Pattern in Software Engineering

Lecture Summary

Authors: Thomas Pettinger
Nils Kunze
Benedikt Schlagberger

2016-02-22

Pattern in Software Engineering
Technische Universität München

Table of Contents

1 Introduction

Original pattern definition: A pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.
categorization of patterns:

- Patterns for Development Activities
  - Analysis Patterns
  - Architecture Patterns
  - Design Patterns
  - Testing Patterns
- Patterns for Crossfunctional Activities
  - Process Patterns
  - Agile Patterns
  - Build and Release Management Patterns
- Antipatterns (Smells)

1.1 Conclusion

- Patterns are Knowledge
- Reusable source for solving problems
- We acquire and describe knowledge to solve recurring design problems
- Patterns are a great way to describe reusable knowledge
- There are even Antipatterns: They are useful for describing lessons learned
- Knowledge is often acquired by accidents or through failure
- Learning from failures is important
- Popper's concept of falsification

2 Object-Oriented Programming

This chapter explains concepts and terms of object-oriented programming. The following topics are covered in this chapter:

- Foundations of object-oriented programming
- Features of object-oriented programming languages
- Java and UML syntax
- Coupling and Cohesion
- Polymorphism
- Binding
- Delegation

2.1 Foundations of Object-Oriented Programming

Phenomenon and Concept

Phenomenon is an object in the world of a domain as you perceive it (e.g. watch on your wrist). Concept describes the common properties of a phenomena (e.g. all watches). A Concept is a 3-touple of name (e.g. watch), purpose (e.g. measure time) and members (e.g. hourglass, digital watch, ...).

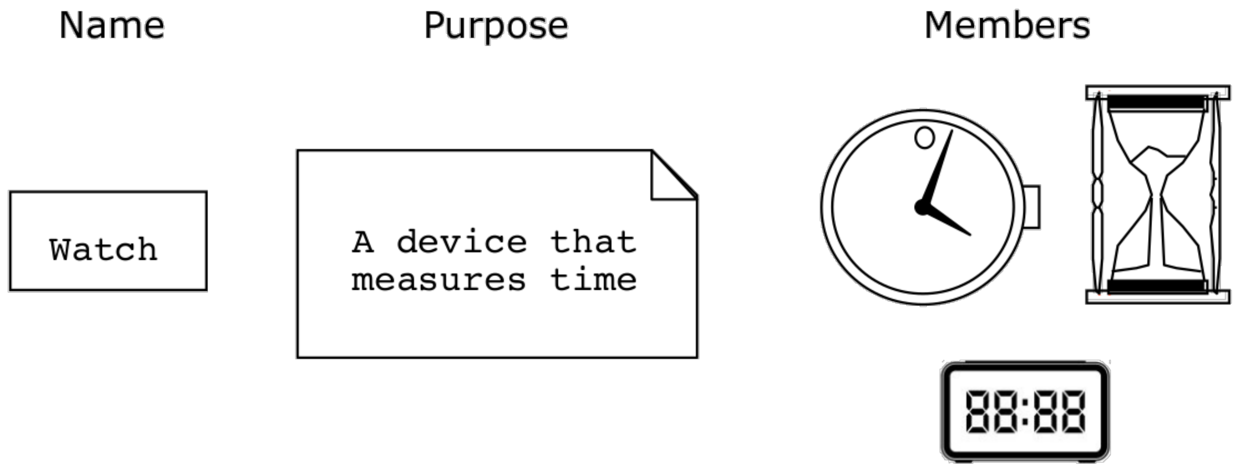


Figure 1: Abstraction of Watch

Abstraction and Modeling

Abstraction is the classification of a phenomena into concepts. Modeling is the development of abstractions to answer specific questions about a set of phenomena while ignoring irrelevant details.

Type and Instance

Type is a *concept* in the context of programming languages e.g.:

Name: int	**Name**: boolean
Purpose: set of integers	**Purpose**: logical truth values
Members: 0, -1, 1, 2, -2, . . .	**Members**: true, false

Instance is a member of a specific type. The type of a variable represents all possible instances of the variable. Types can be *simple* (primitive) (e.g. int, double, boolean, . . .) or *complex* (e.g. Car, Person, FileSystemService, . . .).

Classes and Objects

In most object-oriented programming languages, a complex type is represented by a class. A class is a code template for a concept that is used to create instances of that concept. A class has *attributes (properties)* and *methods (operations)* (e.g. *color* of a car, car can *drive*). Methods and attributes are called the *members* of a class.

An instance of a class at runtime is called *Object*.

2.2 Features of object-oriented programming languages

Abstraction

Creating a model of the problem in terms of classes and the relationships between them.

Inheritance

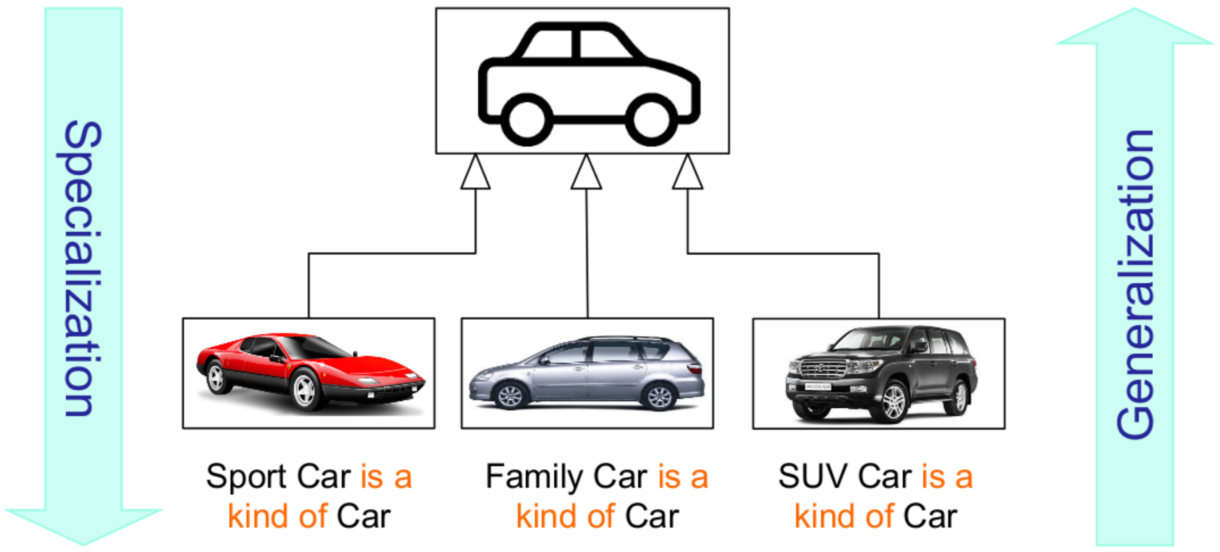


Figure 2: Inheritance of the class „Car“

Encapsulation

Objects are self-contained sets of data and behavior. By using access modifiers such as `public`, `private` or `protected`, the object can determine which part of its data and behavior is exposed to the outer world. The exposed parts of an object are called its *public interface*.

