Patterns in Software Engineering

Marcel Bruckner

Record Cards

WS 2017/18

Prof. Bruegge

Patterns

- 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 falsication

General

Phenomenon	• Object in the world as perceived
Concept	• Describes common properties of phenomena • <name, purpose, members>
Abstraction	• Classification of phenomena into concepts
Modeling	• Development of abstractions to answer specific questions about a set of phenomena while ignoring irrelevant details
Type	• Concept in context of programming languages
Instance	• Member of specific type
Class	• Code template for a concept that is used to create instances

General

Abstraction	• Creating a model of the problem in terms of classes and relationships
Inheritance	• Super-/Subclasses ⟷ Generalization/Specialization
Encapsulation	• Objects are self-contained sets of data and behavior • Access modifiers → Determine which data and behavior exposed to outer world • Exposed part called public interface
Information Hiding	• A calling module does not need to know anything about internals of called module • This can be achieved by making all attributes and operations private unless operations needed by class‘ user • Only public methods used to modify a class‘ attribute

Polymorphism

Polymorphism	• Ability of an abstraction to be realized in multiple ways
Parametric Polymorphism	• Generic Types and Operations • Type parameterliste
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. B can be substituted by A.
Liskov' substitution principle	• If for each object OS of type S there is an object OT of type T such that for all programs P dened in terms of T, the behavior of P is unchanged when OS is substituted for OT, then S is a subtype of T. • All subtype operations must have corresponding subtype operations • Weaker preconditions and stronger postconditions
Subclassing	• Usage of inheritance • Overriding • Selection of method based on type of object at runtime
Overloading	• One featurename for one or more operations • Selection decided by signature

Binding

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 (Dynaming 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.

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.

UML Syntax

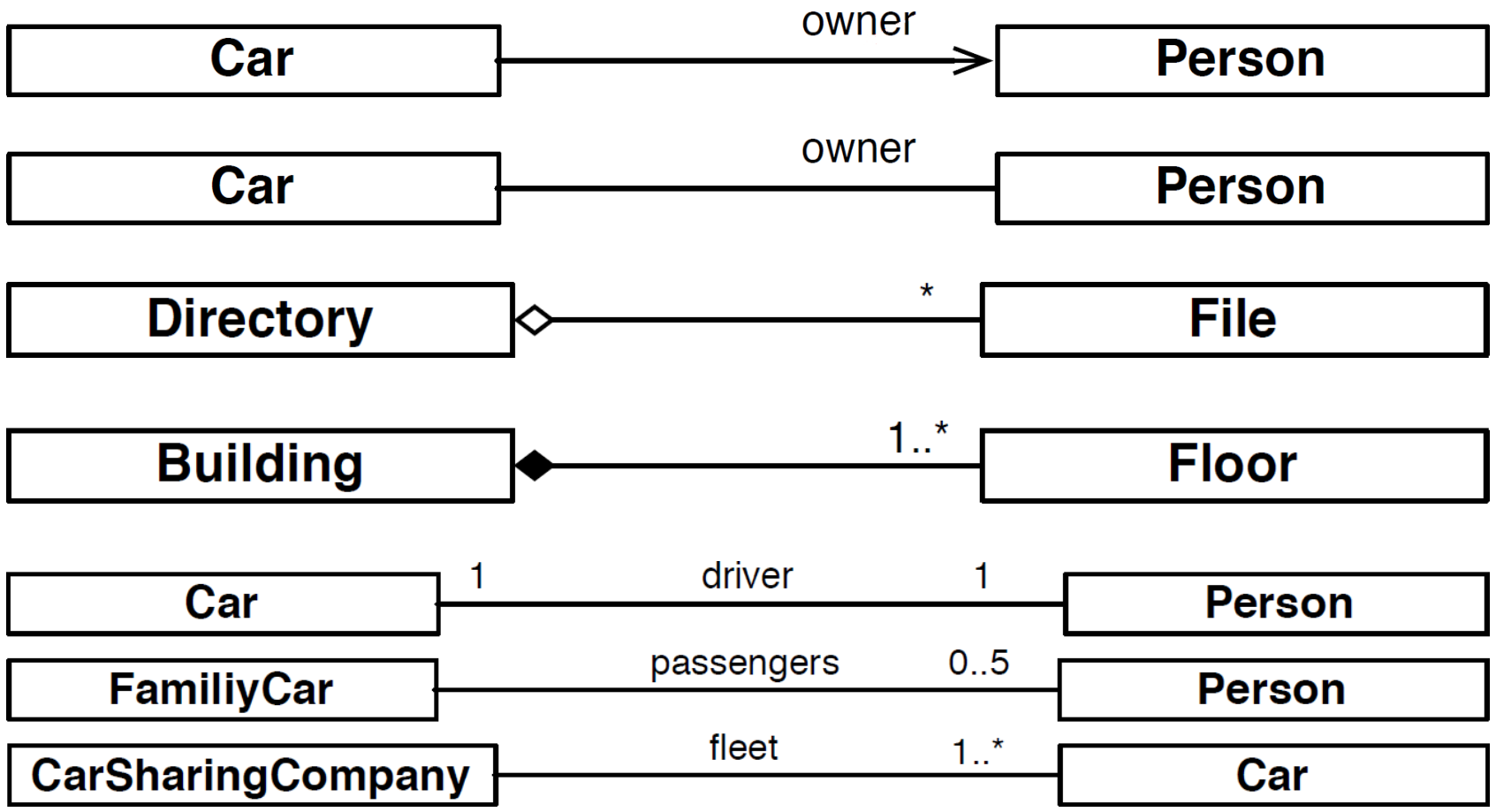
Association

Link

Aggregation
- Part of, consists of

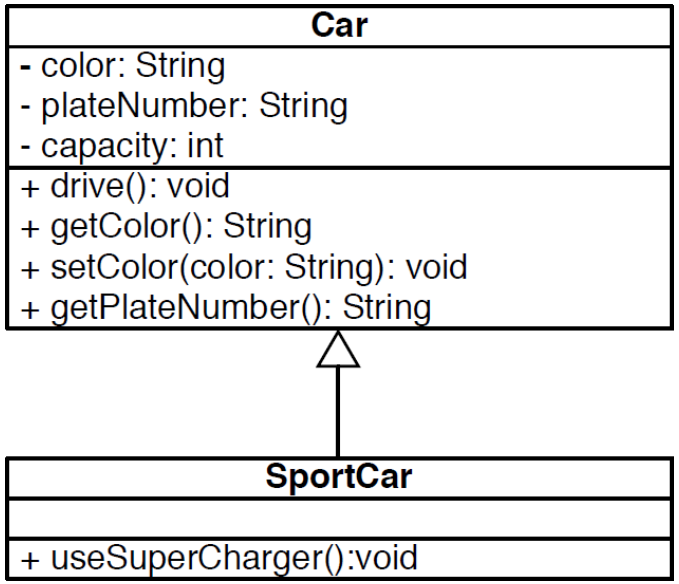
Composition
- Parts cannot exist without whole

Multiplicity

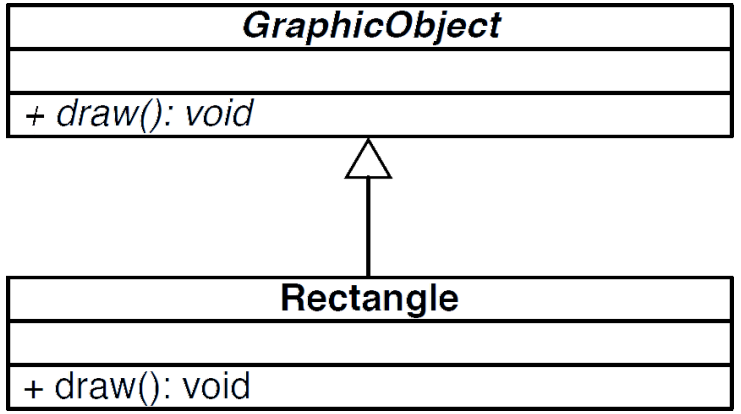


UML Syntax

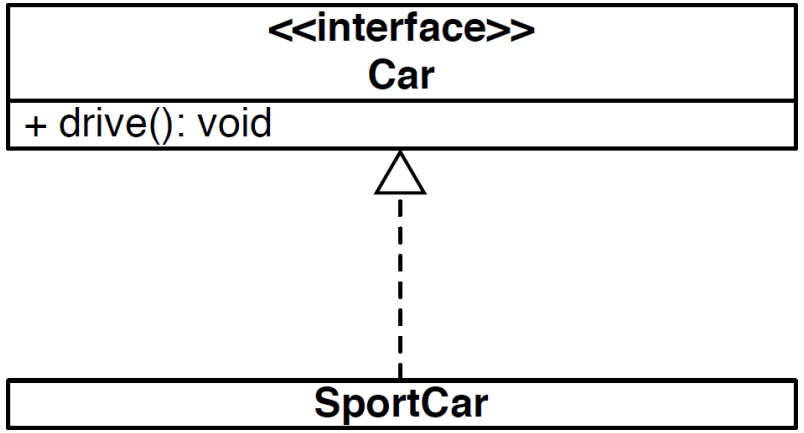
Inheritance



Abstract class



Interface



Design Patterns

General

Structural Patterns

- Reduce coupling between two or more classes
- Introduce an abstract class to enable future extensions
- Encapsulate complex structures

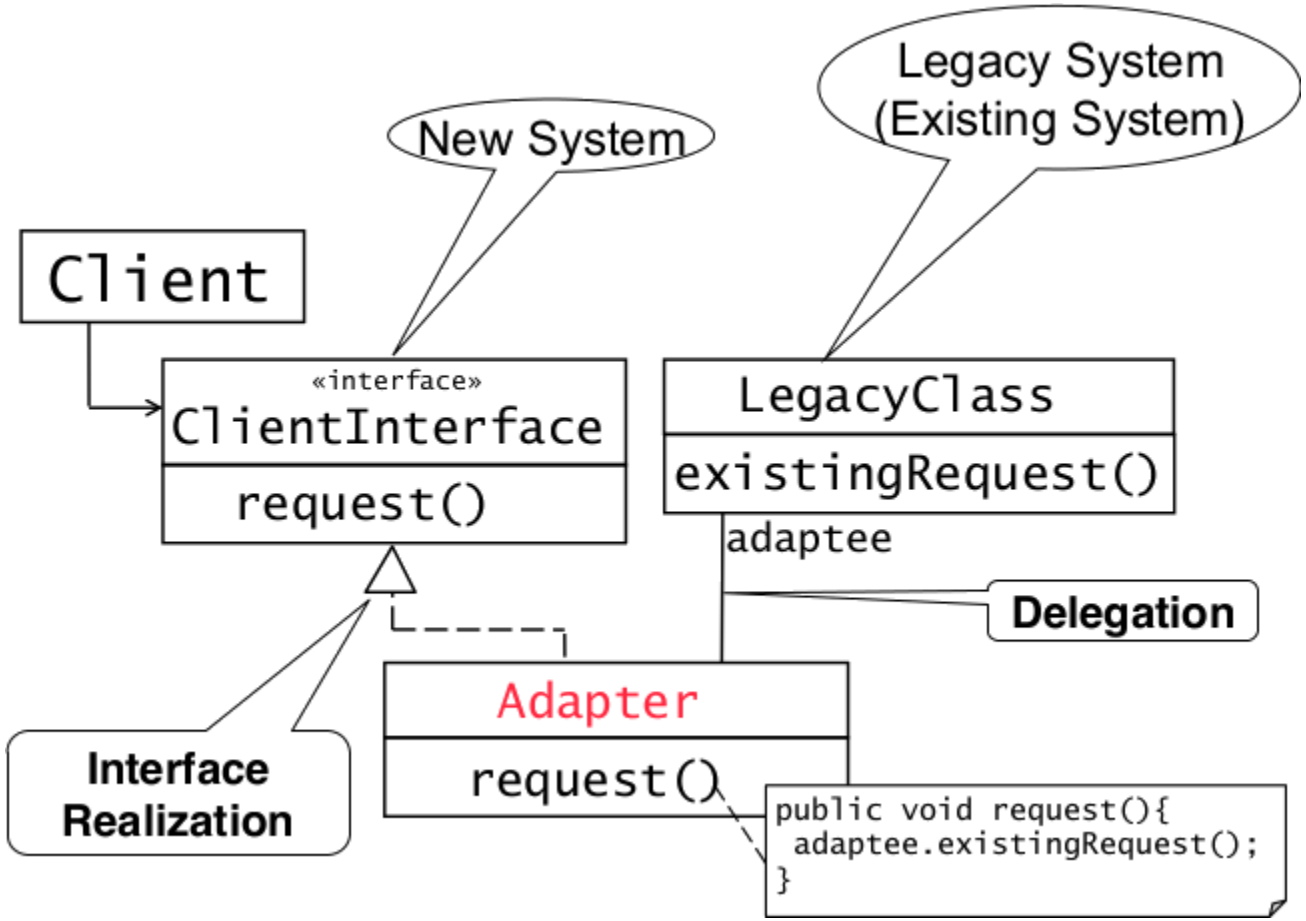
Behavioral Patterns

- Allow a choice between algorithms and the assignment of responsibilities to objects (Who does
- what?)
- Simplify complex control ows that are dicult to follow at runtime

Creational Patterns

- Allow a simplied view from complex instantiation processes
- Make the system independent from the way its objects are created, composed and represented

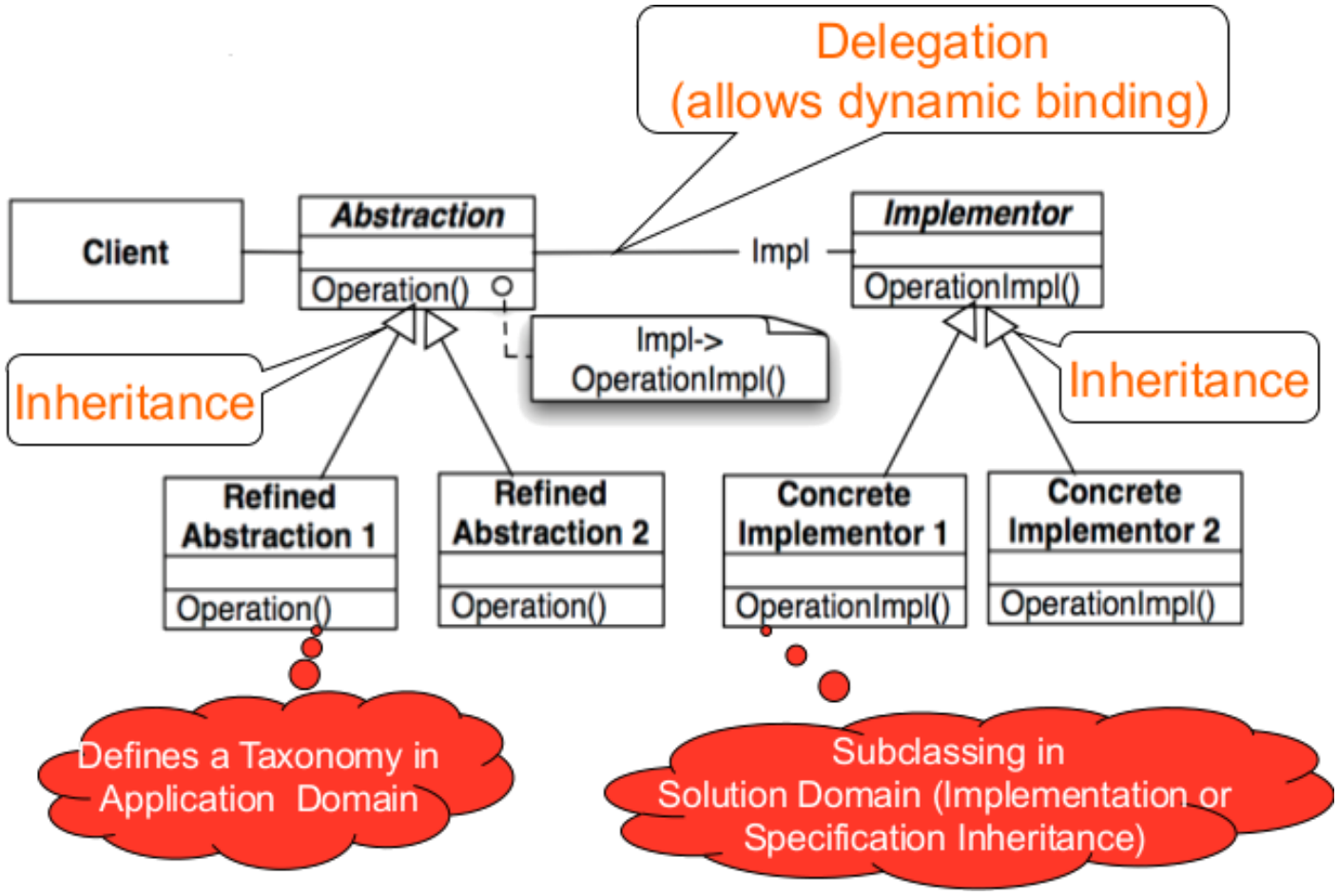
Adapter Pattern



Adapter Pattern

- Connects incompatible components
  - Reuse existing components
  - Convert an interface to another interface

