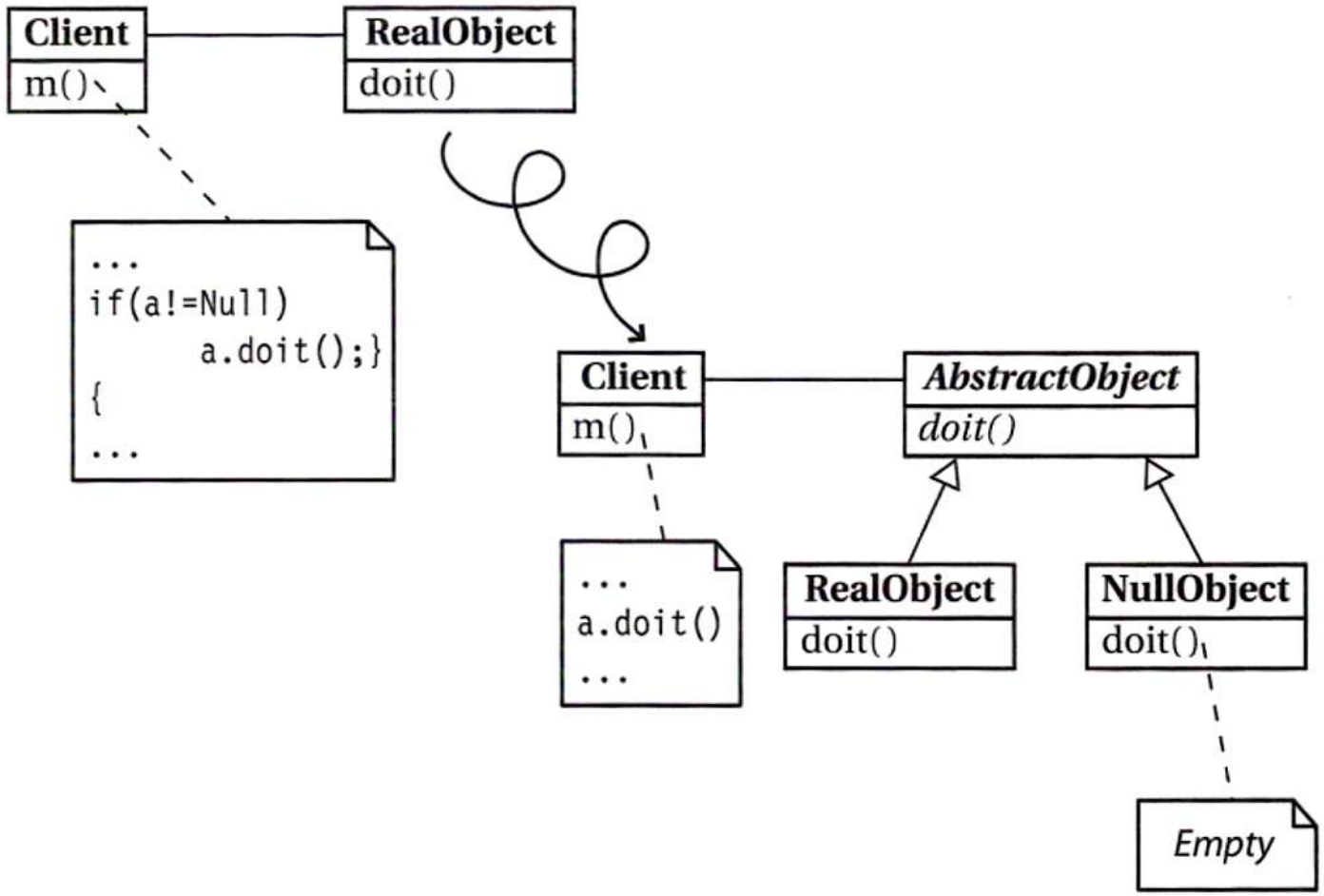


Transform Conditionals to Polymorphism: Introduce Null Object

- **Problem:** How can you ease modification and extension of a class in the presence of repeated tests for null values?

- **Solution:** Apply *Null Object*, that is,
  - encapsulate the null behavior as a separate provider class so that the client class does not have to perform a null test.