Information Hiding

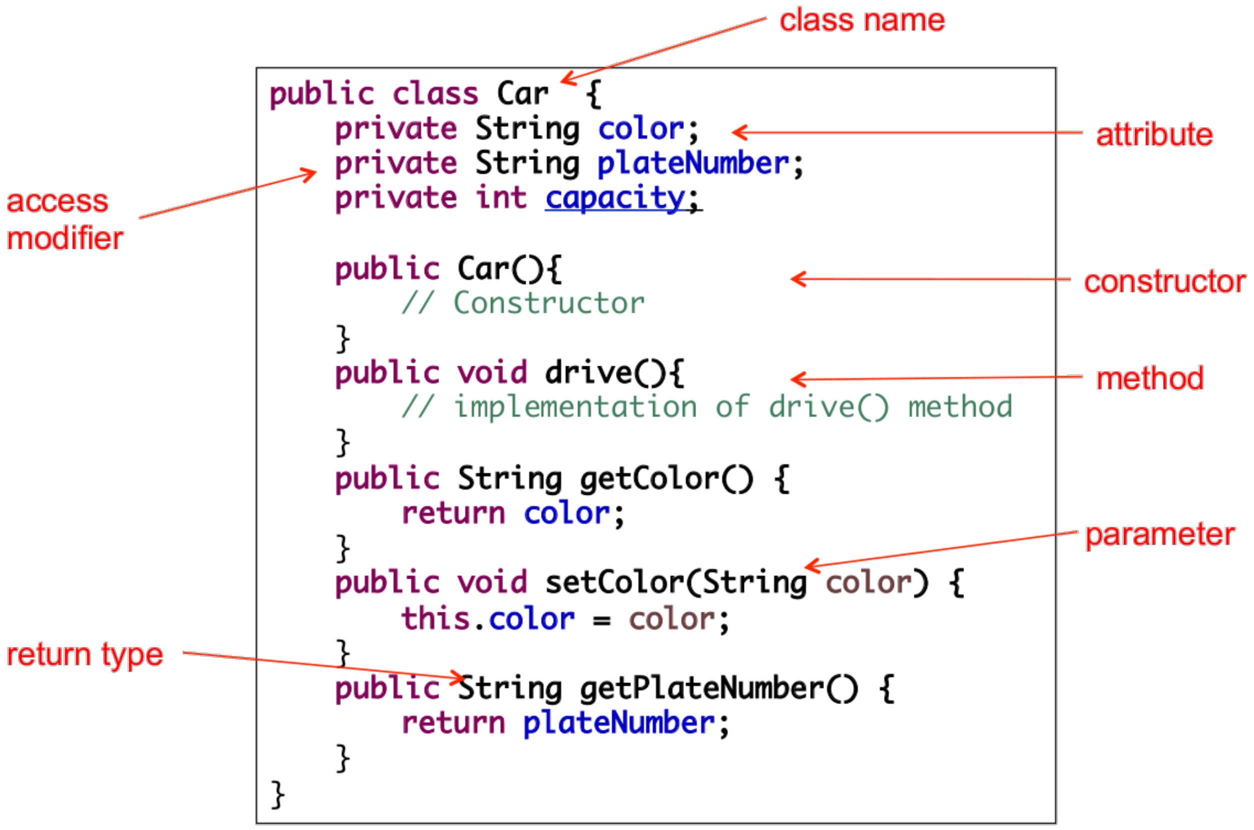
Principle of information hiding (David Parnas): A calling module (class, subsystem) does not need to know anything about the internals of the called module. This can be achieved by making all attributes and operations of a class private, unless the operations are needed by the class' user. Only public methods can be used to modify a class' attributes.

Abott's Heuristics

This approach analyses natural language to identify objects, attributes and associations from problem descriptions.

Part of speech	**Model component**	**Example**
Proper noun	Instance	Alice
Common noun	Class	Field officer
Doing verb	Operation	Creates, submits, selects
Being verb	Inheritance	Is kind of, is one of
Adjective, Genitive case	Attribute	Incident description

2.3 Java and UML Syntax and Structures



```java
public class Car {
    private String color;
    private String plateNumber;
    private int capacity;

    public Car(){
        // Constructor
    }
    public void drive(){
        // implementation of drive() method
    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public String getPlateNumber() {
        return plateNumber;
    }
}
```

Figure 3: Java class of „Car“ and its parts

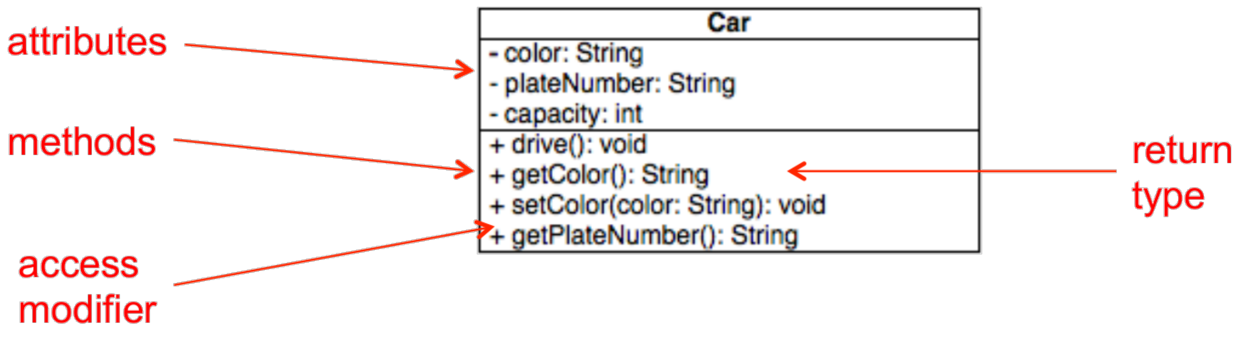


Figure 4: UML class diagram of „Car“ and its parts

2.4 Coupling and Cohesion

Coupling measures the dependencies between subsystems, *cohesion* the dependencies within a subsystem. A good design uses low coupling as this makes the subsystem as independent of each other as possible and therefore, changes in one subsystem won't affect other subsystems. It also uses high cohesion, because subsystems only contain classes which heavily depend on each other. This approach reduces *complexity* and *eases change* of the system.

To achieve low coupling and high cohesion, the *Principle of Least Knowledge*, also called *Law of Demeter* (Ian Holland) should be followed:
A method M of an object O may only invoke the methods of the following kinds of objects:

- O itself
- M's parameters
- Any objects created/instantiated within M
- O's direct component objects

In Java, this can be achieved by using only one level of method calls, e.g. `object.doSomething()`, not two or even more levels e.g. `object.getObjectP().doSomething()`. This produces less coupling and is not dependent on implementation details. On the downside, many wrappers may be needed, which can lead to inefficiency.

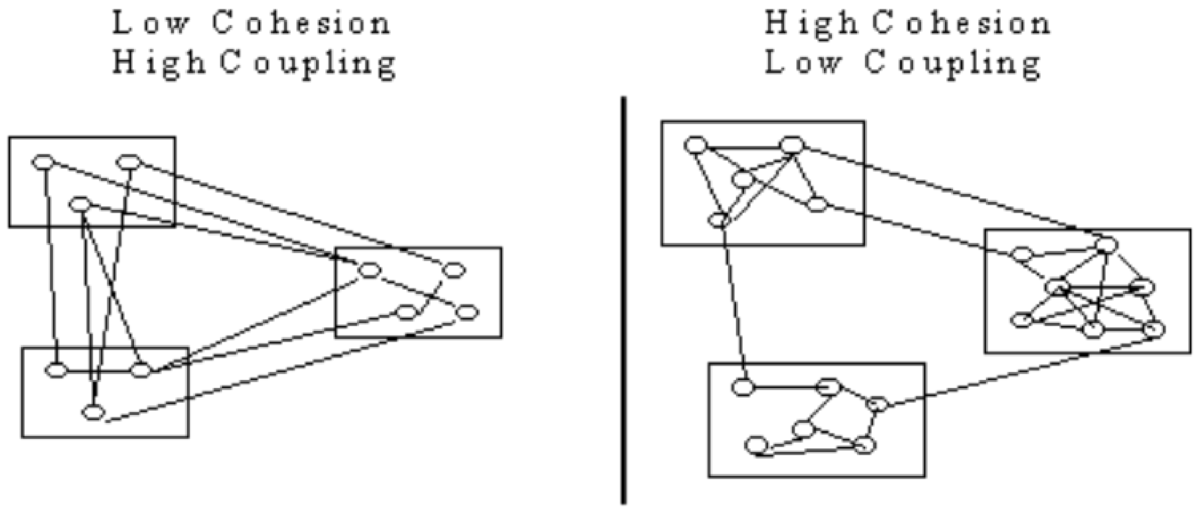


Figure 5: Coupling and Cohesion

2.5 Polymorphism

Polymorphism is the ability of an object to assume different forms or shapes. In computer science this is the ability of an abstractions to be realized in multiple ways. Concretely, this comes down to different ways interfaces can be realized. Objects can hereby be dynamically treated based on their type.

Parametric Polymorphism

In Java parametric polymorphism can be seen in generic types and operations. A generic type has a type parameter e.g. `List<E>`. A type parameter is a placeholder for a specific type which must be specified at the instantiation of a variable of the generic type. Operations on generic types are called *generic operations.*

Inheritance: Subtyping

Type A is a subtype of another type B, exactly when all A, considered as a set of values, is a subset of B. A can be substituted by B. This is called Liskov' substitution principle:

If for each object O_S of type S there is an object O_T of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when O_S is substituted for O_T, then S is a subtype of T.