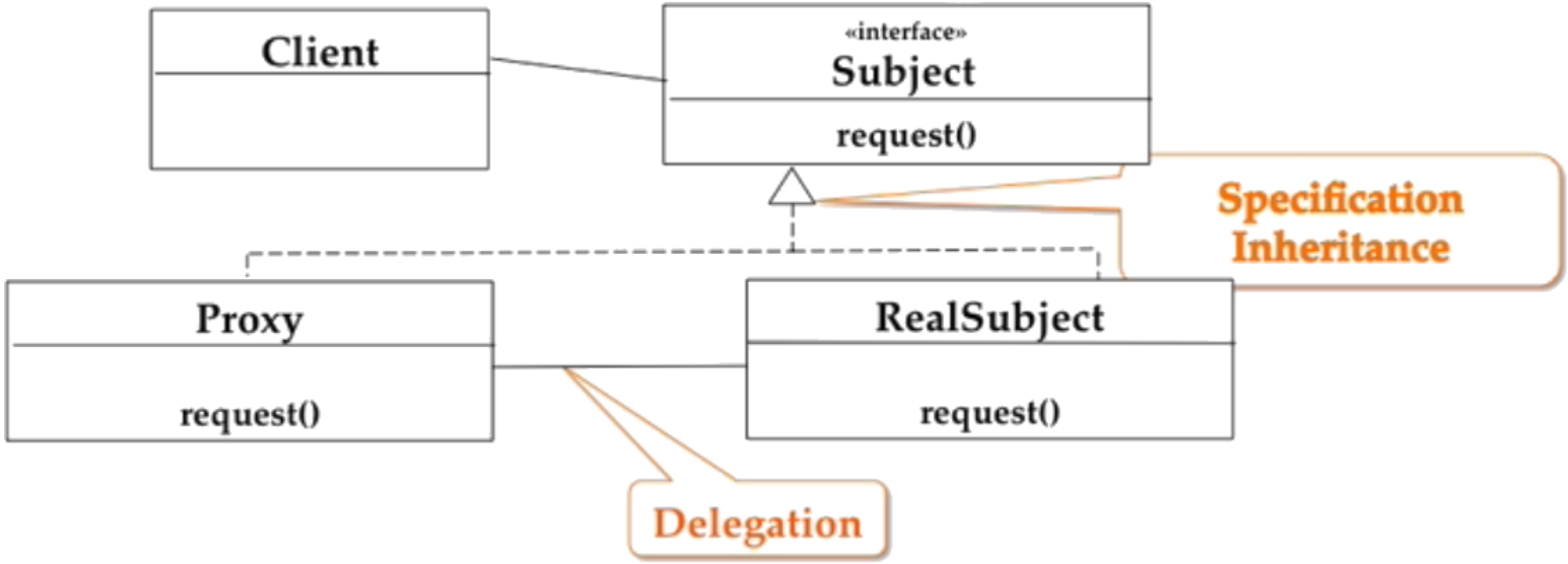
Bridge Pattern



Bridge Pattern

- Allows to delay assignment of an implementation of an interface from compile to runtime
- Degenerated Bridge is the same without taxonomy in application domain
- Used to test application with mocks (e.g. component not yet implemented..)
- Or support multiple vendors of a specific service (e.g. Database vendors..)
- Used up-front in a design to let abstractions and implementations vary independently

Proxy Pattern

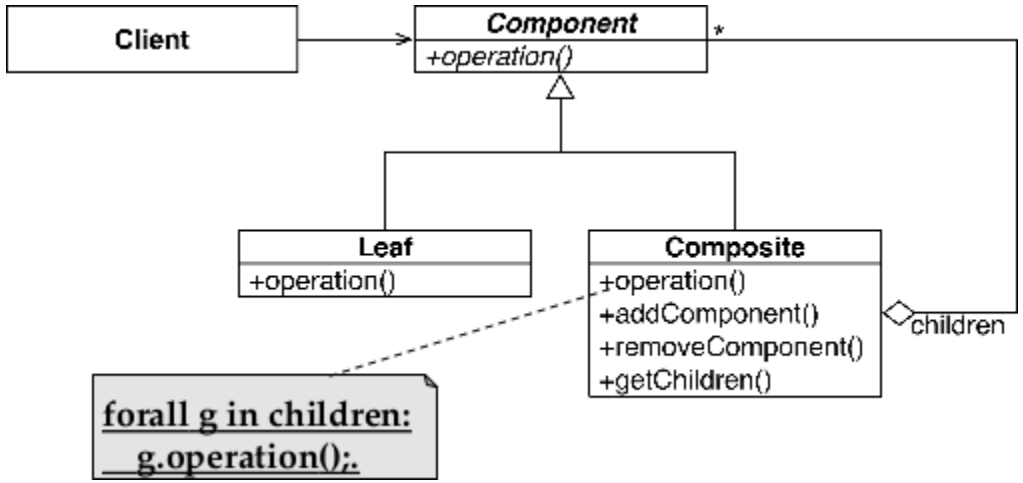


The client never calls *request()* in *RealSubject*, instead it always calls the method in *Proxy* which might delegate it to *RealSubject*

Proxy Pattern

- Allows to defer object creation and object initialization to the time you need the object
- Use cases:
  - Caching (Remote Proxy)
    - Local objects representative for object in different adress space
  - Substitute (Virtual Proxy)
    - Object is expensive to create/download, proxy acts as stand-in
  - Access Control (Protection Proxy/Firewall)
    - Proxy object provides access control to the real object

Composite Pattern

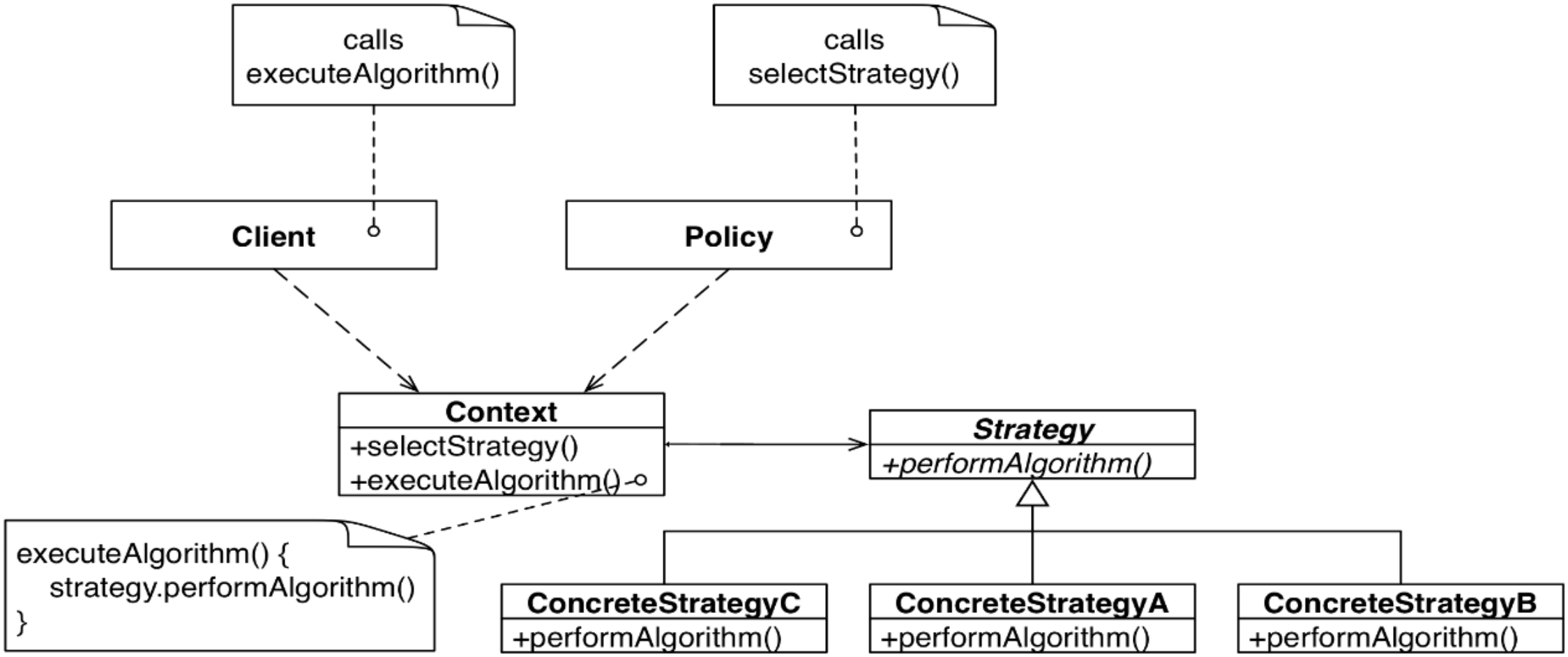


Tree structures which represent part-whole hierarchies with arbitrary depth and width.

Lets the client treat individual objects and groups uniformly

Behavioral Patterns

Strategy Pattern

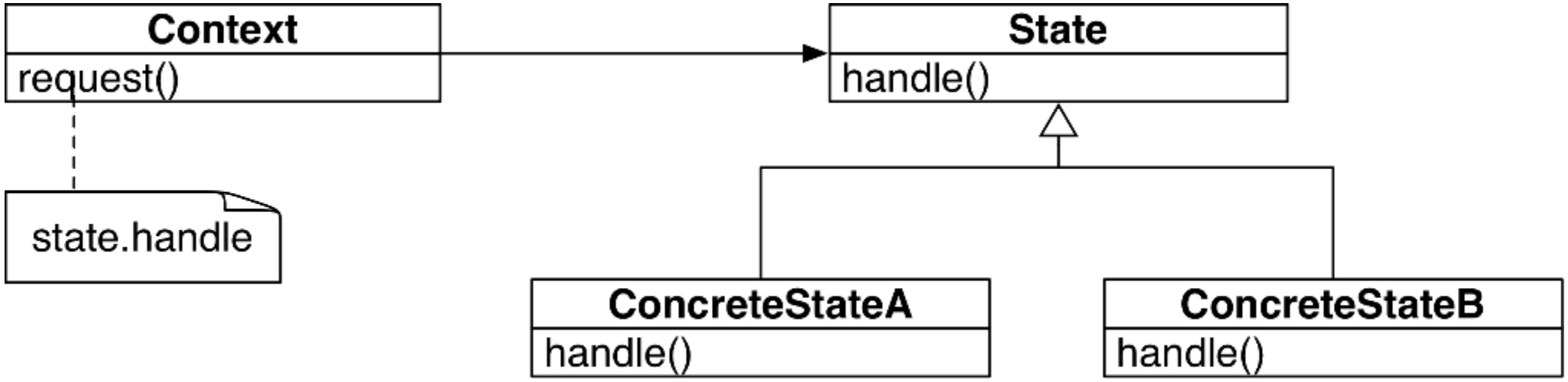


A strategy is chosen on **runtime** by the *Policy* class before the client calls *executeAlgorithm()*

Strategy Pattern

- Suited for situations where different algorithms are available for a problem
- In contrast to the bridge pattern (structural decisions) it choses the implementation to use at runtime and therefore alters the bahavior

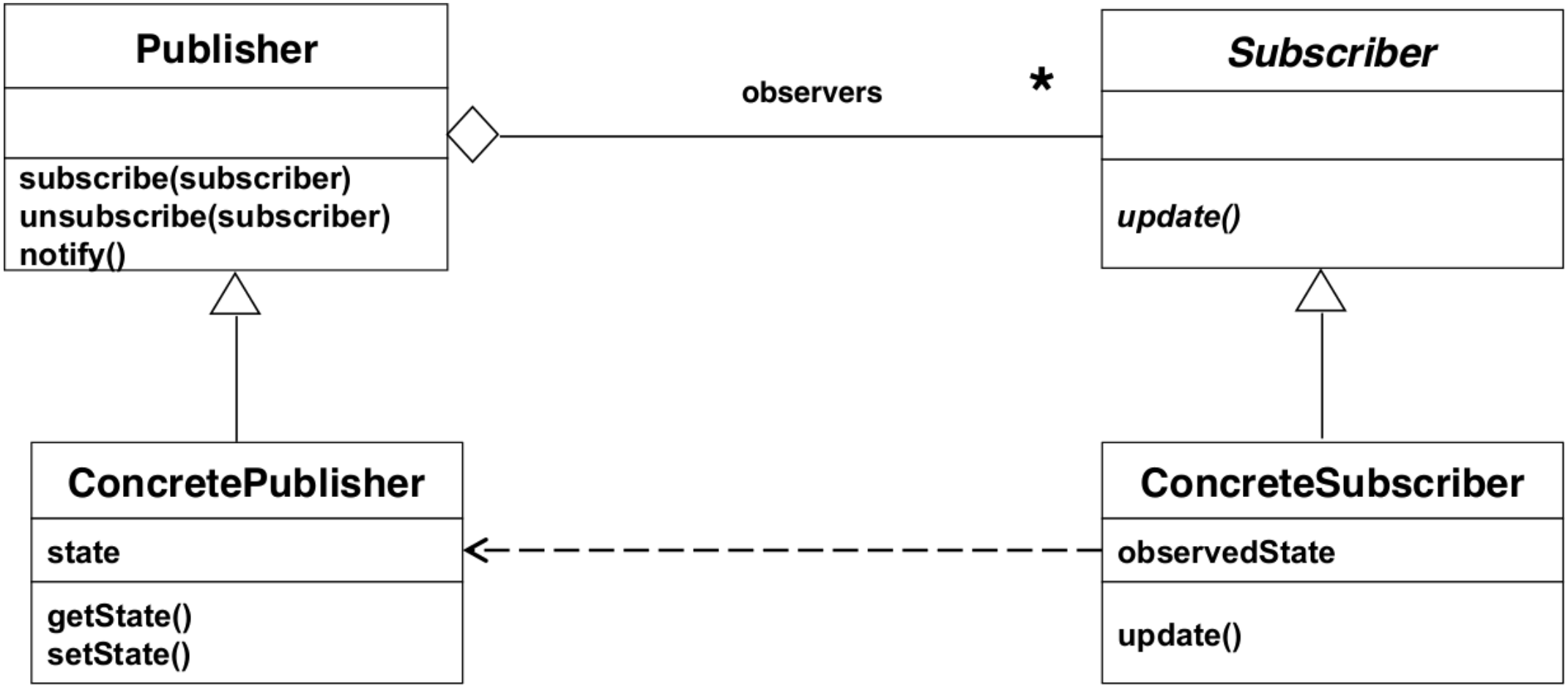
State Pattern



State Pattern

- Dependent on current state of a system, an action should do different things
- E.g. TCP open, close…
- It avoids many *if else* statements and is flexible to add more cases/states
- Also transitions between states are explicit
- In contrast to strategy pattern (handles different algorithms) it handles different states of an object

