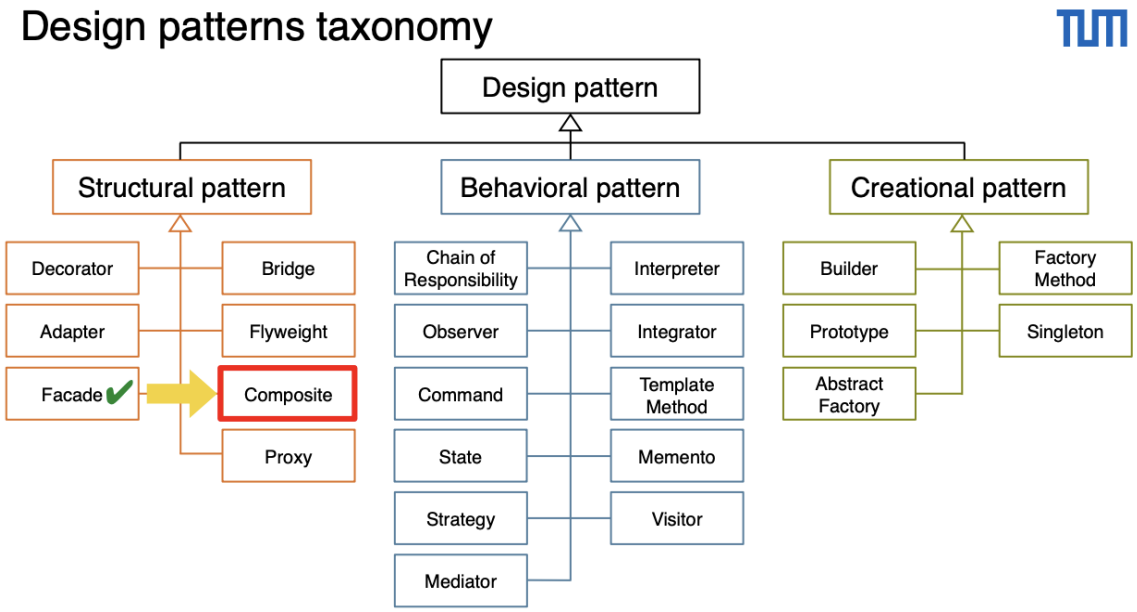
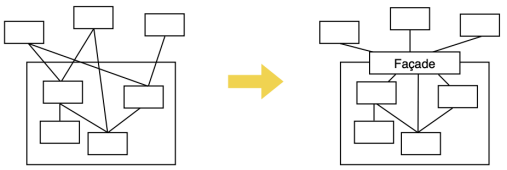
1. Design patterns



Composite pattern

Problem: there are hierarchies with arbitrary depth and width.
Solution: the composite pattern lets a Client treat an individual class called Leaf and Compositions of Leaf classes uniformly.
The Composite Pattern is a structural design pattern used to compose objects into tree structures and then work with these structures as if they were individual objects.
Sirve para construir objetos complejos a partir de otros más similares y simples entre sí, gracias a la composición recursiva y a una estructura en forma de árbol.

Façade pattern



Problem: spaghetti design.
Solution: subsystem interface object, it reduces coupling.

- Provides a unified interface for a subsystem (consists of a set of public operations and each operation is delegated to one or more operations in the classes behind the façade).
- Defines a higher-level interface that makes the subsystem easier to use.
- Allows to hide design spaghetti from the caller.

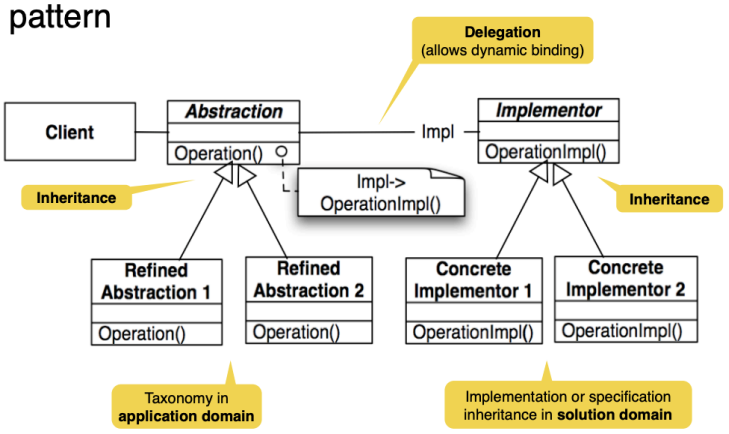
Advantages:

- Reduced complexity
- Fewer recompilations
- A façade can be used during integration testing when the internal classes are not implemented
- Possibility of writing mock objects for each of the public methods in the façade

Bridge pattern

Problem: many design decisions are made final at design time or at **compile time** (example: binding a client to one of many implementation classes of an interface).
Sometimes, it is desirable to delay design decisions until **runtime** (example: one client uses a very old implementation, the other client uses a more recent implementation of an interface).
Solution: the bridge pattern allows to delay the binding between an interface and its subclass to the startup time of the system.