— Barbara Liskov, MIT

If a subtype follows the substitution principle, the following rules apply:
Operations:

- All supertype operations must have corresponding subtype operations
- Each subtype operation

    must have a *weaker precondition*: require less (or the same) than the superclass operation

    must have a *stronger postcondition*: guarantee more (or the same) than the superclass operation

Invariants (called properties by Liskov):

- Any invariant of the supertype (e.g. value constraints) must be guaranteed by the subtype as well

Interfaces and Classes in Java do not guarantee any properties. Therefore, they are not subtypes according to Liskov's definition.

Inheritance: Subclassing

Subclassing denotes the usage of an inheritance association. *Overriding* is a special case in subclassing where a method implemented in the superclass is reimplemented in the subclass. The selection of a method depends on the type of the object at runtime. This allows to change the behavior of an object without extensive case distinctions.

Ad-hoc Polymorphism: Overloading

The ability to let a feature name denote two or more operations is called *overloading*. Selection of the corresponding methods is decided by the signature. This decision is made by the compiler. The signature is defined as the name and parameter types of a method.

2.6 Binding

Binding establishes the mapping between names and data objects and their descriptions.

Early Binding (Static binding, at compile time)

The premature choice of operation variant, resulting in possibly wrong results and (in favorable cases) run-time system crashes.

Late Binding (Dynamic binding, at run time)

The guarantee that every execution of an operation will select the correct version of the operation, based on the type of the operation's target.

2.7 Delegation

Delegation is a mechanism for code reuse in which an operation resends a message to another class to accomplish the desired behavior. It involves passing a method call to another object, transforming the input if necessary. By that, the behavior of an object is extended.

3 Design Patterns

Design patterns are generalizations of detailed design knowledge from existing systems. They provide a shared vocabulary and examples of reusable designs to designers (e.g. polymorphism, delegation/aggregation). In summary, there are three types of design pattern:

- **Structural Patterns**
  - Reduce coupling between two or more classes
  - Introduce an abstract class to enable future extensions
  - Encapsulate complex structures
- **Behavioral Patterns**
  - Allow a choice between algorithms and the assignment of responsibilities to objects („Who does what?“)
  - Simplify complex control flows that are difficult to follow at runtime
- **Creational Patterns**
  - Allow a simplified view from complex instantiation processes
  - Make the system independent from the way its objects are created, composed and represented

The following sections list some of the most relevant patterns of those categories.

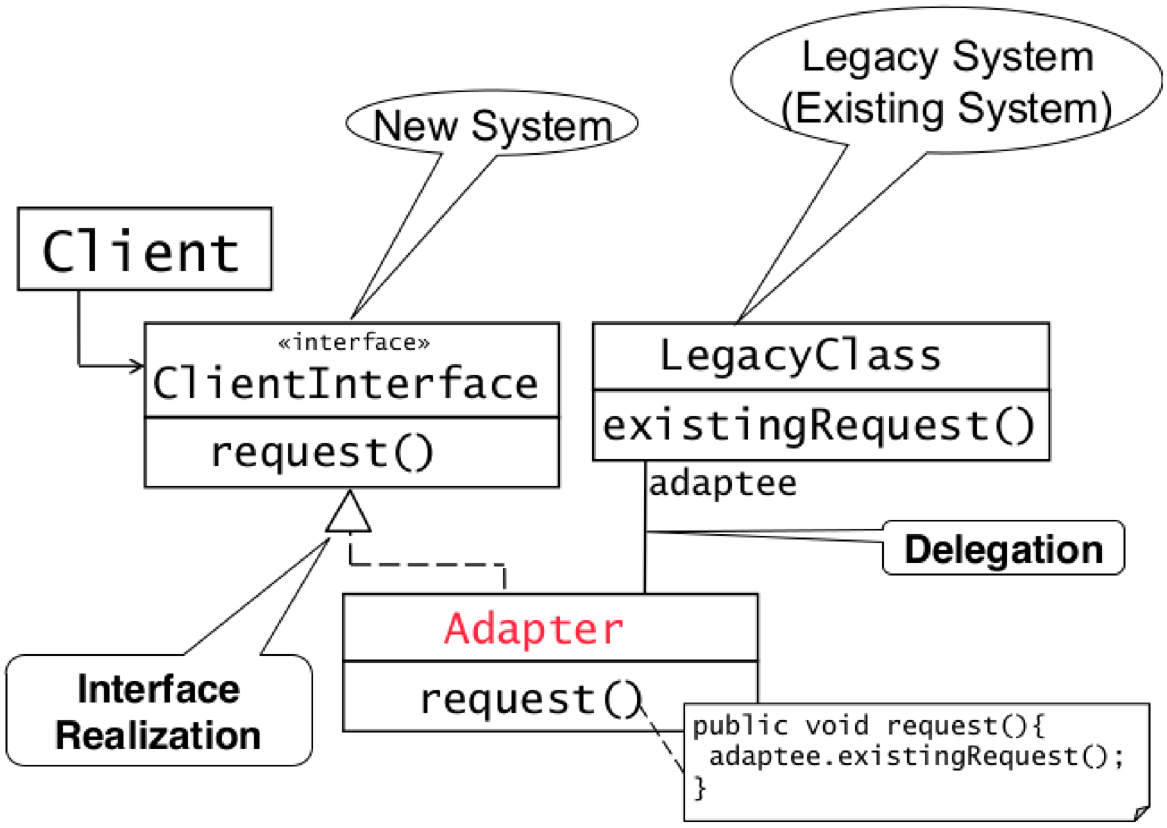
3.1 Structural Patterns

Structural patterns aim for less coupling between classes and easier extensibility by encapsulating complex structures.

3.1.1 Adapter Pattern/Wrapper

Connects incompatible components to for example

- Reuse existing components
- Convert an interface to another interface (maybe needed by an API call)



```
public void request(){
 adaptee.existingRequest();
}
```

Figure 6: Structure of Adapter Pattern/Wrapper

3.1.2 Bridge Pattern

Allows to delay the assignment of an implementation of an interface from compile to run time. The **degenerated bridge pattern** is the same as the bridge pattern without the taxonomy in the application domain.