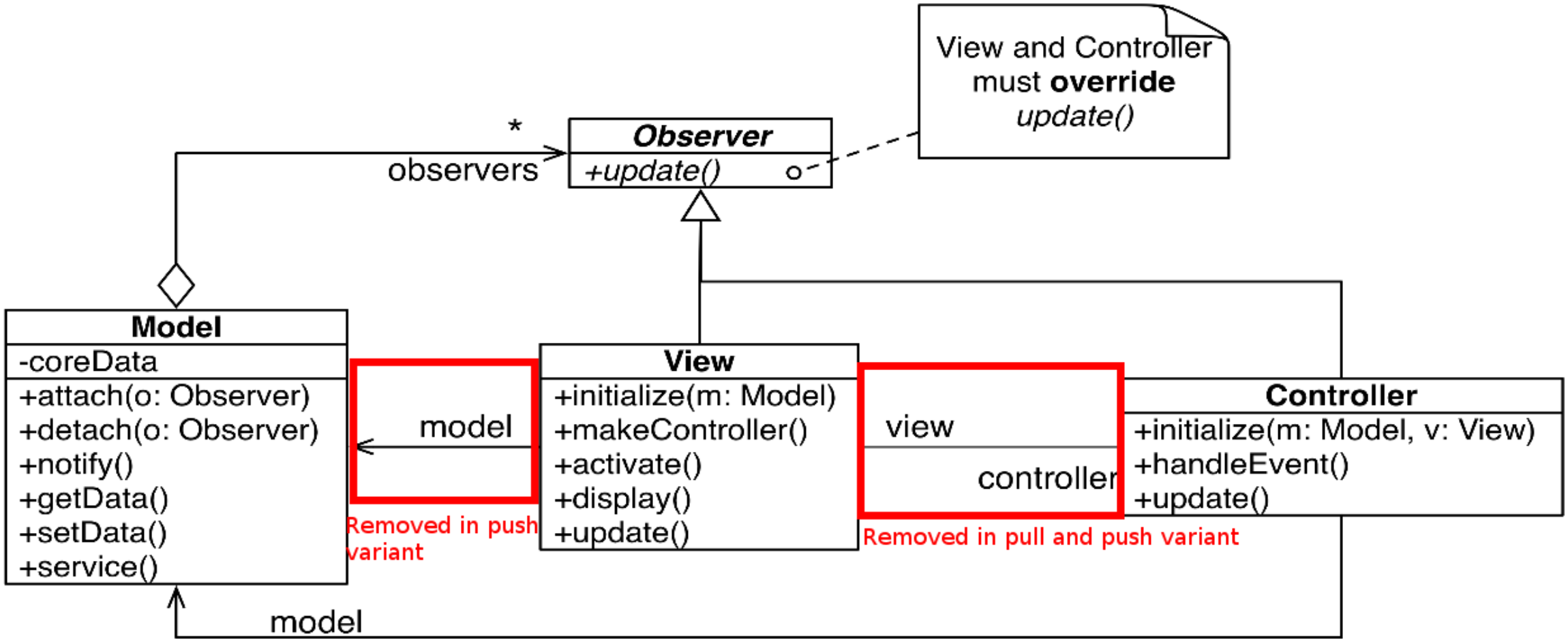
Observer Pattern



Observer Pattern

- Handles changes in a publisher class and notifies all subscribers about that change
- Maintains consistency
- Decouples an abstraction from ist views
- Three variants
  - **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 inquiries about the state of the publisher

Model View Controller Pattern

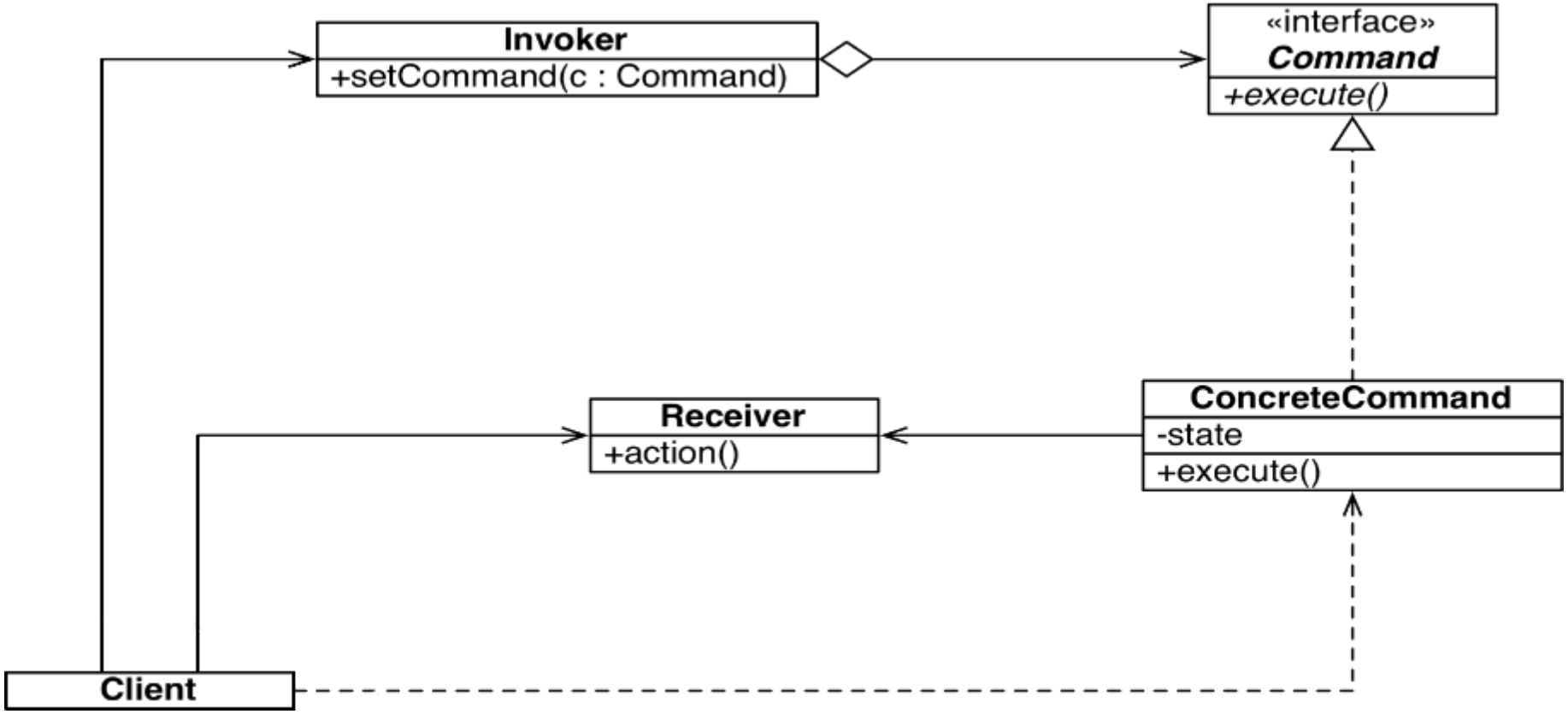


Model View Controller Pattern

- Decouples data access and data representation
- The *view* handles the data representation
- The *model handles* the data access
- The *controller* handles the communication between the other two

- **Pull variant:**
  - Connection between the controller and the view is removed
  - The view asks the model for the data explicitly
- **Push notification variant:**
  - Both connections between view and model and controller are removed
  - When a change in the model occurs, the view and controller are updated via the observer pattern

Command Pattern



Command Pattern

- Design user interfaces with multiple commands without multiple if-statements
- Used to make menus reusable across applications
- Reduces complexity by decoupling boundary objects (menu buttons) from control objects (concrete objects)
- Only these command objects can modify entity objects (the receivers)
- When user interface changed, only boundary objects need to be modified

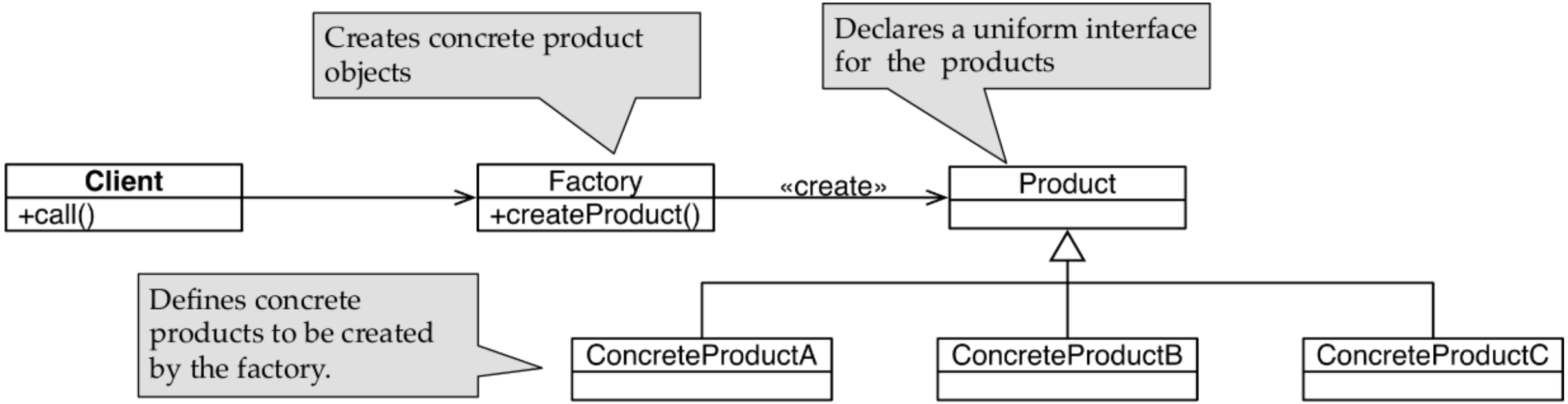
- **Command Applications:**
  - Command manager
  - Redu/Undo manager
  - Queue
  - Dispatcher

5 steps to realize a command pattern

1. Create the command interface
2. Create the ConcreteCommand classes
3. Create the Receiver
4. Create the Invoker
5. Create the Client

Creational Patterns

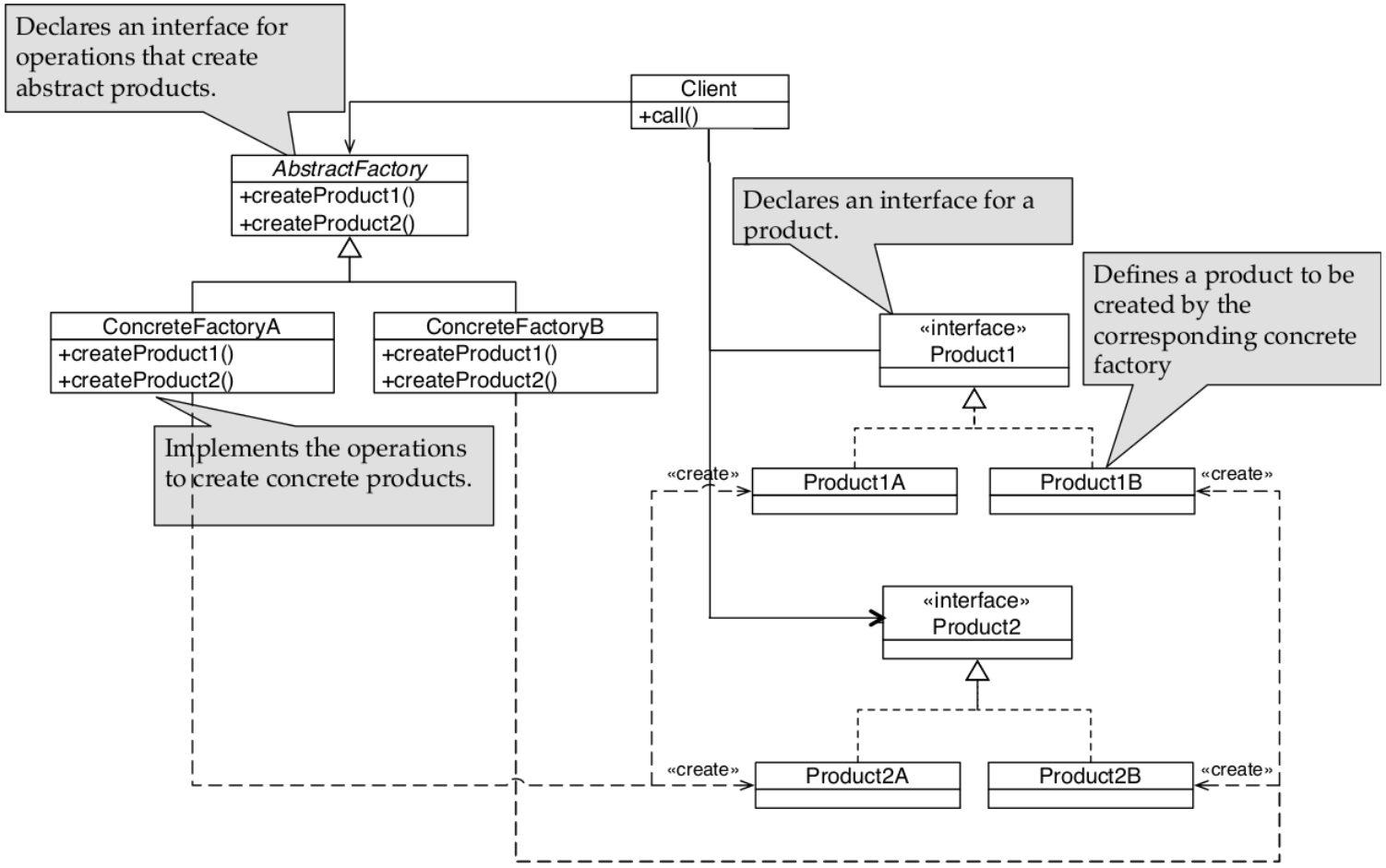
Factory Pattern



Factory Pattern

- Handles the instantiation of objects inheriting from one superclass depending on a keyword or value
- Acts as a delegate for the creation of products and allows the client to use a single interface to the products
- Due to polymorphism the client can use each of the concrete products uniformly

Abstract Factory Pattern



Abstract Factory Pattern

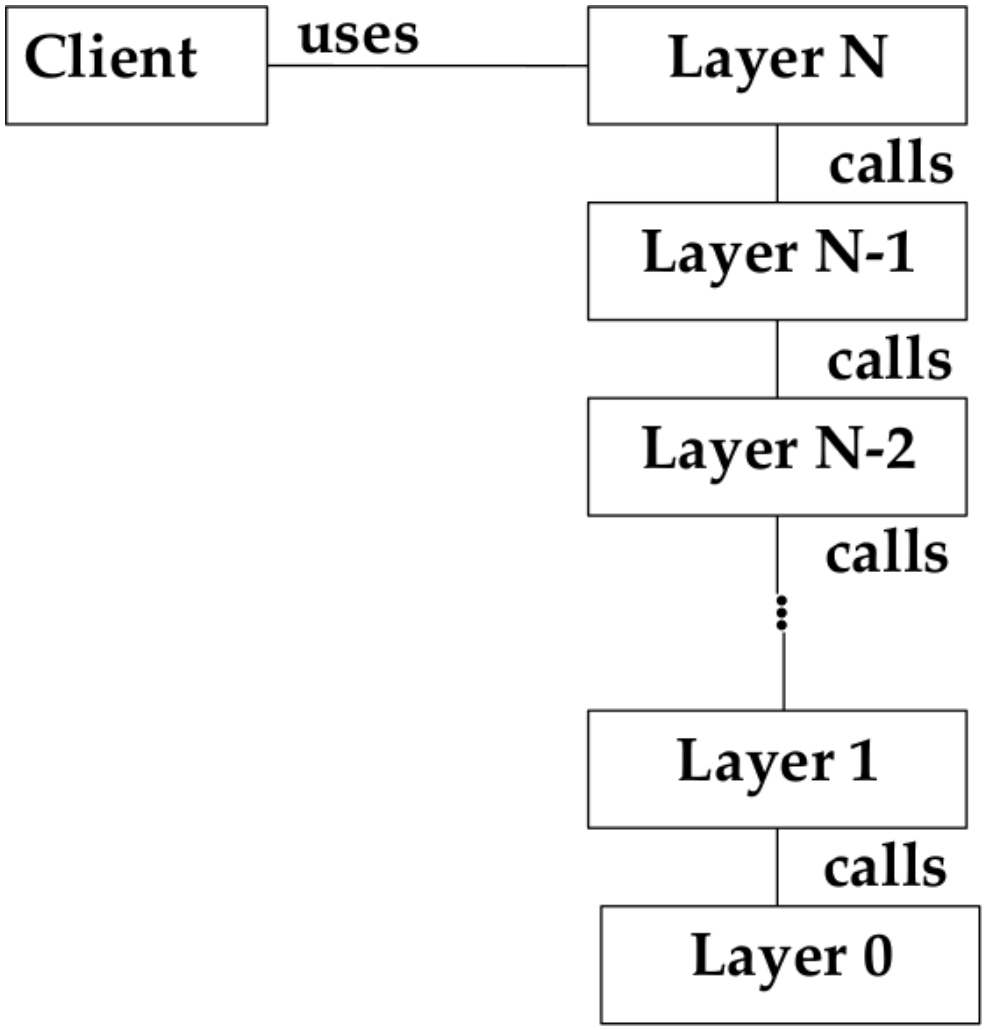
- Instantiate or initialize an object consisting of more subparts
- Every implementation of the abstract factory creates a set of components consisting of a variant of every part of the whole object