Introduce Null Object

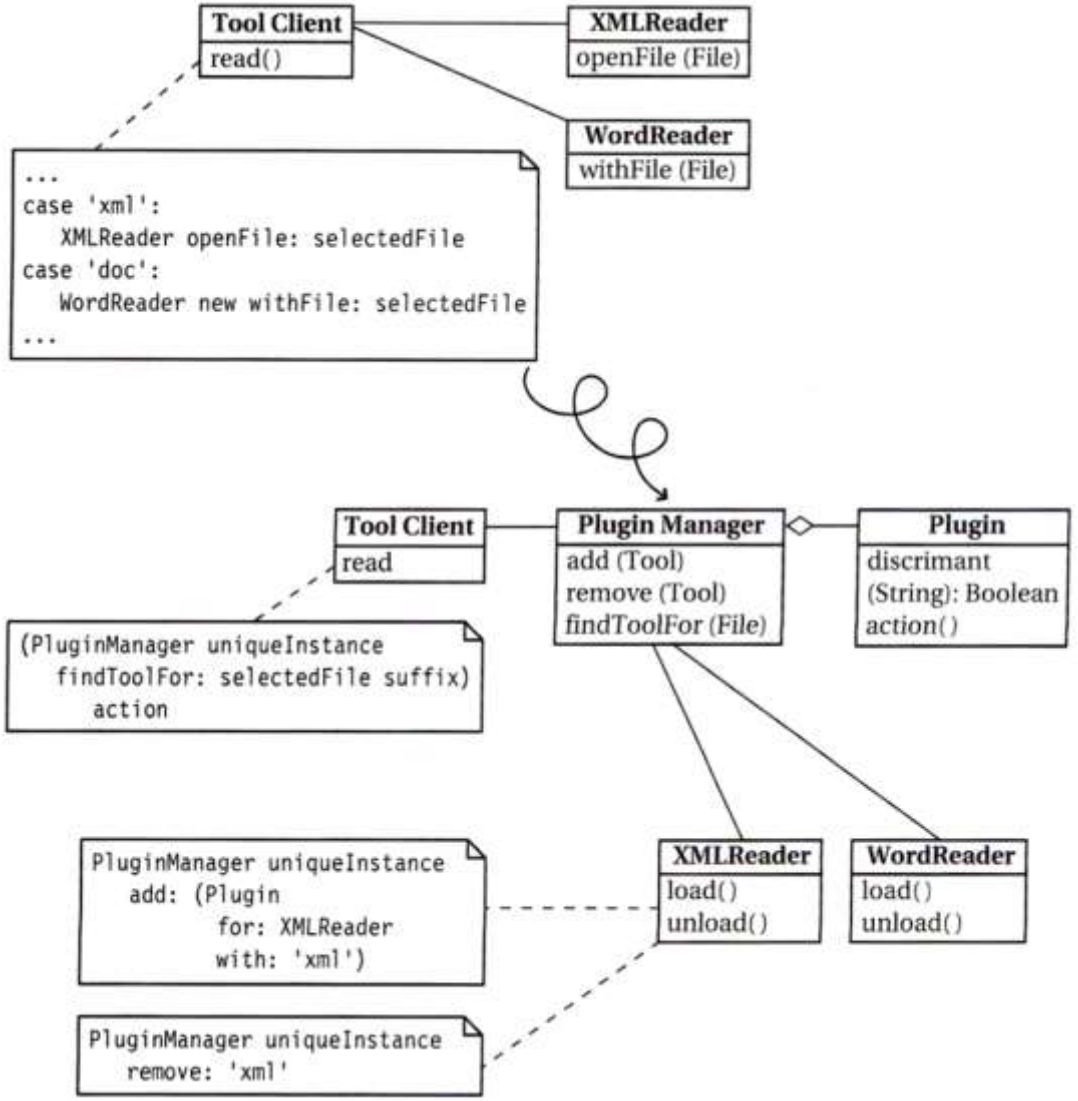


Transform Conditionals to Polymorphism: Transform Conditionals into Registration

- **Problem:** How can you reduce the coupling between *tools* providing services and *clients* so that the addition or removal of tools does not lead to changing the code of the clients?

- **Solution:**
  - Introduce a *registration mechanism* to which each tool is responsible for registering itself, and
  - transform the tool clients to query the registration repository instead of performing conditionals.

Transform Conditionals into Registration



Reference

- Demeyer, S., Ducasse, S., and Nierstrasz, O., *Object-Oriented Reengineering Patterns,* Elsevier Science, 2003.