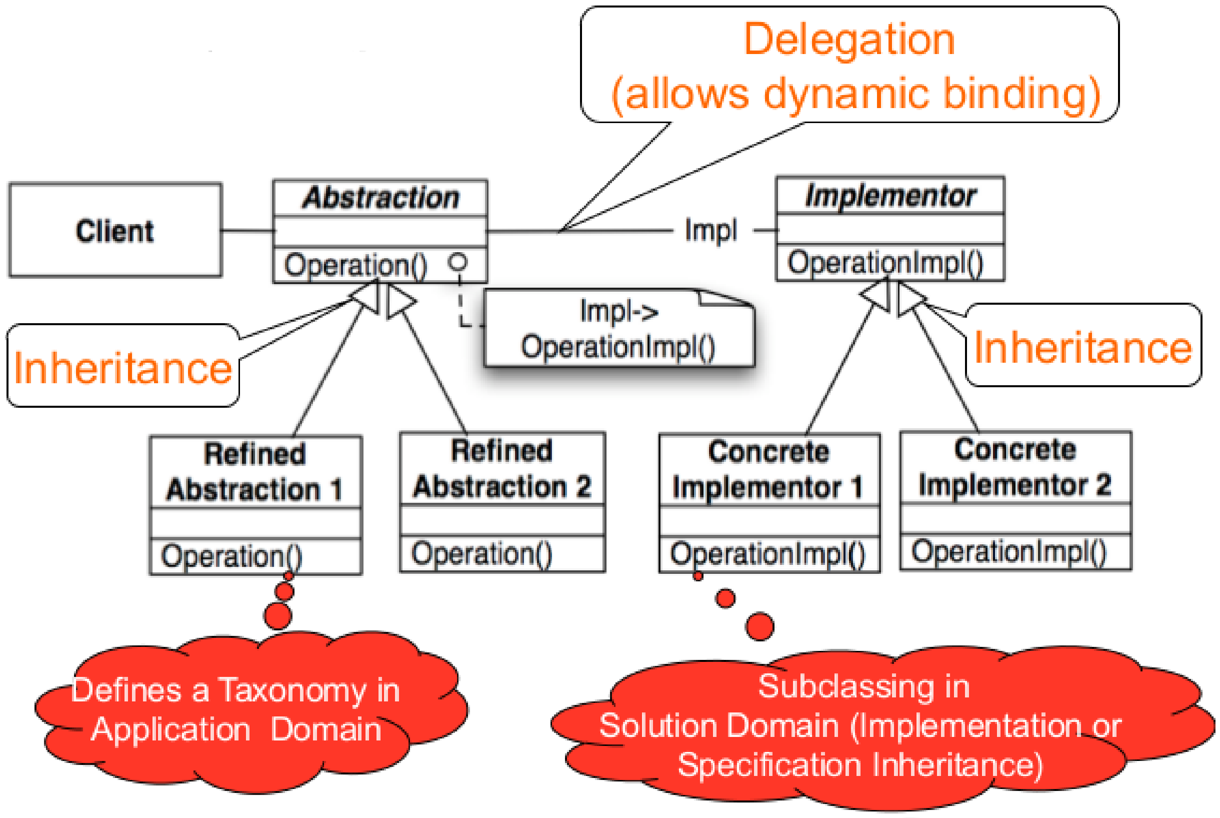


Figure 7: Structure of Adapter Pattern

3.1.3 Proxy Pattern/Caching

The proxy pattern allows to defer object creation and object initialization to the time you need the object (Remote Proxy (Caching), Substitute (Virtual Proxy), Protection Proxy (Access control/Firewall)).

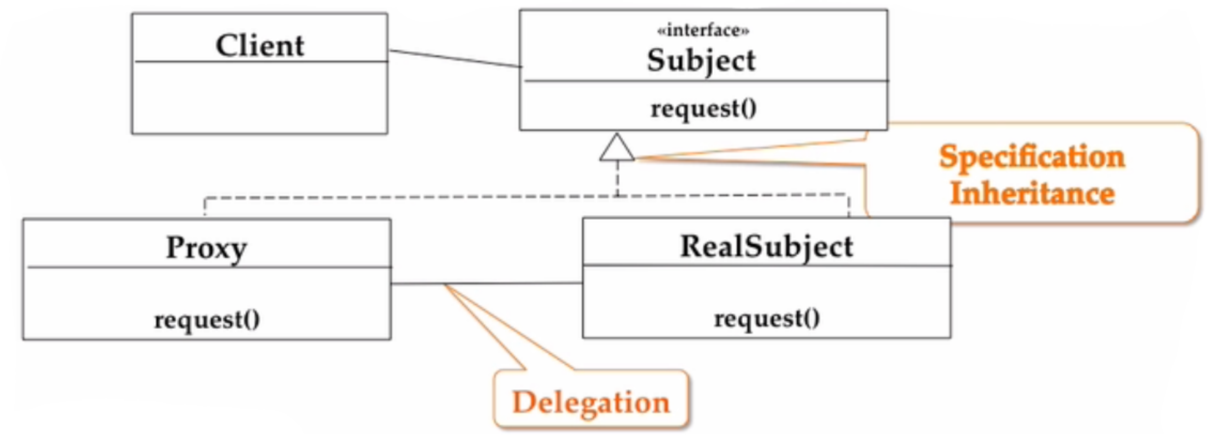


Figure 8: Structure of Proxy Pattern/Caching
The client never calls `request()` *in* `RealSubject`*, instead it always calls the method in* `Proxy` *which might delegate it to* `RealSubject`*.*

3.1.4 Composite Pattern

The composite pattern models tree structures that represent part-whole hierarchies with arbitrary depth and width. It lets the client treat individual objects and groups uniformly.

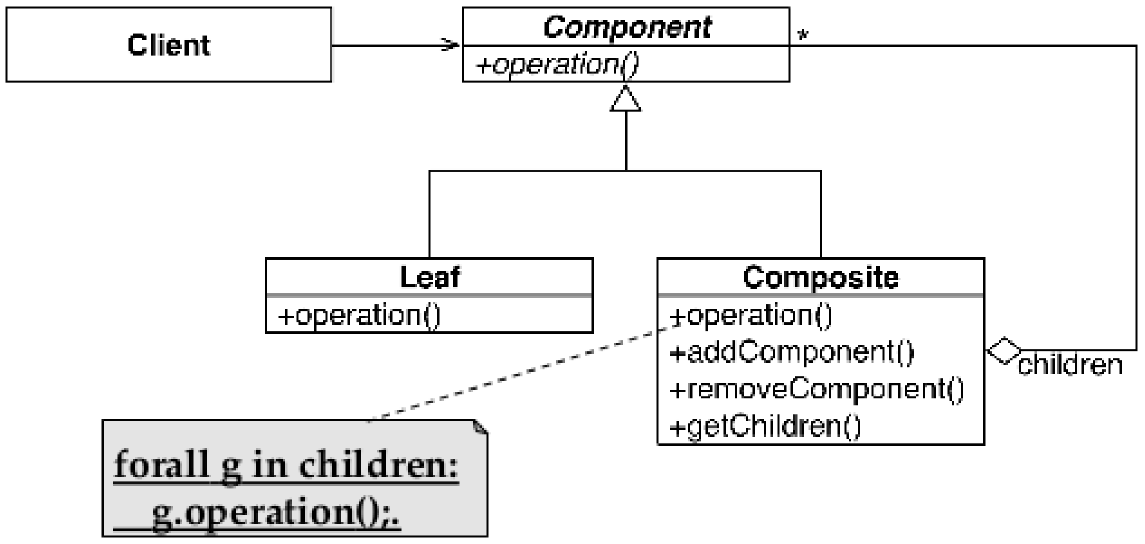


Figure 9: Structure of Composite Pattern

3.2 Behavioral Patterns

Behavioral patterns identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

3.2.1 Strategy Pattern

Suited for situations where different algorithms are available for a problem (e.g. sorting).

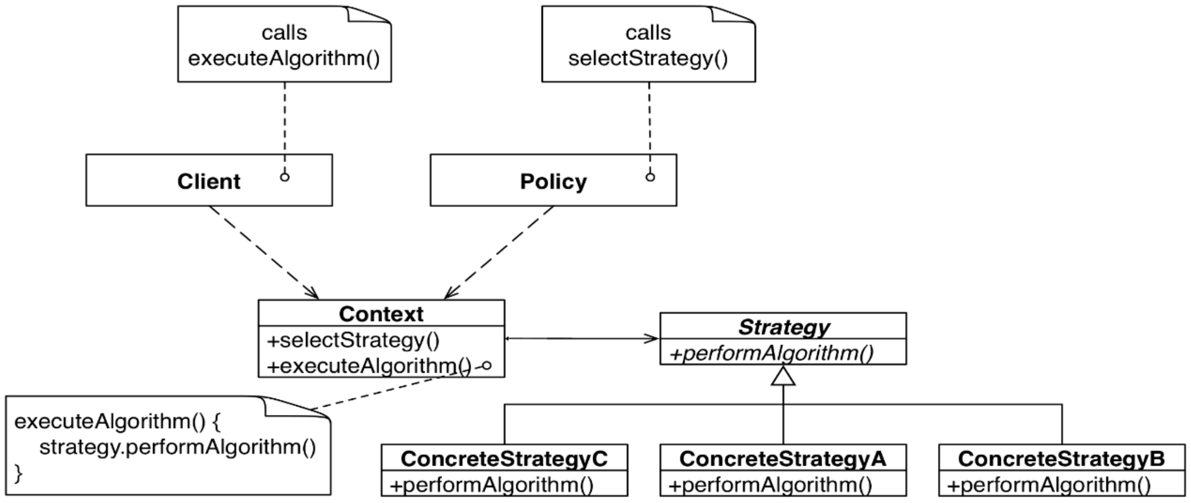


Figure 10: Structure of Strategy Pattern

A strategy is chosen on ***runtime*** *by the* `Policy` *class before the client calls* `executeAlgorithm()`

3.2.2 State Pattern

Dependent on the current state of a system, an action should do different things (e.g. TCP open, close). The state pattern avoids many if else statements and is flexible to add more cases/states.