Comparisons

- **Adapter vs. Bridge**
  - Adapter (inheritance followed by delegation) handles incompabilities
  - Bridge (delegation followed by inheritance) differentiates between abstraction and implementation up-front
- **Bridge vs. Strategy**
  - Bridge used for structural decisions on system startup
  - Strategy handles behavioral decisions on runtime based on changing criteria
- **Strategy vs. State**
  - Strategy handles different algorithms at runtime
  - State handles different states of an object in the architecture

Architectural Patterns

Layer Pattern



Advantages of the layer pattern

- Reusability of layers, especially in a closed architecture
- Support für standardization
- Low coupling
- Improved testability

Disadvantages

- A local change in lower layer may require rework in higher layers
- Lower efficiency

Layer Pattern

Closed Architecture (Opaque Layering)

- Each layer can only call operations from the layer below

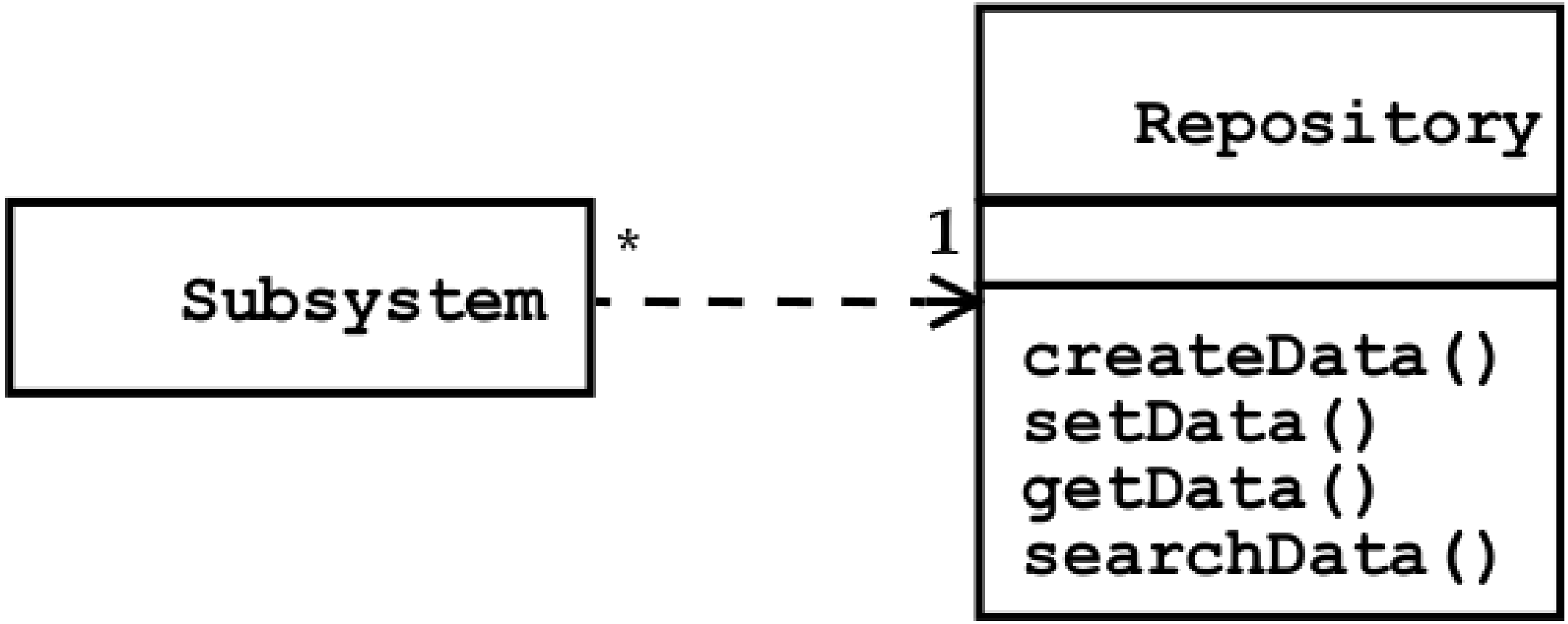
Open Architecture

- Each layer can call operations from any layer below

5 Steps to create a Layered Architecture

1. Identify subsystems, specify interface for each layer
2. Structure the individual layers (patterns)
3. Specify the communication protocol between adjacent layers (push/pull)
4. Decouple adjacent layers (return results only as parameters to upper layers/use callbacks)
5. Design an error-handling strategy (handling on lowest possible layer)

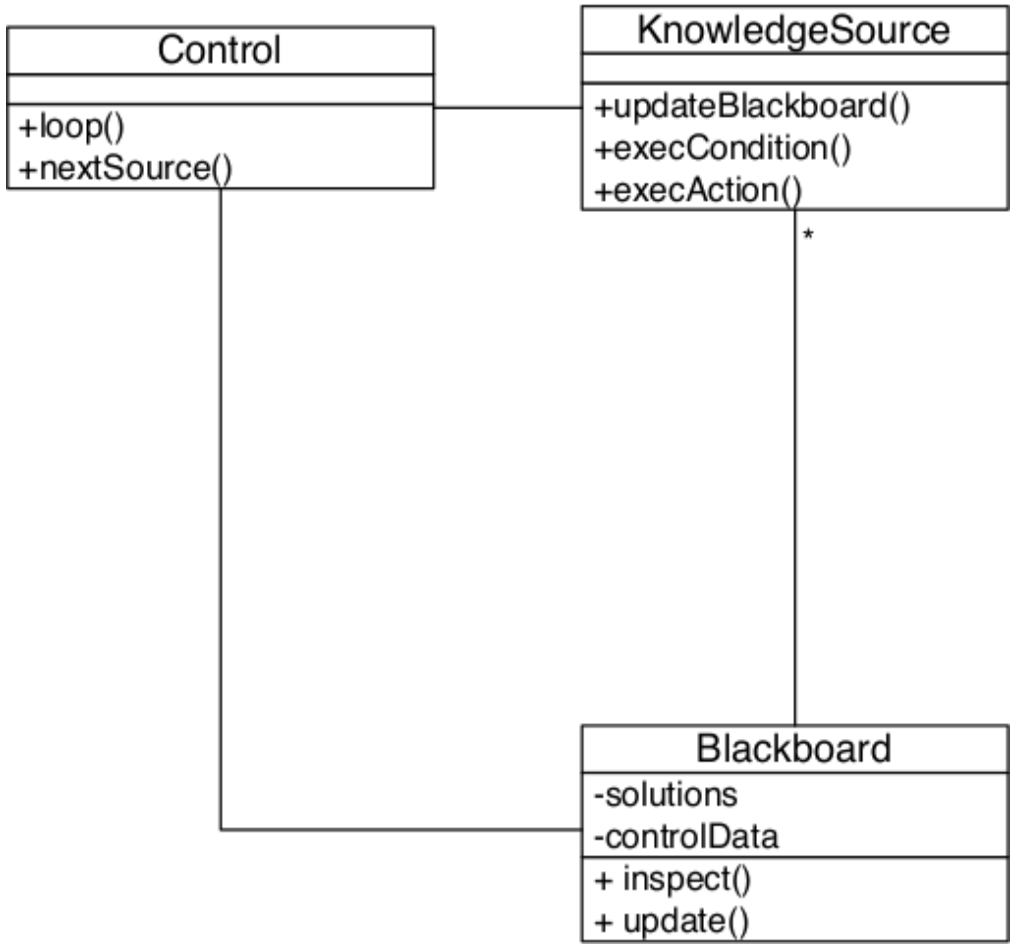
Repository Pattern



Repository Pattern

- Used to support a collection of independent programs that work cooperatively on a common datastructure (Repository)
- The subsystems exchange data via the repository and are therefore loosely coupled
- The control flow is not specified
- Control flow can be established by the subsystems themselves e.g. through locks and synchronization primitives

Blackboard Pattern



Advantages

- Problem solving support
- Changeability and maintainability
- Fault tolerance and robustness

Disadvantages

- Difficulty of testing
- No solution guaranteed
- Difficulty to establish a good control strategy
- High development effort

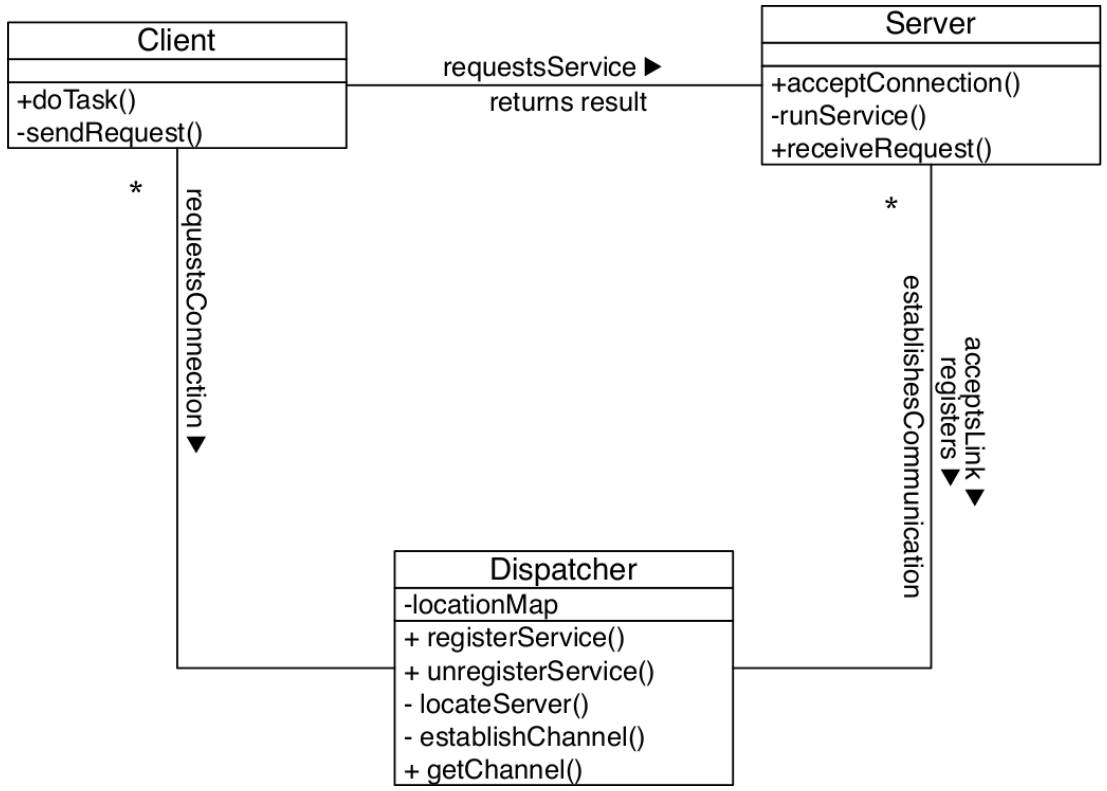
Blackboard Pattern

- Uses knowledge sources which communicate over a blackboard to solve a problem
- Blackboard is the repository for the problem
- Each knowledge source reads the content, processes it and generates new hypotheses
- A control instance governs the flow of problem-solving activities
- In general used when no algorithm for the problem is known

6 Steps to realize a Blackboard Pattern

1. Define the problem
2. Define the solution space
3. Identify the knowledge sources
4. Define the blackboard, identify needed representations
5. Define the control
6. Implement the knowledge sources

Client-Dispatcher-Server Pattern



- Decouples client from server by seperating establishment of a connection and communication over the channel
- Dispatcher allows the client to refer server by name instead of physical adress
- Allows server to dynamically change it‘s location

CD protocol:

- Spec. how a client must look for a server
- Deals with communication errors

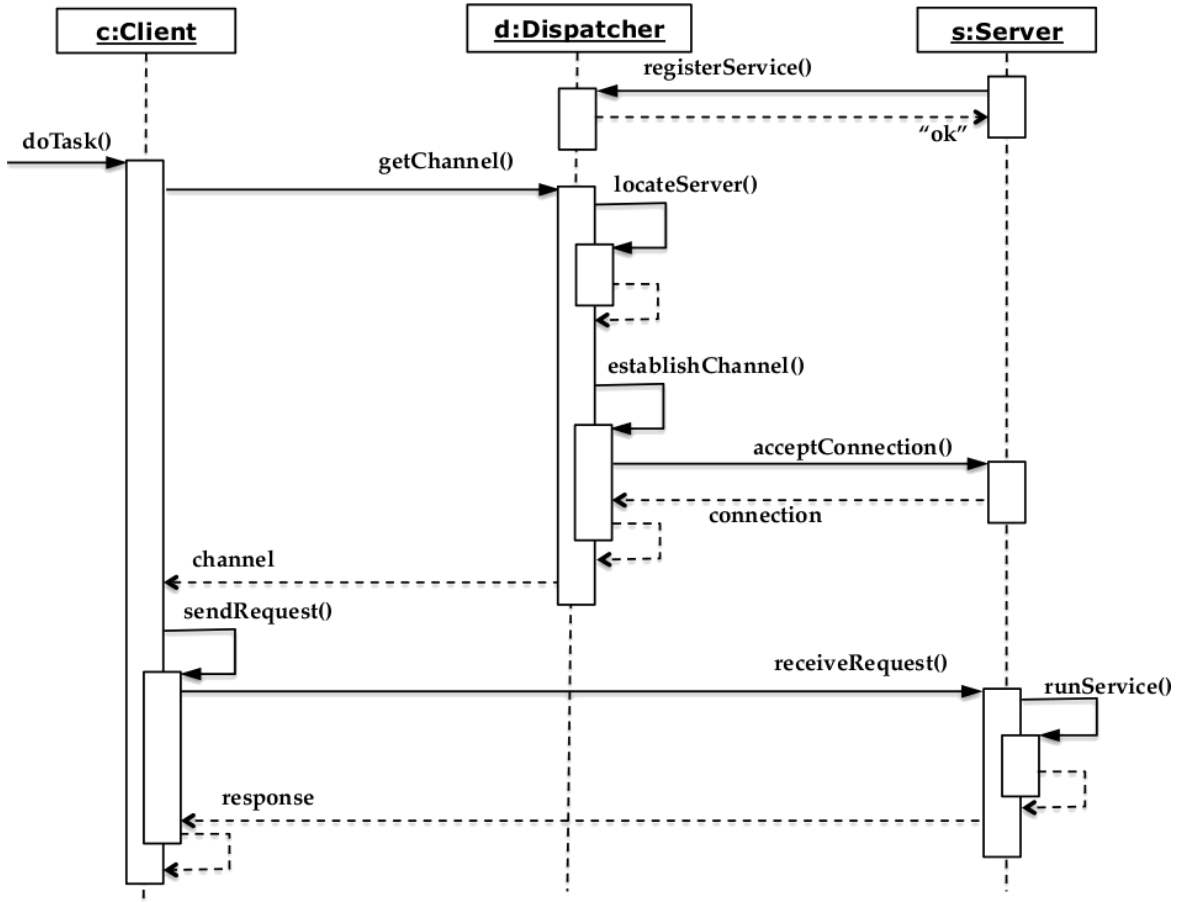
DS protocol:

- Specifies how servers register with dispatcher
- Determines activites needed to establish a communication channel between client and server

CS protocol:

- Specifies the communication between client and server

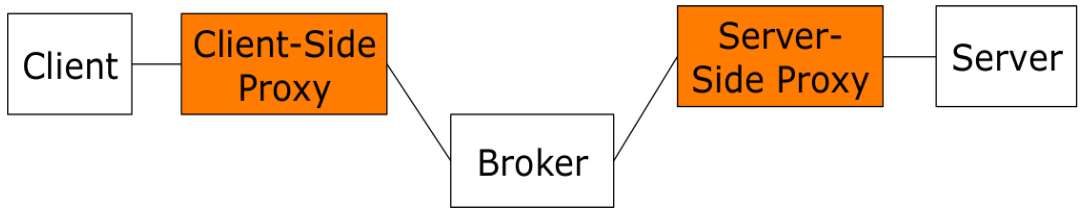
Client-Dispatcher-Server Pattern



6 Steps to implement Client-Server-Disp.

1. During system design, identify subsystems that act as clients and servers
2. Decide on the communication mechanism to be used for the protocols
3. Specify the protocols
4. Decide on a naming scheme for the dispatcher
5. Implement the dispatcher
6. Implement the client and server

Broker Pattern



- A broker coordinates the communication between heterogenous nodes

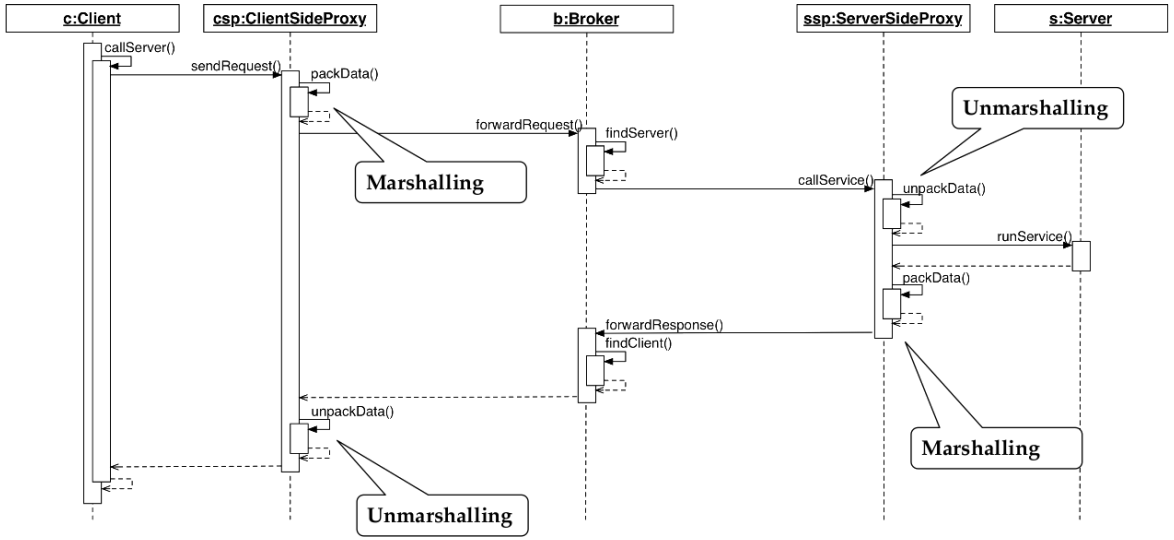
Nonfunctional requirements

- Low coupling
- Location Transparency
- Runtime Extensibility
- Platform Transparency

Client-Side Proxy

- Lets the remote object appear as local one, hides the inter-process communication details
- Provides(un-)marshalling

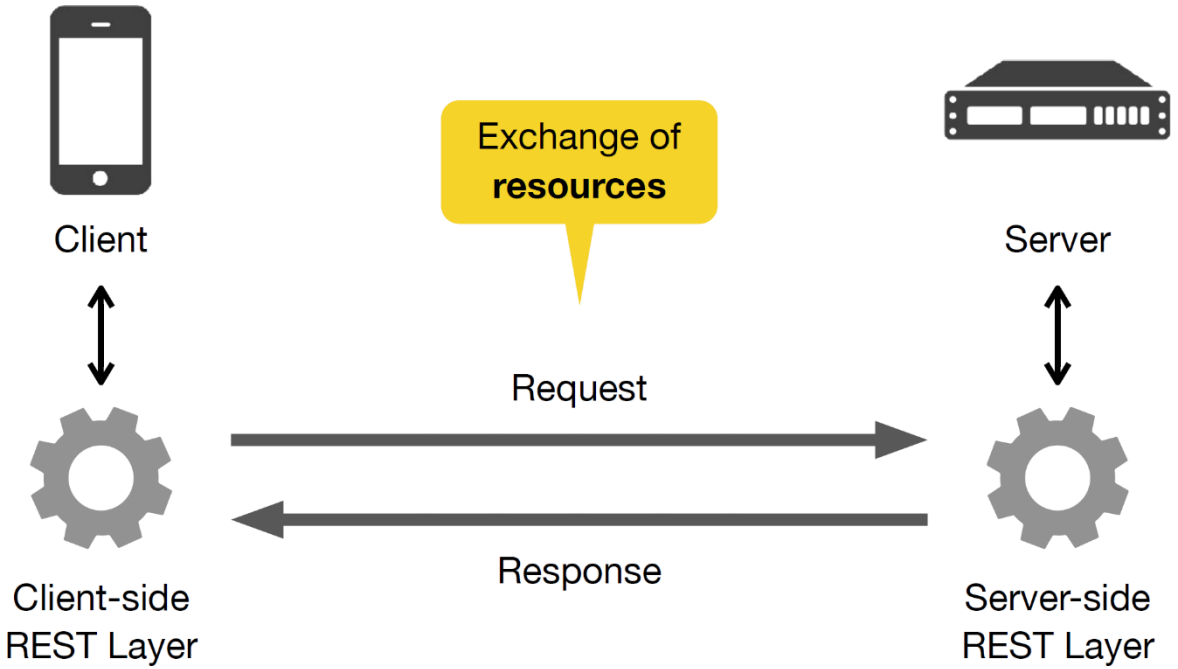
Broker Pattern



4 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

REST Pattern



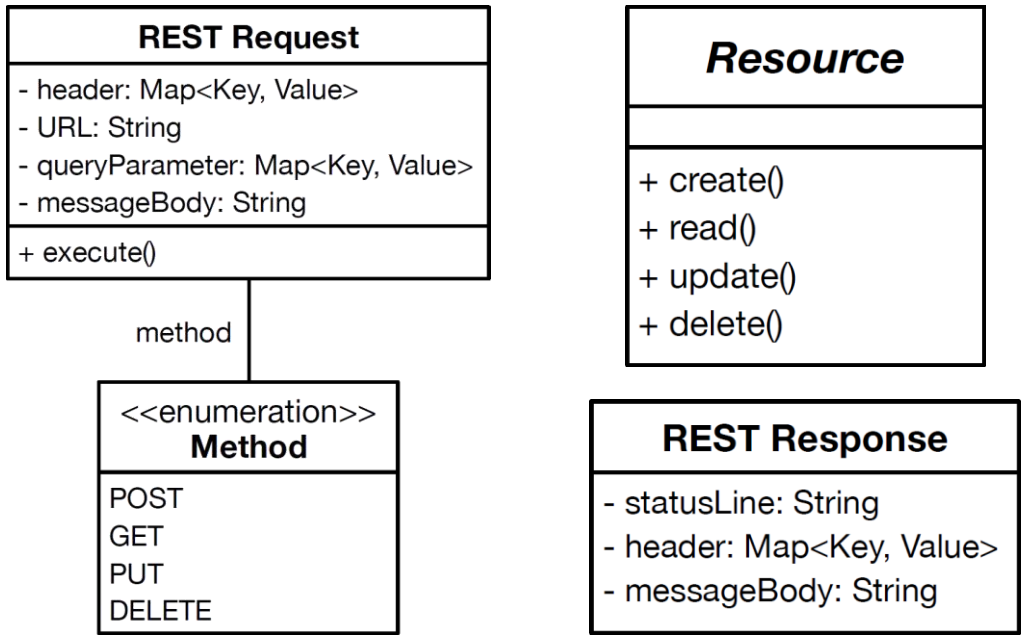
- Seperates between client and server
- Provides access to resources
- Server holds no session data → stateless

6 Elements of the REST architecture

- Client-Server
- Stateless
- Cachable
- Uniform Interface
- Layered System
- Code-On-Demand

Rest Pattern

Properties	POST	GET	PUT	DELETE
Usage	Create new resource	Retrieve existing resources	Create or update an existing resource with an identifier	Delete existing resources
Idempotent	❌	✅	✅	✅
Safe	❌	✅	❌	❌



REST Methods

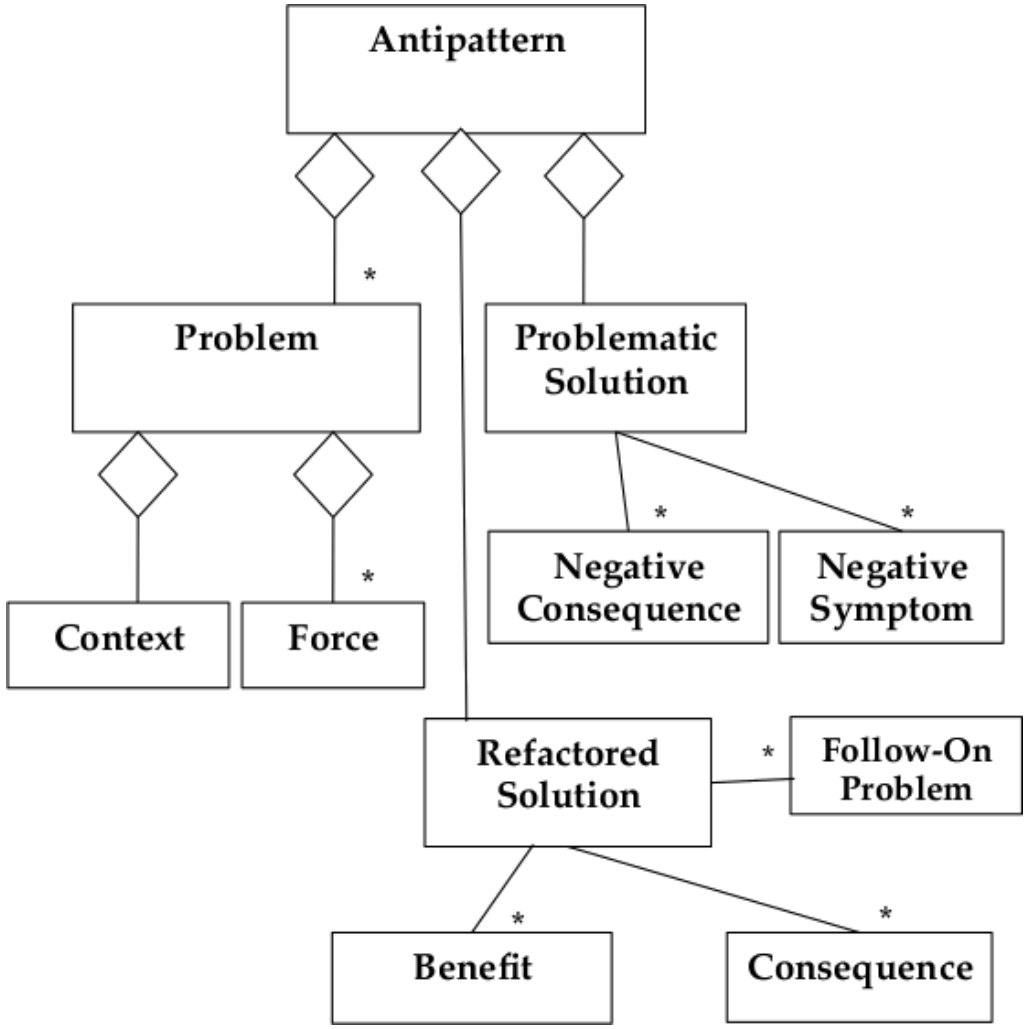
- **C**reate, **R**ead, **U**pdate, **D**elete
- POST, GET, PUT, DELETE

Request-Response

1. Requestor sends a message to a replier system
2. Replier system receives and processes the request
3. Replier returns a message in response

Antipatterns

Antipatterns - General



- Consists of a problem and two solutions
  1. Problematic solution
     - Commonly occuring solution that generates overwhelming negative consequences
  2. Refactored solution
     - How the problematic solution can be reengineered to avoid these negative consequences and lead to benefits again
- Patterns can evolve into antipatterns when changes occur

Antipatterns - General

7 Typical mistakes in software development

1. **Apathy**
   - Not caring about problem, unwillingness to attempt a solution
2. **Hastle**
   - Solution based on hasty decisions
3. **Narrow-mindedness**
   - The refusal to use solutions that are widely known
4. **Sloth**
   - Making poor decisions based on easy answers
5. **Avarice (excessive complexity)**
   - No use of abstractions, excessive modeling of details
6. **Ingorance**
   - Failure to seek understanding
7. **Pride**
   - Not willing to adopt anything from the outside

3 Types of Antipatterns

1. Developer antipattern
   - Software refactoring
   - Modification of source code
2. Architecture
   - Partitioning of subsystems and components
   - Platform independent defenition of interfaces
   - Connectivity of components
3. Management
   - Software project organization/management
   - Software process model
   - Human communication
   - Rational management

Functional Decomposition Antipattern

- Describes the decomposition of a system in terms of functions
- Insted of use cases and/or objects (object-oriented decomposition)
- So functions are hidden somewhere in the system where nobody might expect them

Recommended approach:

- First decompose the system in use cases, then objects

Functional Decomposition Antipattern

Also known as	• No OO
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 os often awkward and non-intuitive
Typical causes	• Wrong trained personal
Unbalanced forces	• Management of complexity • Change management
Refactored solution name	• Object-oriented reengineering
Refactored solution type	• Process

Golden Hammer Antipattern

General form	• Developer has high level of competence in a particular solution • Every new development effort is solved with this solution • Developer is unwiliing to learn and apply new approach
Symptoms and Consequences	• Identical tools used for many diverse products • System architecture depends on a particular application suite and a specific vendor tool set
Typical Causes	• Large investments in product for specific technologies maybe with exclusive features • Reliance on proprietary product features that are not available from other vendors
Variants	• Obsessive use of favorite software concept or GoF design pattern
Known Exceptions	• Product is part of vendor suite that provides for all needs
Refactored Solution	• Projet organization develops commitment to explore new technologies • Software developers keep up to date on technology trends • Management adopts commitment to open systems and architectures

Lava Flow

Also known as	• Dead code
General form	• Lava like flows of previous development hardened into basalt like 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
Known Exceptions	• Small-scale, papidly development
Refactored Solution	• Architecture-centric Management • Avoid architecture changes during active development

Blob Antipattern

Also known as	• God class
General form	• Majority of responsibilities are in one complex controller and are associated with simple data classes
Symptoms and Consequences	• Huge class with many unrelated attributes and operations enapsulated • Usually too complex to reuse and testing
Typical Causes	• Lack of (object-oriented) architecture • Too limited intervention in iterative projects
Known Exceptions	• Wrapping of legacy systems
Refactored Solution	• Identify or categorize related attributes and operations • Move them into classes they belong to (Source code refactoring) • Remove redundant, indirect associations