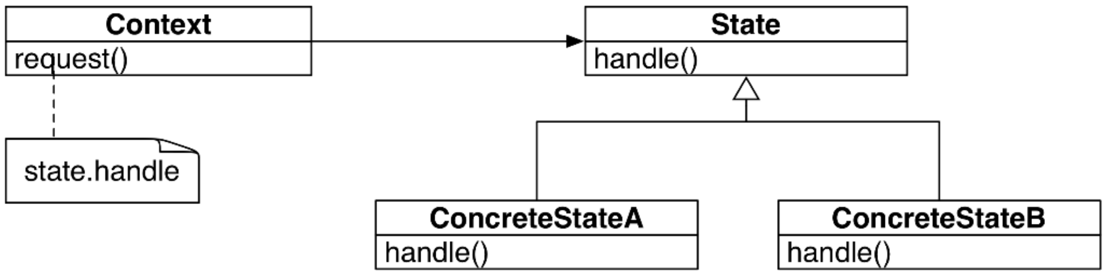


Figure 11: Structure of State Pattern

Problem: Where are state transactions handled? (in the exercise of the lecture in the states)

3.2.3 Observer Pattern

The observer pattern handles changes in a publisher class and notifies all subscribers about that change (e.g. the user interface) to maintain consistency. There are three variants for maintaining consistency:

- **Push Notification:** Every time a state changes, all subscribers are notified
- **Push-Update Notification:** The publisher also sends the state that has changed
- **Pull Notification:** A subscriber inquires about the state of the publisher

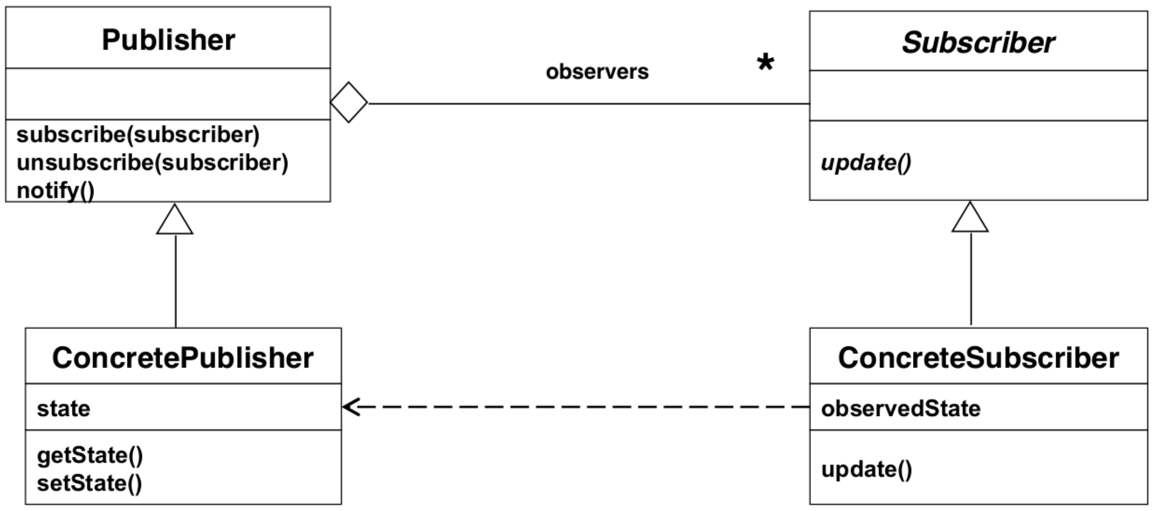


Figure 12: Structure of Observer Pattern

3.2.4 Model View Controller Pattern

The model-view-controller architectural style decouples data access and data representation. The view handles the data representation, the model the data access and the controller handles the communication between the other two.

In the **pull variant** the connection between the controller and the view is removed. In this variant the view asks the model for the data explicitly.
In the **push notification variant** both the connection between the view and the model and controller respectively are removed. When a change in the model occurs, the view and controller are updated via the observer pattern.

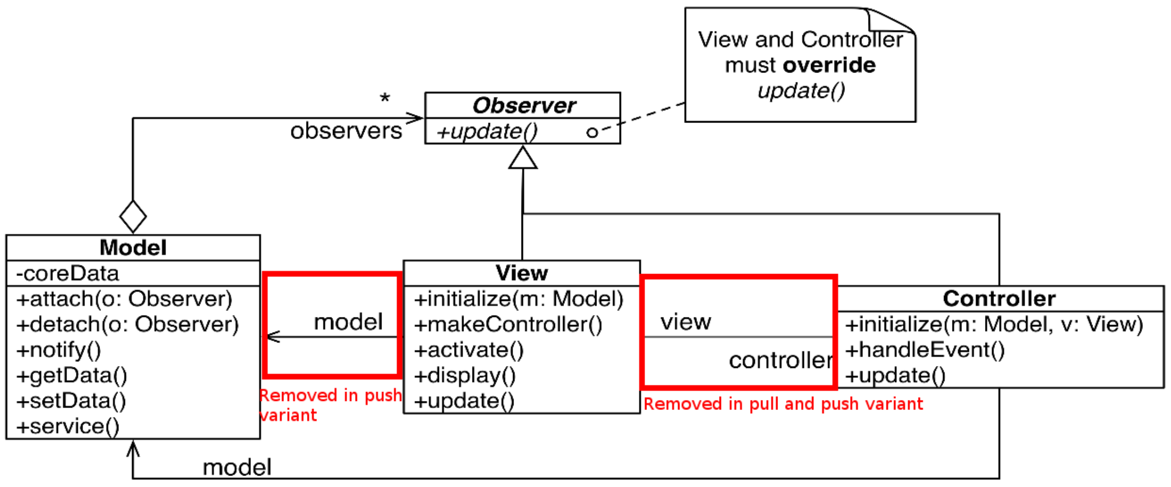


Figure 13: Structure of Model View Controller Pattern

3.2.5 Command Pattern

The command pattern is used to design user interfaces with multiple commands without using multiple `if`-statements (i.e. `if(command == x) ... else if(command == y) ...`). It can be used to make menus reusable across applications.

Implementing Commands as classes allows command histories and thus *undo* and *redo* operations.

Common Applications:

- **Command Manager** Central repository for all commands
- **Redo/Undo Manager**
- **Queue** Holds commands until others objects are ready to do something with them
- **Dispatcher** (e.g. keyboard event loop)

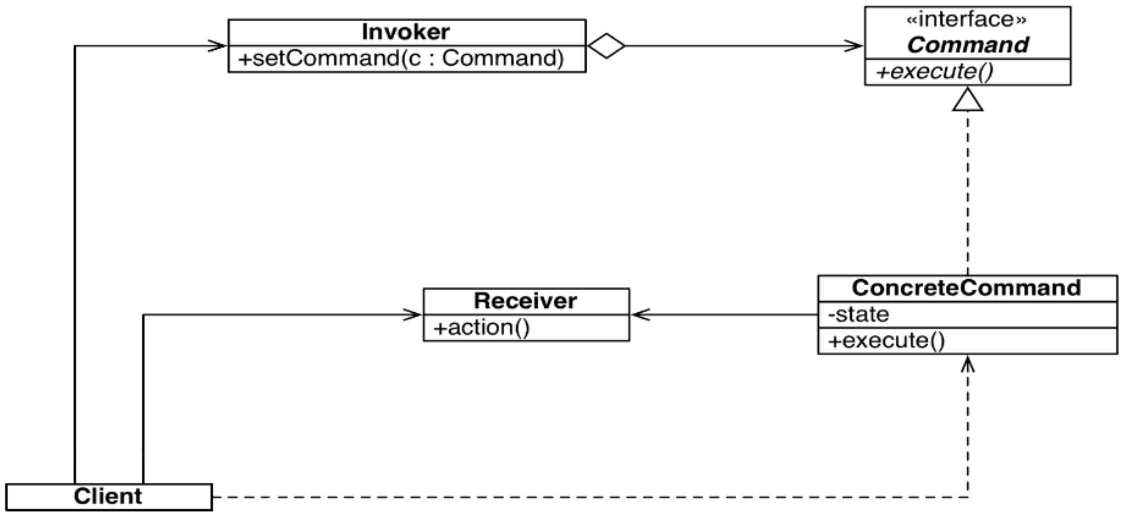


Figure 14: Structure of Command Pattern

3.3 Creational Patterns

Creational patterns allow a simplified view from complex instantiation processes and make the system independent from the way its objects are created, composed and represented.