Spaghetti Code Antipattern

General form	• Software with very little structure where object methods are invoked in a single, multistage process flow
Symptoms and Consequences	• Methods are process oriented, objects are named as processes • Execution flow is dictated by the class implementation of objects instead by the users of that class • No inheritance, no polymorphism • Source code difficult to reuse • Point of diminishing returns: Software maintenance effort higher than a compete reengineering effort
Typical Causes	• No design prior to implementation • Inexperience with object-oriented design
Refactored Solution	• Software Refactoring, Code Cleanup • Incremental refactoring

Vendor Lock-In Antipattern

General form	• A software project adopts product technology and becomes completely dependent of the vendors implementation
Symptoms and Consequences	• Maintainance cycle driven by the product update cycle • Promised product features are delayed or never delivered, subsequently causing failure to deliver application updates • Application programming requires in-depth product knowledge
Typical Causes	• Product is selected because of marketing instead of technical inspection • Product varies from published open system standards beacause there is no effective conformance process for the standard
Known Exceptions	• Single vendors code is the majority of the code needed for an application
Refactored Solution	• Isolation layer (Closed architecture) • Separation of infrastructure knowledge from application knowledge • Level of abstraction between application software and lower-level infrastructure • Adapter design pattern

Analysis Paralysis Antipattern

General form	• Goal is to achieve perfection and completeness of the analysis phase • Very detailed models
Symptoms and Consequences	• Analysis cost exceeds expectations without a predictable end point • Analysis documents no longer make sense to domain experts
Typical Causes	• Management assumes waterfall progression of phases • Analysis goals are not well defined
Refactored Solution	• Vertical prototyping • Scenario based design • Sprint based development (Agile methods) • Incremental, iterative, adaptive development

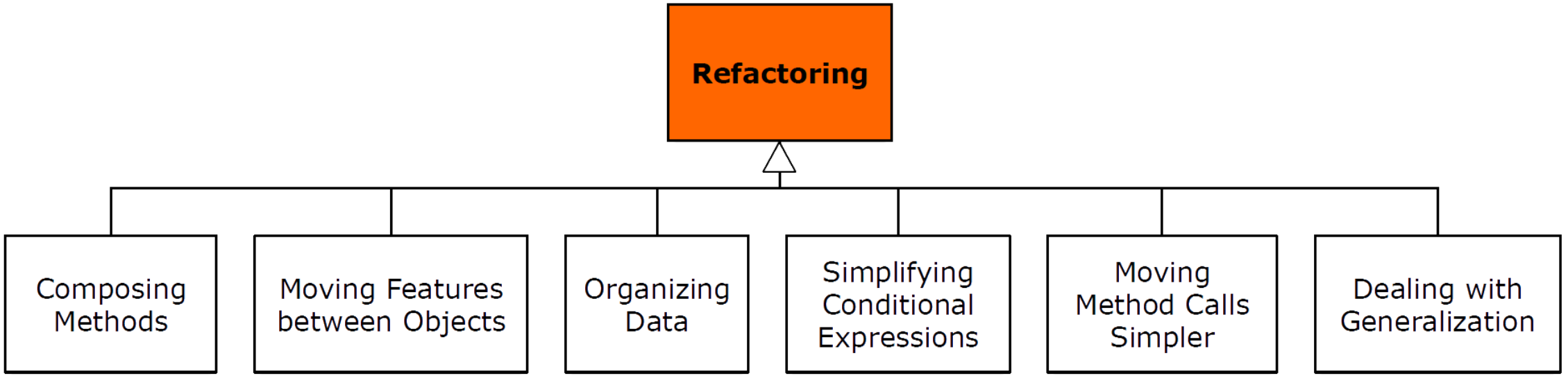
Code smells and Refactoring

Code smells

- Symptoms in the source code of a program that possibly indicate bigger problems
- It is a heuristic indicaton when to refactor and what specific technique to use

Method too long	Extract method
Duplicated code	Extract/pull up method, extract class
Class too large	Extract superclass
Parameter list too long	Replace parameter with explicit method, introduce parameter object
Feature envy (class uses methods of another class excessively)	Move class
Lazy class (no interesting behavior)	Turn class into attribute
Speculative generality (excessive use on inheritance)	Collapse inheritance tree
Refudes bequest (subclass reusing behavior of superclass, but no interface supported)	Replace inheritance with delegation

Refactoring



Replace Inheritance with Delegation

➢**Dealing with Generalization**

- A subclass is only using parts of the superclass interface or does not want to inherit data
- Results in source code that does one thing when your intention something else

✓Replace inheritance with delegation

✓This makes clear that only parts of the delegated class are used

4 Steps for dealing with generalization

1. Create a field in the subclass that refers to an instance of the superclass
2. In the subclass call public methods of the superclass via this field
3. Break the inheritance relationship by removing the extends declaration from the subclass definition
4. Create delegating methods in the subclass for those superclass methods that you want to use in the subclass

Refactorings

Extract Method (Composing Methods)

- Shortens methods by extracting methods
- Related parts of the methods body are extracted and moved to seperate methods

Extract Class (Moving Features between Objects)

- If a class contains an implicit abstraction that is not explicitly modeled
- Attributes can be summarized in a seperate class and then used in the original class

Replace Data Value with Object (Organizing Data)

- A simple attribute of a class gets more versatile
- The value can be replaced by an object of a newly introduced class

Replace Conditional with Polymorphism

➢**Simplifying Conditional Expressions**

- A simple if-then-else may mutate to a switch-case over time
- As possible cases grow code may become confusing and hard to maintain

✓Introduce polymorphism

✓Turn object on which decision is made into abstract class

✓Subtypes encapsulate different cases

✓E.g. Command pattern

Replace Error Codes with Exceptions

➢**Making Method Calls Simpler**

- When error codes used as return values
- Developers using methods have to resolve error code
- Look up their meaning

✓Use exceptions to transfer information on the error to the caller of a method

✓Moves responsibilities to resolve the error to the caller

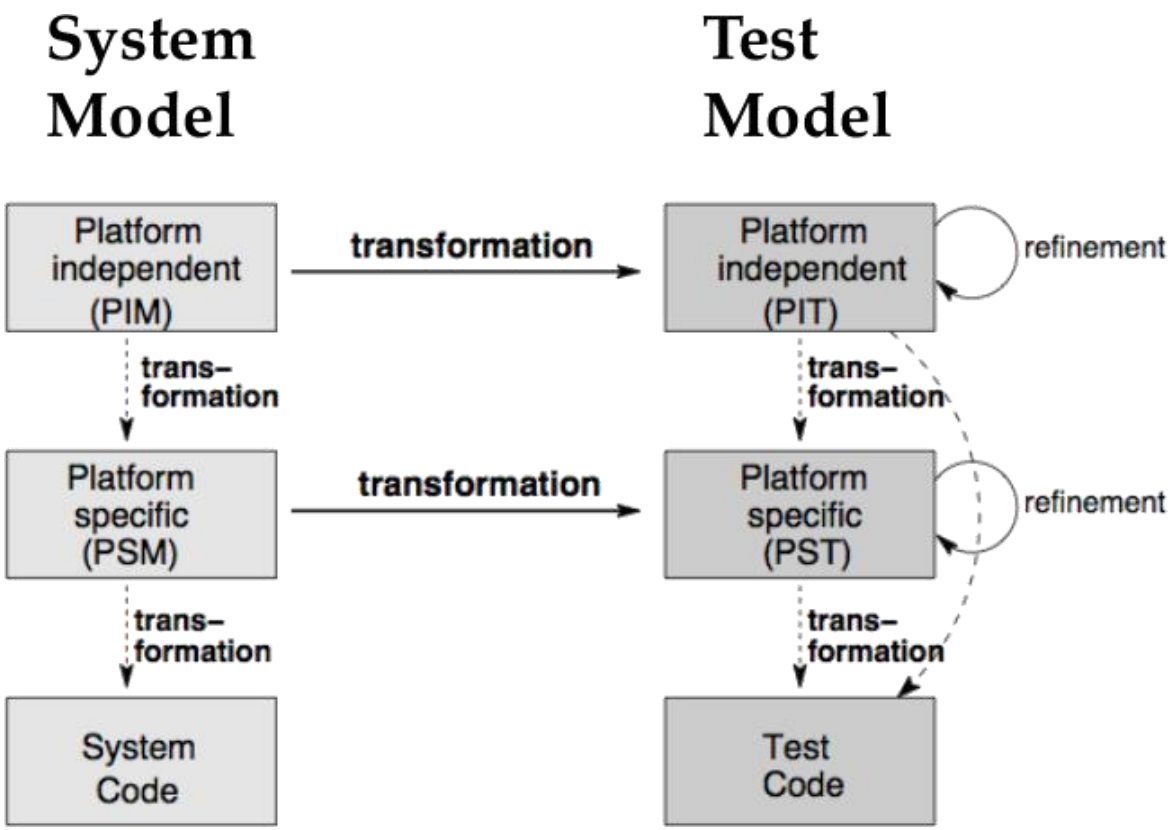
Testing Patterns

- Test is successful if it generates a failure (Goal is falsification of a model)
- Test is successful if it does not generate a failure (Commonly used)

Test Model

Test Cases/Tests	• Description of the testing activities • Derived from use cases
Test Driver	• Programs executing tests
Input Data	• Needed for the test
Oracle	• Compares expected output with actual output of the test
Test Harness/Testing Framework	• Software components/framework running tests under varying conditions and monitoring behavior

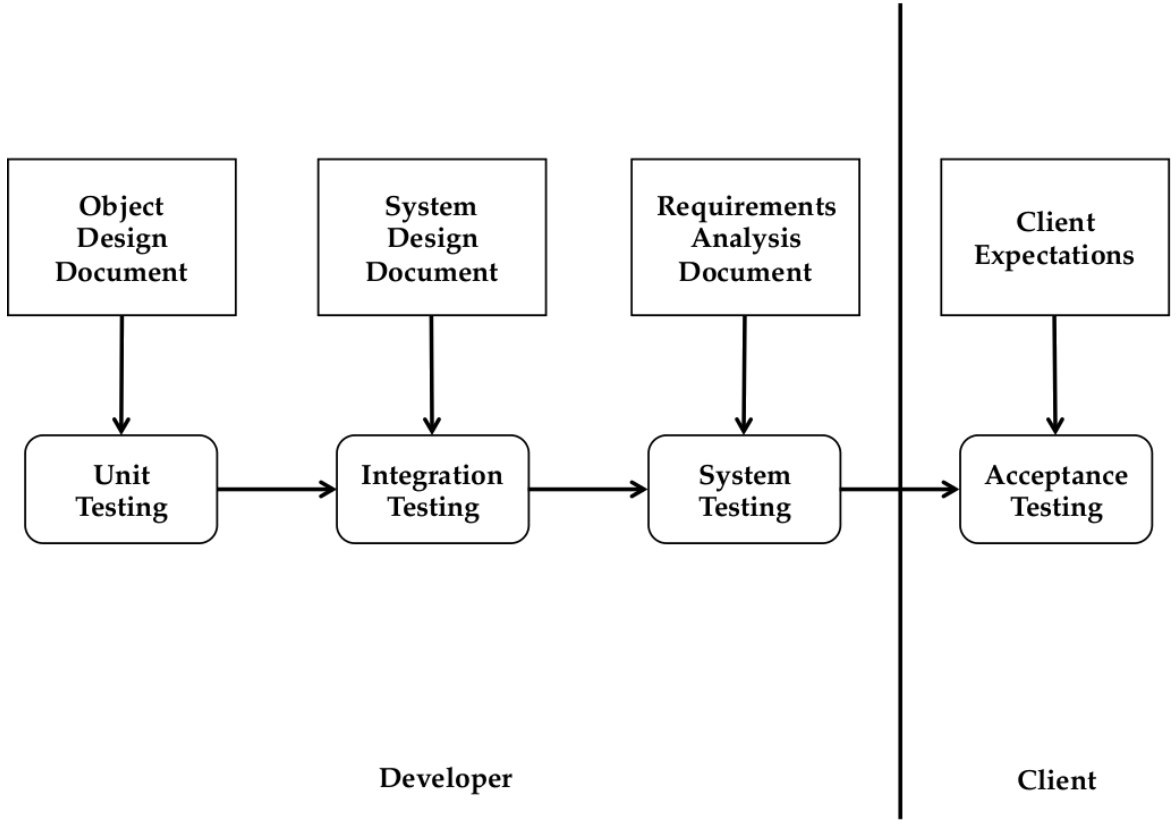
Model-Based Testing



- System model is used for generation of the test model
- Increses the effectiveness of testing, as costs are reduced and maintenance is easier
- Analysis and design models are reused
- The platform independent system model can be transformed into the platform independent test model
- The platform specific test model can be derived from the platform specific system model or the platform independent test model

✓ Enables early integration of testing into system development process (test driven development)

Testing Activities



Tests after every change are called **Regression Tests**

Unit Testing/Module Testing

- Developers test individual components, confirms correct coding of component or subsystem

Integration Testing

- Developers test groups of subsystems, confirm interface specifications of subsytems

System Testing

- Developers test the entire system, checks if system requirements are met

Acceptance Testing

- Client evaluates the system by executing typical use cases, checks requirements

JUnit Testing

@Test public void foo()	foo is a test
@Before public void bar()	bar is executed before every test
@After public void foobar()	Any test method must finish with call to foobar
@BeforeClass public void foofoo()	foofoo is executed before the start of all tests
@AfterClass public void blabla()	blabla is executed after all tests have nished
@Ignore(String s)	Ignores the prexed method and prints s instead
@Test(expected=IllegalArgumentException)	Tests if the test method throws the named exception
@Test(timeout=100)	Test fails, if it takes longer than 100 milliseconds

JUnit Testing

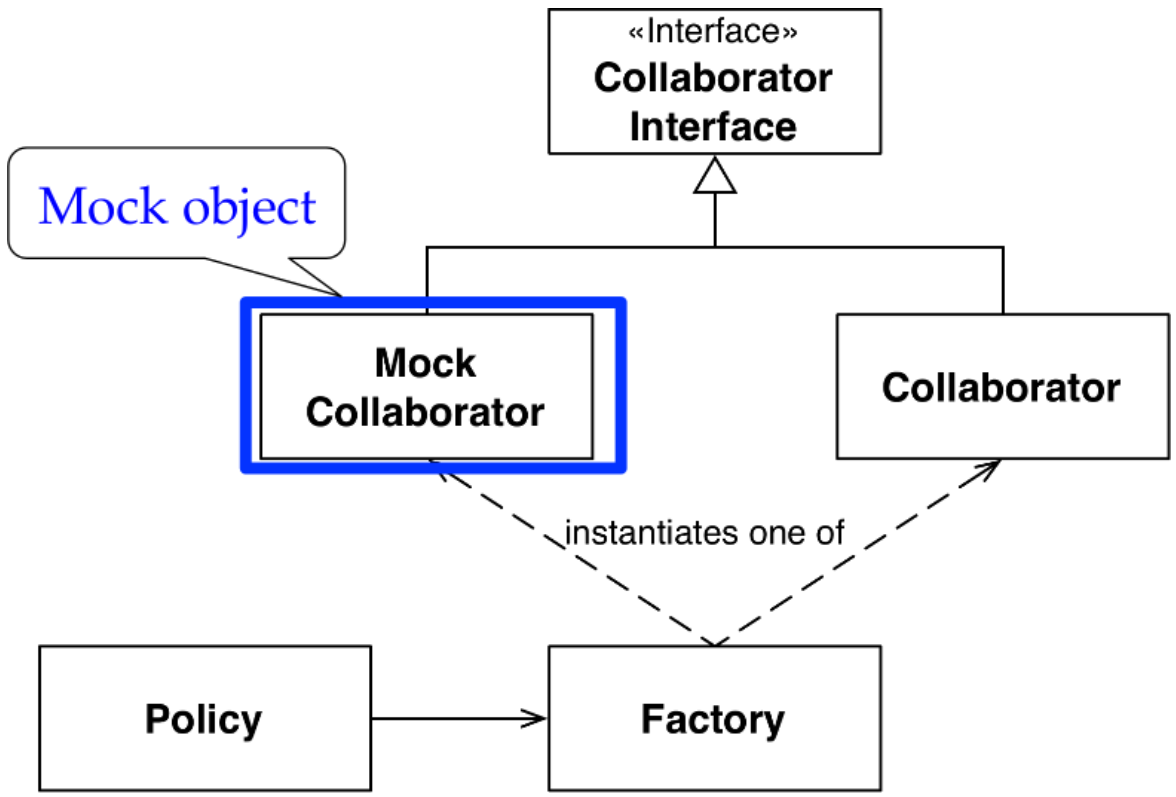
Assertions

- assertTrue(predicate)
- assertFalse(predicate)
- fail(String) lets the method fail, used in code which should be unreachable
- assertsEquals([String message], expected, actual)
- assertsEquals([String message], expected, actual, tolerance) used for float and double
- assertNull([message], object) prints message if object is null
- assertNotNull([message], object)
- assertSame([String], expected, actual) expected == actual (not equals!)
- assertNotSame([String], expected, actual)

Object-Oriented Model-Based Testing

Dummy object	• Used to fill parameter holes, never actually used
Fake object	• Functional class, that has not yet the actual funcionality of the real class
Stub	• Returns always the same values
Mock Object	• Imitates real behavior of an object • Requires a good architecture to let the mock object class inherit from the desired interface

Mock-Object Pattern



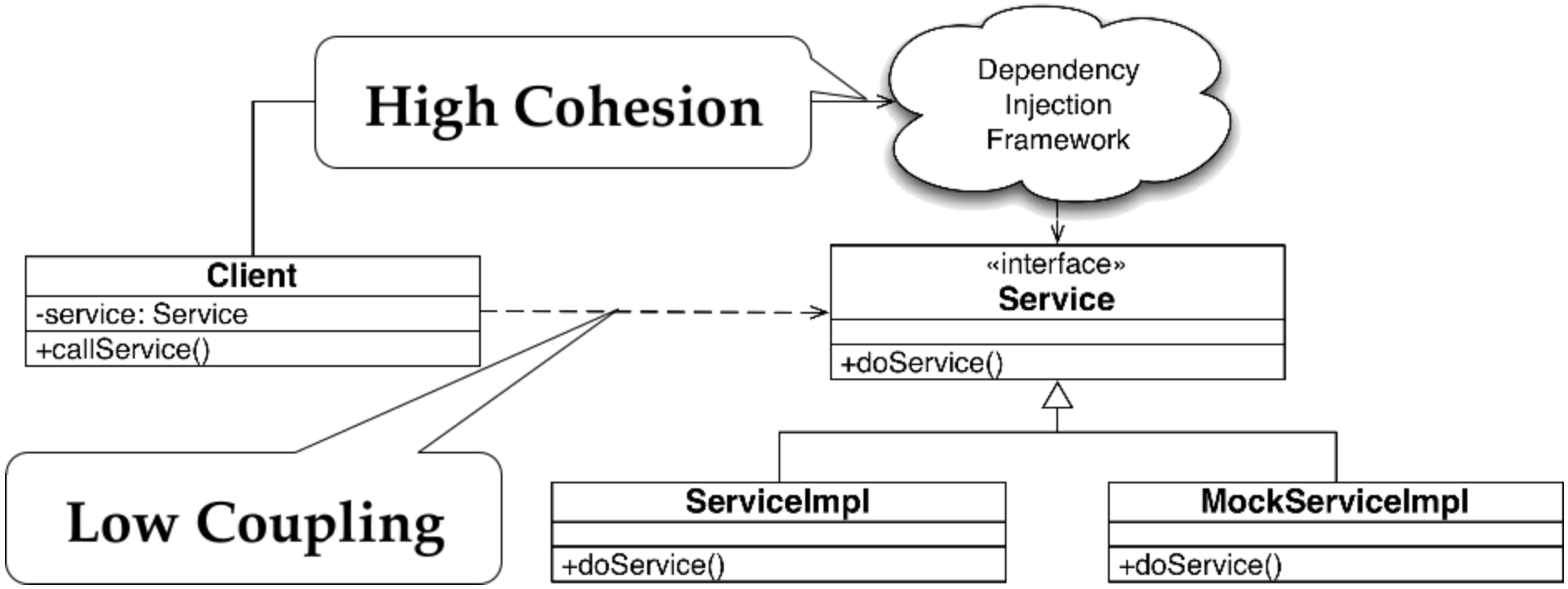
- Unit tests with nondeterministic behavior
- Object is difficult to set up
- Specific behavior is hard to trigger
- Slow methods
- Object has an user interface or is the user
- The real object is not testable

Mock-Object Pattern

How to use Easy Mock

1. Instantiate mock object: mock = createMock(foo.class)
2. Specify the expected behavior
   - Void methods are called as in Java
   - Methods with return values use expect() to specify the return value and andReturn() to specify the expected value
   - times() defines how often the method can be called
3. Use replay(mock) to make the mock object available
4. Invoke methods in the SUT
5. Make sure the SUT used the mock object as specified with verify(mock)

Dependency Injection Pattern



Avoid high coupling between test classes an SUT

Dependency Injection Pattern

Google Guice Framework

1. Place @Inject annotation (constructors, methods, fields)
2. Create a Module to define binding

```
configure(){
    Bind(Service.class).to(ServiceImpl.class)
}
```

3. Instantiate an injector to tell which module to use

```
Guice.createInjector(new ProductionModule())
```

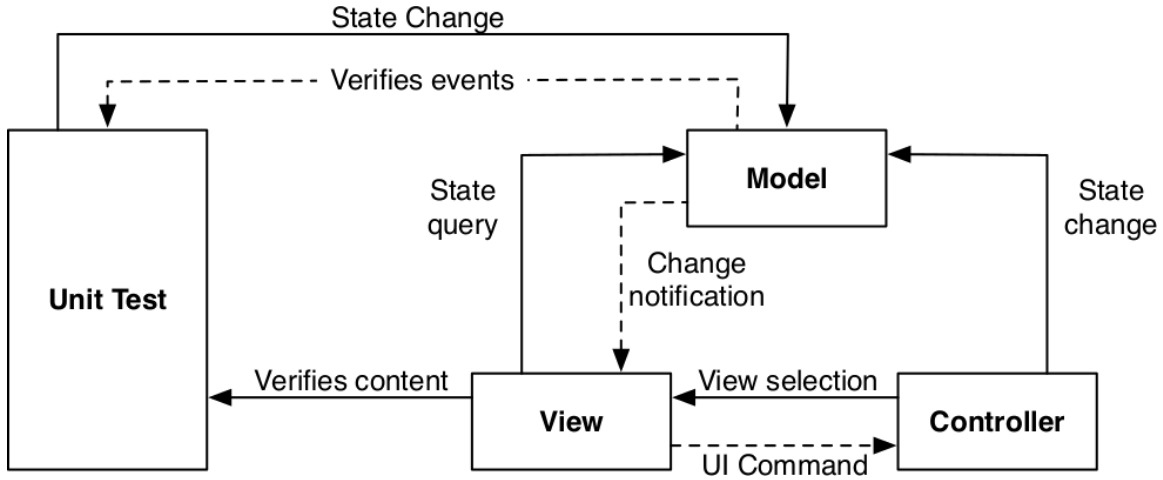
4. Instantiate an instance of the class needing injection

```
Injector.getInstance(Service.class)
```

Four-Stage Testing Pattern

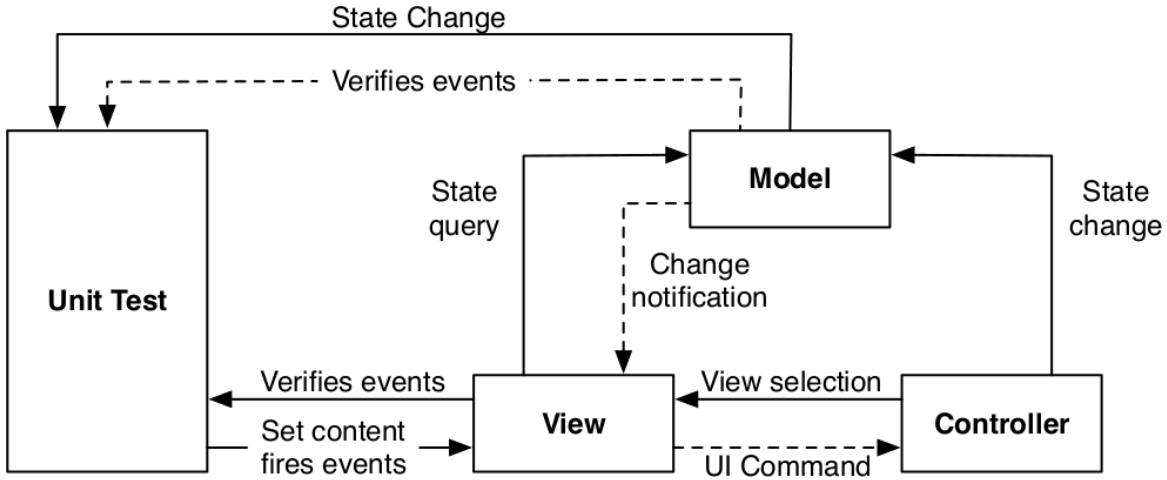
View-State Test Pattern

- Is the view updated when the model changes?

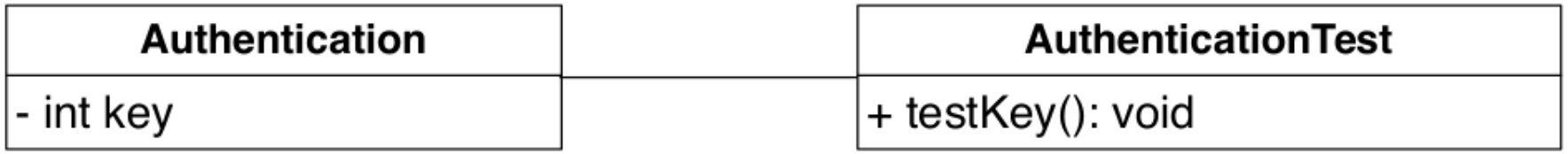


Model-State Test Pattern

- Is the model updated when the view is changed via the user



Reflection




```
public class AuthPrivacyTest {
    @Test public void testKey() throws Exception {
        Authentication auth = new Authentication();
        String privateKey = "privateKey";
        auth.setKey(privateKey);
        Class<? extends Authentication> cl = auth.getClass();
        // get the reflected object
        Field field = cl.getDeclaredField("key");
        // set accessible true
        field.setAccessible(true);
        Assert.assertEquals(field.get(auth), privateKey);
    }
}
```

Get the class object for Authentication

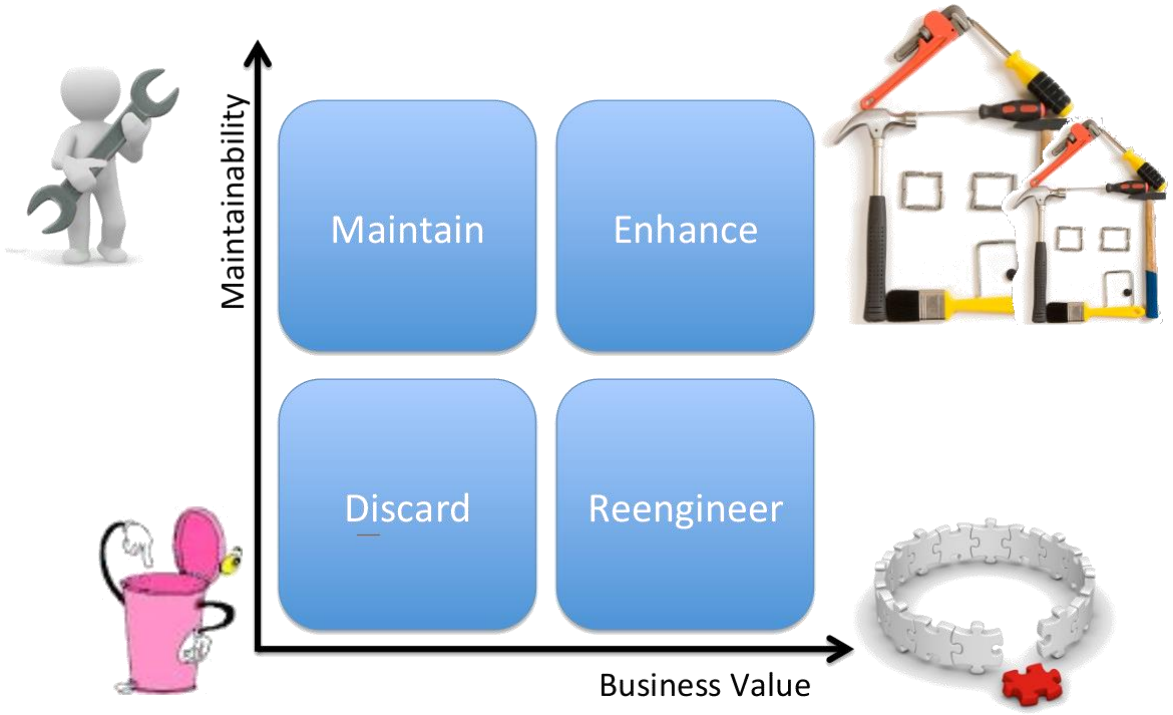
Get the field "key"

Set the field to be accessible

Used to test private attributes

Pattern-based Reengineering

When to reengineer & Reenginering process



- Inventory analysis
- Refactoring
- Analysis
- Object Design
- System Design

Refactoring & Rule of 7±2

- Process of incrementally changing the bad structure of a system or organization
- Functionality is not changed

Rules when refactoring

- Small, locale and testable steps
- Only refactor with automated tests
- Test changes
- Finish refactoring steps before beginning new ones

- If element consists of more than 7±2 elements there is a high chance that there is a problem

Rule of 7±2

- Methods > 30 lines
- Class > 7±2 methods
- Package > 7±2 classes
- Subsystem > 7±2 packages