3.3.1 Factory Pattern

A factory class handles the instantiation of objects inheriting from one superclass depending on a keyword or value.

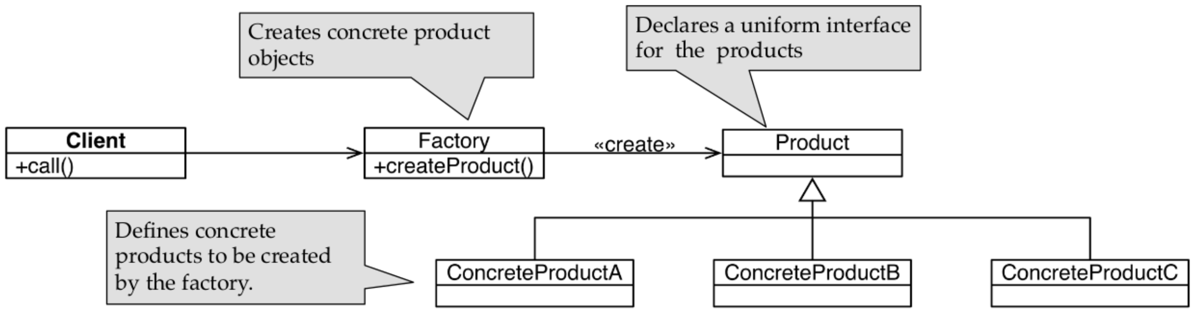

Creates concrete product objects
Declares a uniform interface for the products
Client
+call()
Factory
+createProduct()
«create»
Product
Defines concrete products to be created by the factory.
ConcreteProductA
ConcreteProductB
ConcreteProductC

Figure 15: Structure of Factory Pattern

3.3.2 Abstract Factory Pattern

The abstract factory pattern is used to instantiate or initialize an object consisting of more subparts. Every implementation of the abstract factory creates a set of components consisting of a variant of every part of the whole object.

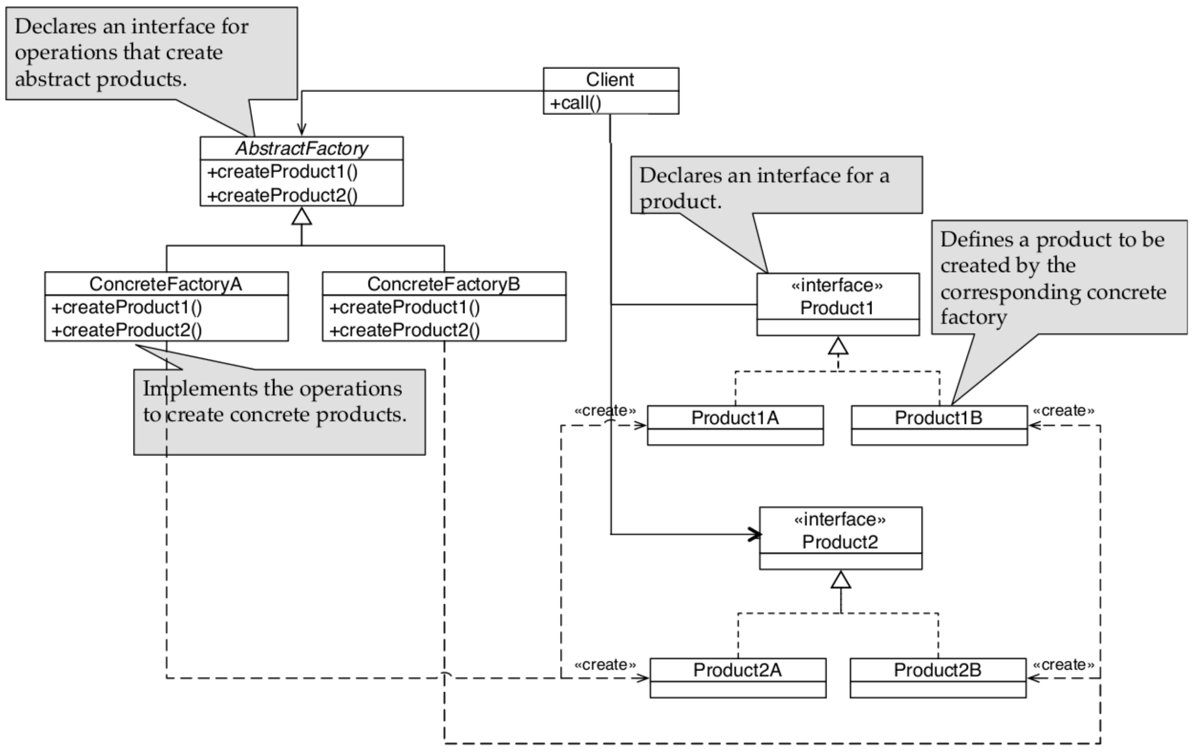
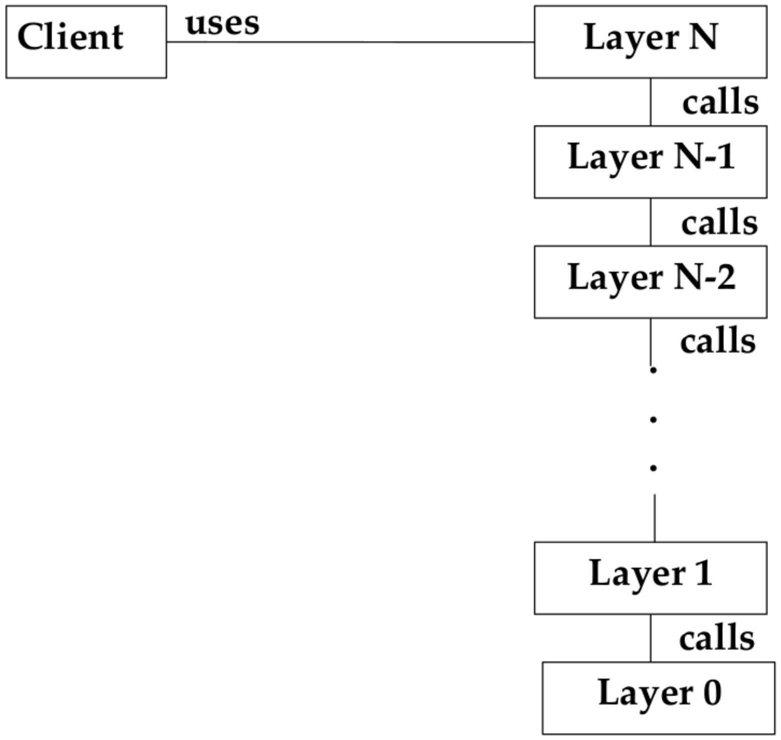


Figure 16: Structure of Abstract Factory Pattern

4 Architectural Patterns

4.1 Layer Pattern

Structure:



Closed Architecture (Opaque Layering):
Each layer can only call operations from the layer below.