Terminology

Terminology

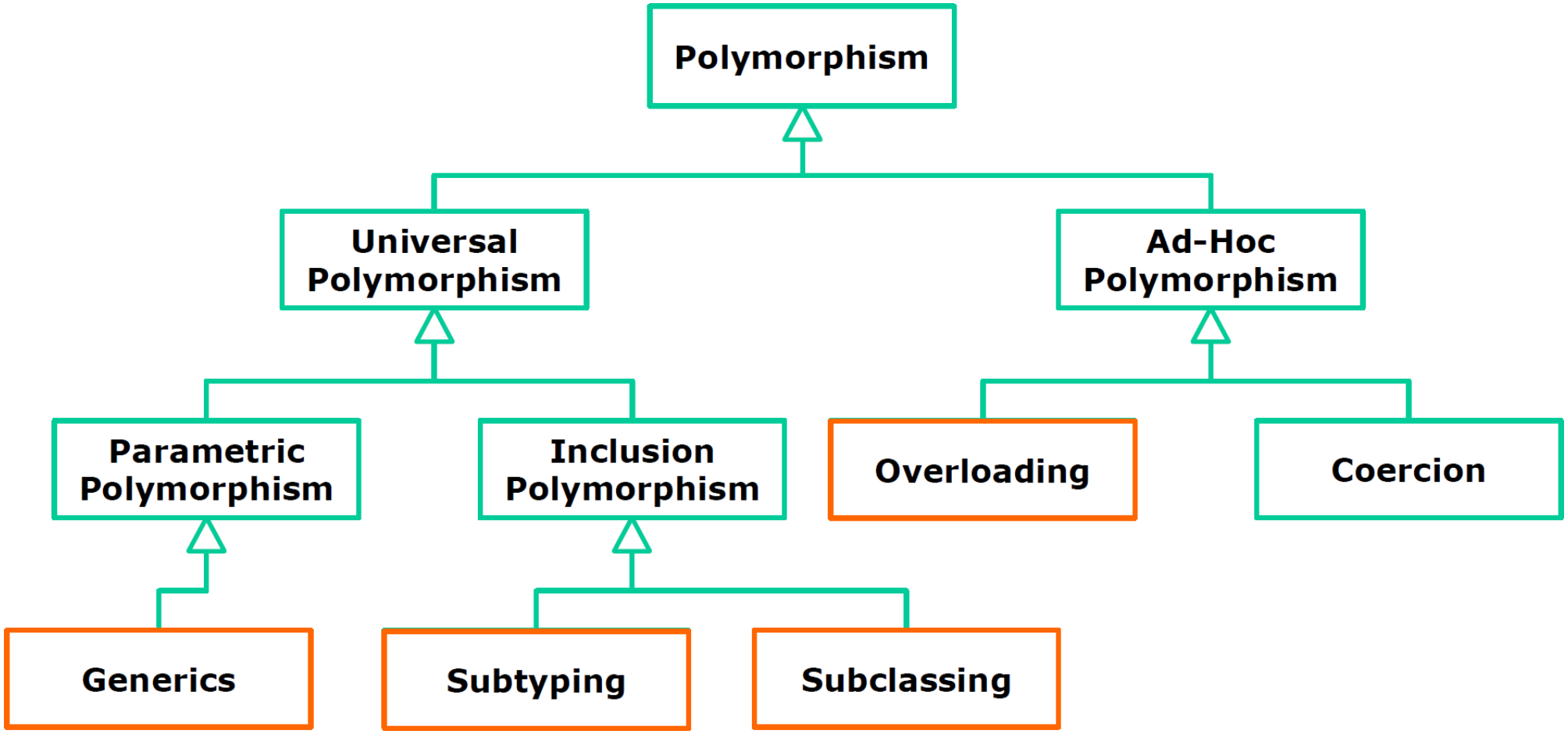
(Low) Coupling	measures the dependencies between subsystems
(High) Cohesion	Measures the dependencies among classes within a subsystem
Design Pattern	describes associations and collaborations of a set of classes
Architectural Style	is a pattern for a subsystem decomposition, i.e. describes relationships 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 underlying 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 specied interface
Connectors/Communication	Interactions between the components/subsystems
Failure	Deviation of the observed behavior from the specied one

Terminology

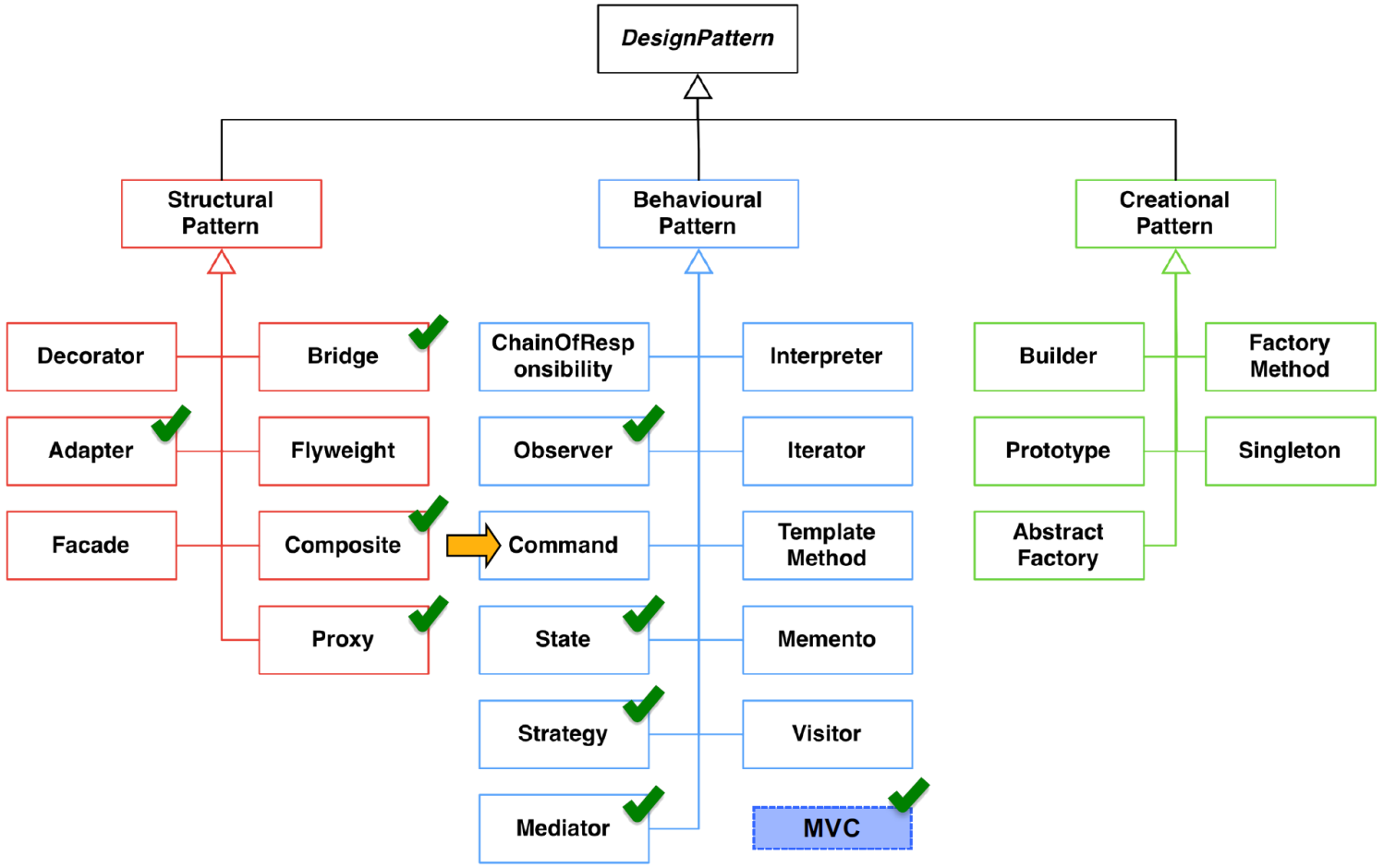
Fault/Bug	Mechanical or algorithmic cause of an error
Error	The system is in a state such that further processing by the system can lead to a failure.
Verification	Activity that checks if the observed behavior complies with the specied behavior of the system
Validation	Activity that checks if the observed behavior meets the needs informally expressed by a stakeholder
Marshalling	Transforming object to common representation and serializing afterwards to send over network
Unmarshalling	Deserializing data from network and transform the created object to a representation understandable by the receiver
Good Architecture	is the result of a consistent set of principles and techniques, applied consistently through all phases of a project. It is resilient in the face of changes and is a source of guidance throughout the product lifetime.
(A-)synchronous Communication	The client issues the method call and • waits (blocks) until the result is returned, • continues (non-blocking) and gets notied (Callbacks) by broker when the result is ready.

Taxonomies

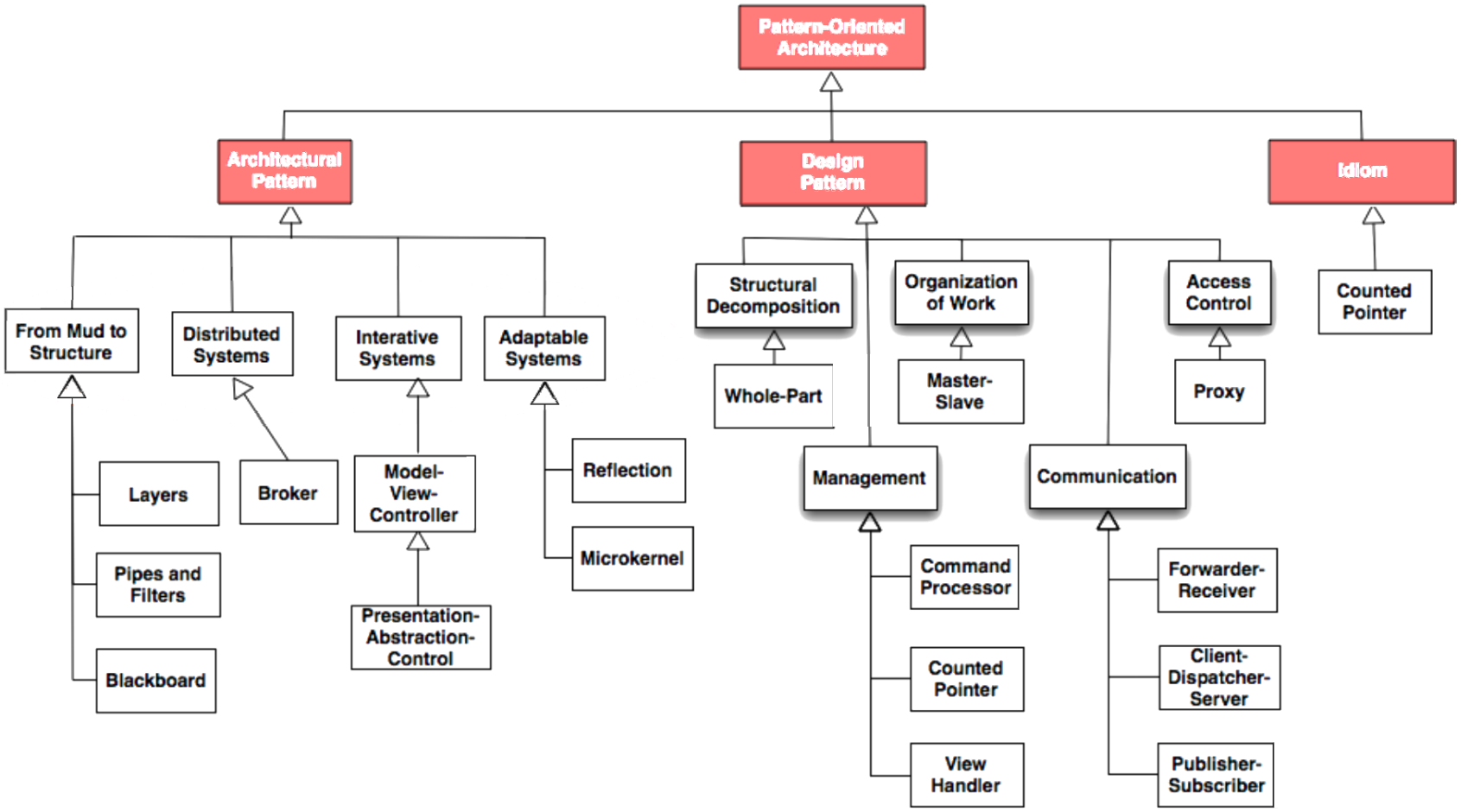
Polymorphism



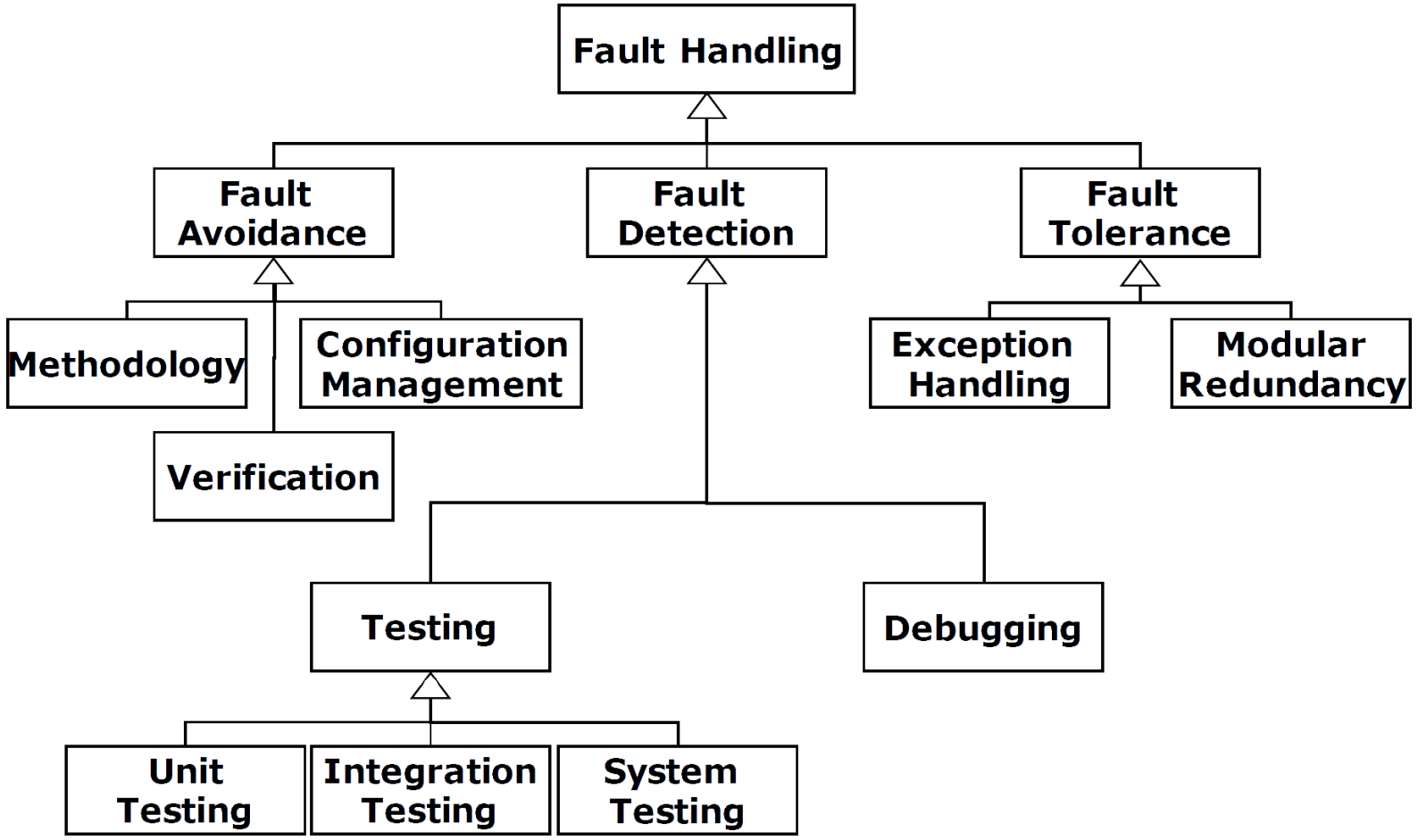
Design Patterns



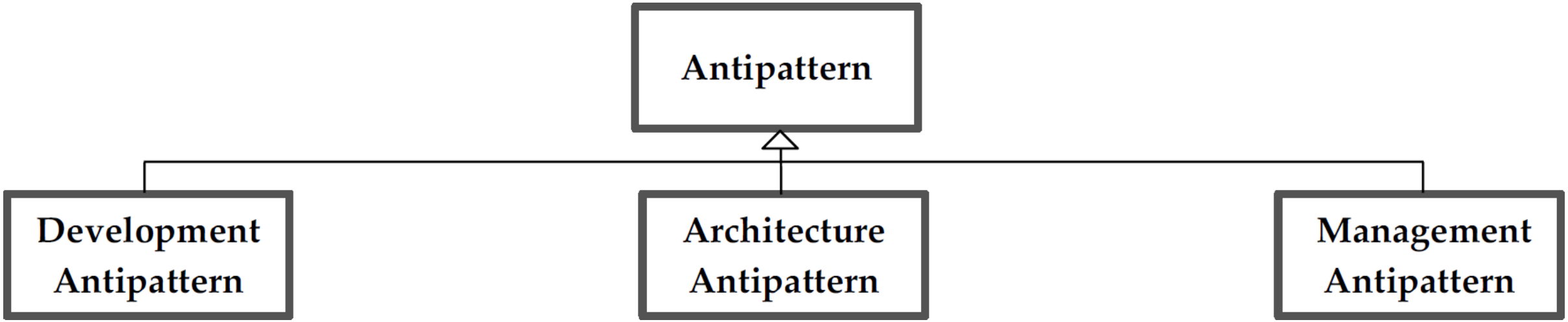
Pattern-oriented Architecture (GoF)



Fault Handling



Antipattern



Source Code Refactoring