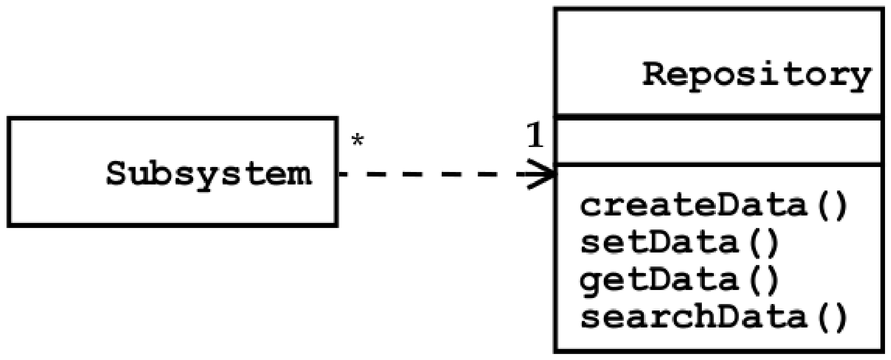
Open Architecture (Transparent Layering):
Each layer can call operations from **any** layer below

5 Steps to Create a Layered Architecture

1. Identify subsystems
2. Structure the individual layers
3. Specify the communication protocol between adjacent layers (push/pull)
4. Decouple adjacent layers
5. Design an error-handling strategy (try handling errors on lowest possible layer)

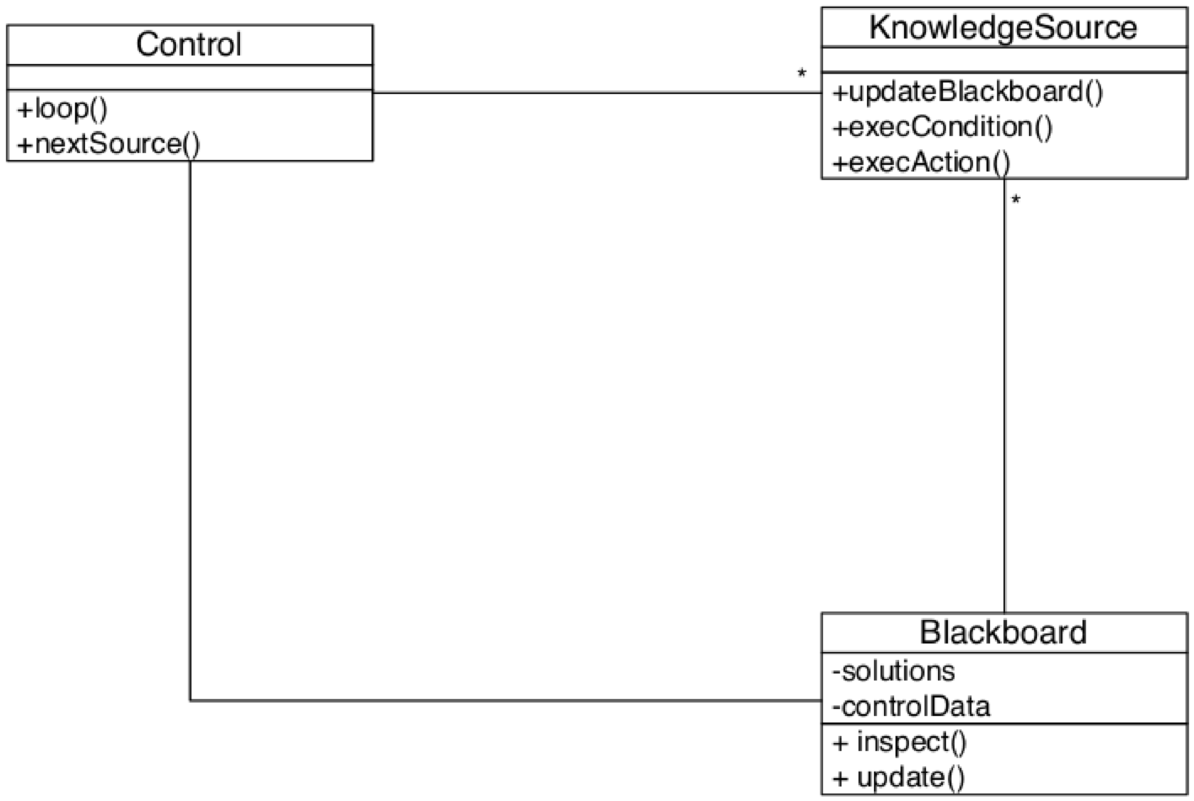
4.2 Repository Pattern

The repository pattern is used to support a collection of independent programs that work cooperatively on a common data structure called the repository. The control flow is not specified by the pattern.
Structure:



4.3 Blackboard Pattern

Experts throwing knowledge onto a blackboard (repository) which might be correct or not. Some can be extracted to higher order knowledge and other might be rejected.
Structure:



The knowledge sources inspect the current content of the blackboard and create new hypotheses. Control handles the control flow of the knowledge sources.
The blackboard pattern is used when no algorithm for the problem is known.

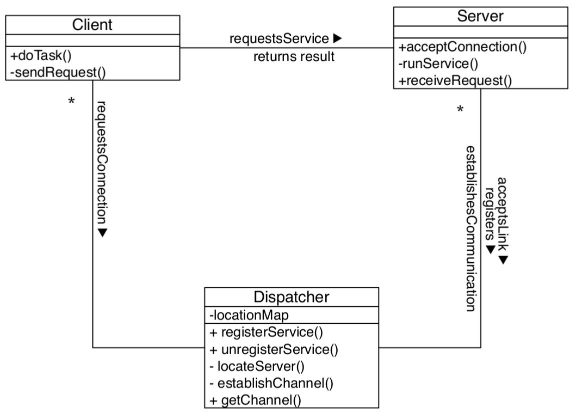
6 Steps to Realize a Blackboard Pattern

1. Define the Problem (Identify the application domain, the requirements and the actors)
2. Define the solution space (top-level and intermediate)
3. Identify the knowledge sources
4. Define the blackboard (not every information has to be understandable for every knowledge source)
5. Define control
6. Implement the knowledge sources (split into condition part and action part, use computational intelligence or conventional methods)

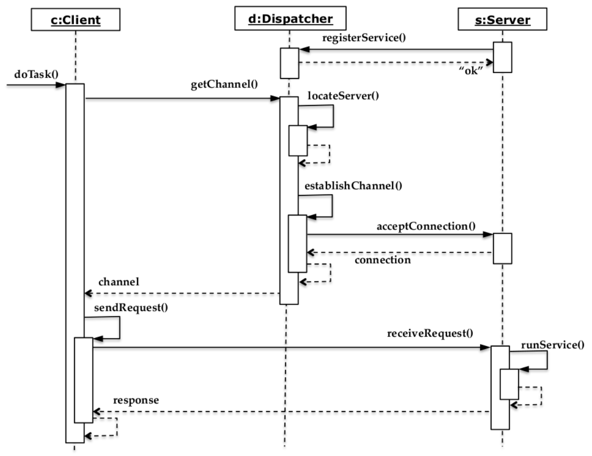
4.4 Client-Dispatcher-Server Pattern

The client-dispatcher-server pattern decouples the client from the server. Usually the client had to know where the server is, now the server is even dynamically interchangeable (server registers to the dispatcher on startup/runtime) what is good for re-configurations and fault tolerance.

Structure:



Eventflow:



Communication Protocols:

CDprotocol: Specifies how to ask for servers and handles communication errors.

DSprotocol: Specifies how the server registers with the dispatcher and determines the step necessary for a client to establish a connection.

CSprotocol: Specifies the communication between client and server.

6 Steps to Implement Client-Dispatcher-Server

1. During system design identify the subsystems that act as clients and servers
2. Decide on the communication mechanism to be used for the protocols (Shared memory, sockets)
3. Specify the protocols
4. Decide on a naming scheme for the dispatcher (URLs are ok, IPs not)
5. Implement the dispatcher
   (a) Decide how to implement the 3 protocols (sockets, RPM)
   (b) Implement requests, responses and errors
   (c) Implement a repository that maps server names to addresses
6. Implement the client and server (also: when does the server register? Startup or runtime?)

4.5 Broker Pattern

The broker pattern coordinates the communication between heterogeneous nodes.

Nonfunctional Requirements:

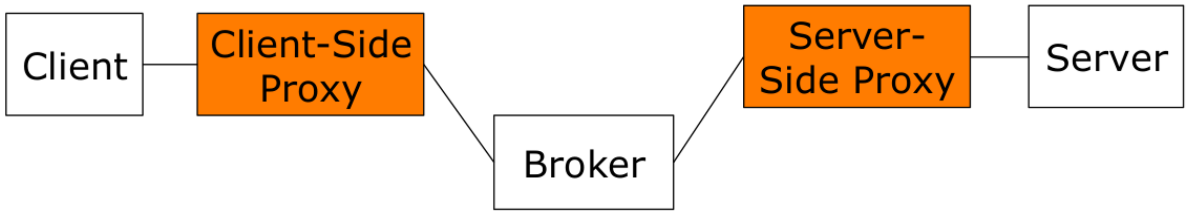
Low Coupling: Decoupling of service and communication

Location Transparency: Services are independent of the server location

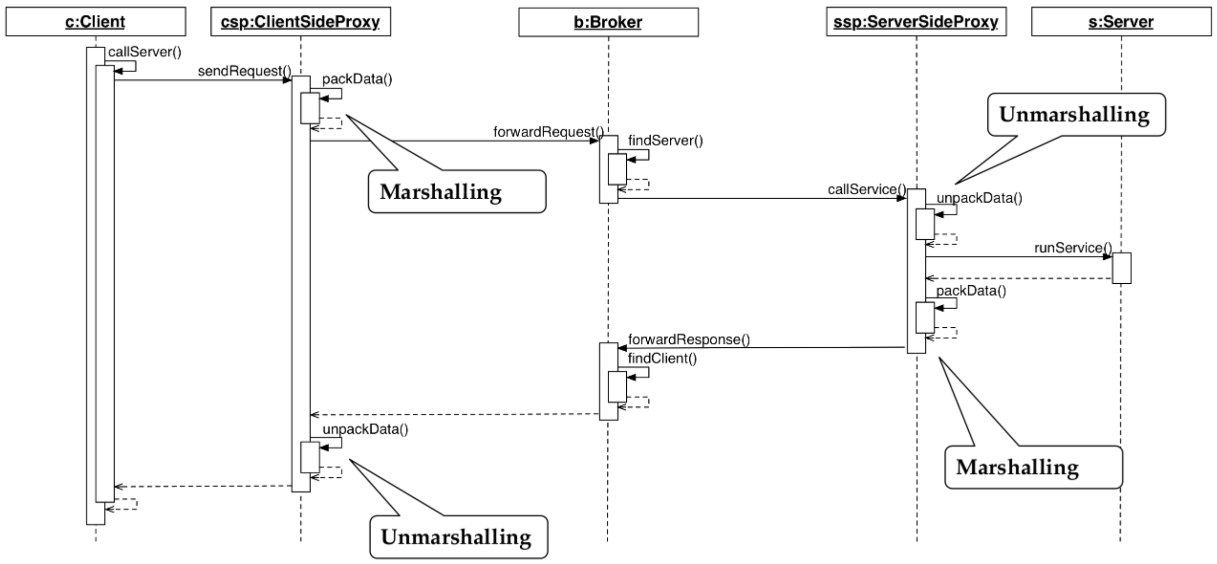
Runtime Extensibility: Ability to add, remove, exchange components at runtime

Platform Transparency: Clients and servers can be written in different languages.

Structure:



Client-Side Proxy: Lets the remote object appear as local one, hides the inter-process communication details used for message transfer between client and broker and provides (un-)marshalling/(de-)serialization of parameters and results.
Server-Side Proxy: Same as Client-Side proxy only for the server.
Eventflow:

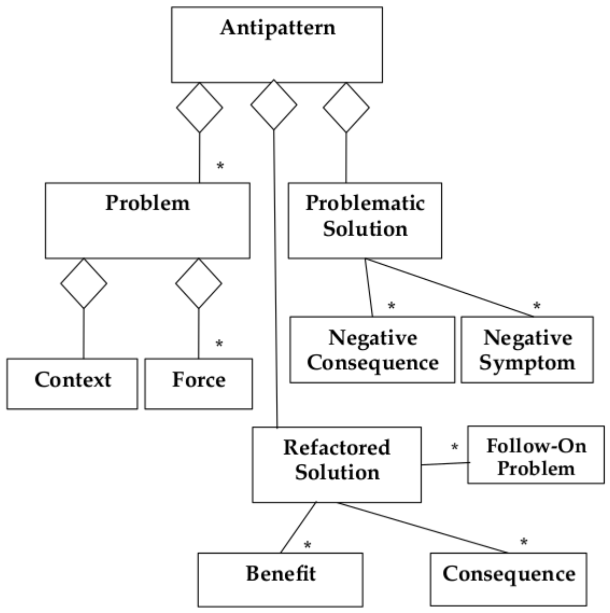


Steps to Realize a Broker Pattern:

1. Provide the object model and service definitions.
2. Define the broker service.
3. Implement the broker component and proxy object at the client and server side.
4. Implement the client and server.

5 Antipatterns

Developer antipatterns: Focus on the viewpoint of the software developer.
Issues: software refactoring, modification of source code to improve the software structure with respect to long-term maintainability.
Architecture antipatterns Focus on the viewpoint of the software architect.
Issues: partitioning of subsystems and components, platform independent definition of interfaces, and connectivity of components.
Management antipatterns Focus on the viewpoint of the software project manager.
Issues: software project organization, software project management, software process model, human communication, rationale management and resolution of issues.



7 Deadly Sins in Software Practice:

Apathy Not caring about a problem, followed by unwillingness To attempt a solution

Haste Solutions based on hasty decisions lead to compromises in software quality

Narrow-mindedness The refusal to use solutions that are widely known

Sloth Making poor decisions based on "easy" answeres

Avarice (excessive complexity) no use of abstractions, excessive modeling of details

Ignorance Failure to seek understanding

Pride Not invented here: not willing to adopt anything from the outside

5.1 Functional Decomposition Antipattern

Functional decomposition describes the decomposition of a system in terms of functions instead of use cases and/or objects (object-oriented decomposition) so that functions are hidden somewhere in the system where nobody might expect them.
recommended approach: first decompose in use cases, then in objects.

General Form Everything is a function, lots of files named misc, util,aux,...

Symptoms and Consequences

- Maintainer must understand the whole system to make changes
- Code is hard to understand
- Code is complex, high coupling between code sections in different files
- User interface is often awkward and non-intuitive

Typical Causes Wrong trained personal (programmers, designers)

5.2 Golden Hammer Antipattern

Everything is solved with a specific tool.

General Form

- Developer has high level of competence in a particular solution
- Every new development effort is solved with this solution
- Developer is unwilling to learn and apply new approach

Symptoms and Consequences

- Identical tools used for a many divers products
- System architecture depends on a specific tool chain

Typical Causes Large investment in product for specific technologies maybe with exclusive features.

Known Exceptions Product is part of a vendor suite that provides for all needs

5.3 Lava Flow

Also known as dead code.

General Form Lavalike flows of previous development hardened into a basaltlike mass of code, difficult to remove once solidified

Symptoms and Consequences

- Unused or commented-out code
- Undocumented complex, important-looking code
- Functions or classes that do not relate to the system architecture
- evolving architecture

Typical Causes

- Research and development code placed into production
- Implementation of several trial approaches
- High programmer turnover rate
- Fear of breaking something and not knowing how to fix it
- Unclear, repeatedly changing project goals
- Architectural scars

Exceptions Throwaway code

5.4 Blob Antipattern

Also known as god class

General Form Majority of responsibilities are in one complex controller associated with simple data classes

Symptoms and Consequences

- Huge class with many unrelated attributes and operations encapsulated
- The blob is usually to complex to reuse and test

Typical Causes

- Lack of architecture
- Too limited intervention in iterative projects

Known Exceptions Wrapping of legacy systems

6 Comparisons

6.1 Adapter vs. Bridge

Both hide the detail of the underlying implementation, but:

- the adapter (inheritance followed by delegation) is designed to handle incompatibilities
- the bridge (delegation followed by inheritance) is intended to differentiate between abstraction and implementation up-front

6.2 Bridge vs. Strategy

The bridge is used for structural decisions on system startup whereas the strategy handles behavioral decisions on runtime based on changing criteria.

6.3 Strategy vs. State

The strategy pattern handles different algorithms at runtime whereas the state pattern handles different states of an object in the architecture.

7 Terminology

Coupling measures the dependencies between subsystems

Cohesion Measures the dependencies among classes within a subsystem

Design Pattern describes associations and collaborations of a set of classes.

Architectural Style is a pattern for a subsystem decomposition, i.e. describes relationships and collaborations of different subsystems.

Software Architecture is an instance of an architectural style.

User Model is imagined by the user in their mind. It helps the user to know and understand the underlaying application domain model.

Natural Mapping (UI) is a mapping between UI controls of a system and objects in the real world such that the mapping does not tax the user's memory when performing a task that involves the manipulation of these controls.

Components/Subsystems Computational units with a specified interface

Connectors/Communication Interactions between the components/subsystems