Bridge pattern

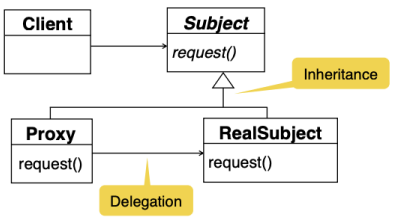


Proxy pattern

It solves t**wo problems**:

1) The object is complex, its instantiation is expensive.
   Solution:
   - Delay the instantiation until the object is actually used.
   - If the object is never used, then the costs for its instantiation do not occur.
2) The object is located on another node (i.e. on a web server), accessing the object is expensive.
   Solution:
   - Instantiate and initialize a "smaller" local object, which acts as a representative ("proxy") for the remote object.
   - Try to access mostly the local object.
   - Access the remote object only if really necessary.

- **`Proxy`** and **`RealSubject`** are subclasses of the *abstract* class **`Subject`**
- The **`Client`** always calls **`request()`** in an instance of type **`Proxy`**
- The implementation of **`request()`** in **`Proxy`** then uses delegation to access **`request()`** in **`RealSubject`**



Why is expensive in object-oriented systems?

- object creation
- object initialization

The proxy pattern allows to defer object creation and object initialization to the time the object is needed:

- reduces the cost of accessing objects.
- the proxy acts as a stand-in for the real object
- the proxy creates the real object only if the user asks for it.
- provides location transparency.

Use cases of the proxy pattern:

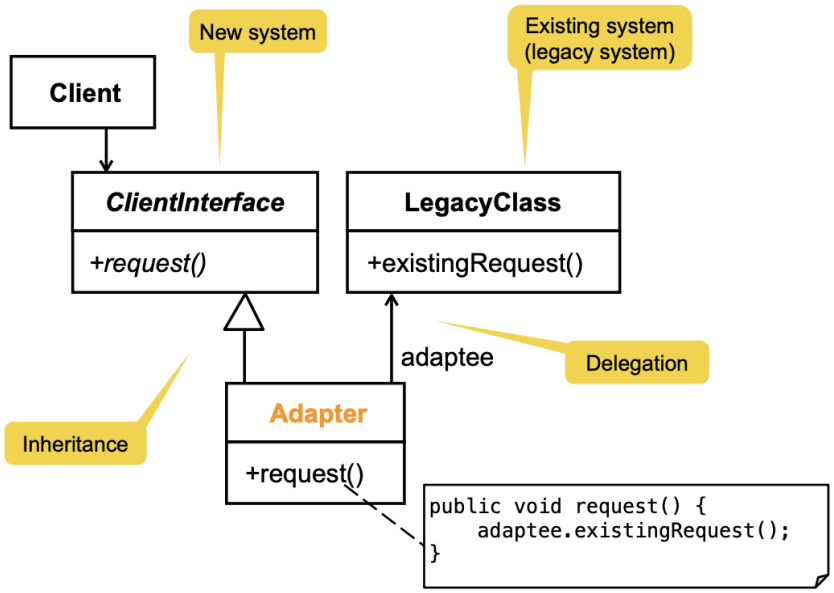
- Caching (remote proxy): the proxy object is a local representative for an object in a different address space (caching is good if information does not change too often, if information changes, the cache needs to be flushed).
- Substitute (virtual proxy): the proxy object acts as a stand-in for an object which is expensive to create or download (good for information that is not immediately accessed and useful for objects that are not visible).
- Access control (protection proxy): the proxy object provides access control to the real object (beneficial when different objects should have different access and viewing rights).

Adapter pattern

Problem: an existing component offers functionality, but is not compatible with the new system being developed.

Solution: the adapter pattern connects incompatible components:

- allows the reuse of existing components.
- converts the interface of the existing component into another interface expected by the calling component.
- useful in interface engineering projects and in reengineering projects.
- often used to provide a new interface for a legacy system.
- also called wrapper.



Observer pattern

Problem: an object that changes its state often and multiple views of the current state.
Requirements:

- the system should maintain consistency across the views, whenever the state of the observed object changes.
- the system design should be highly extensible.
- it should be possible to add new views without having to recompile the observed object or existing views.

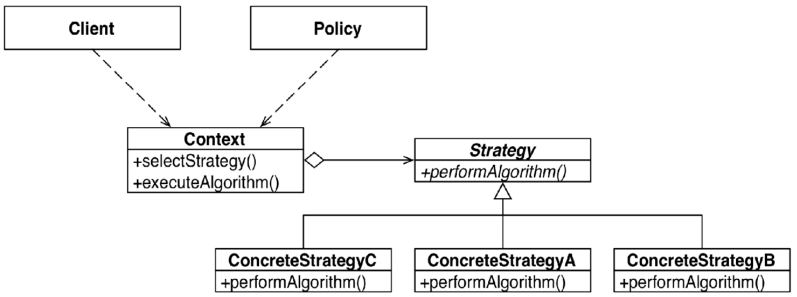
Solution: model a 1-to-many dependency between objects. Connect the state of an observed object, the subject with many observing objects and the observers.
Benefits:

- maintain consistency across redundant observers.
- optimize a batch of changes to maintain consistency.

Strategy pattern

Problem: different algorithms exist for a specific task.
Solution: the strategy pattern allows to switch between different algorithms at run time based on the context and a policy,

The **`Policy`** decides which **`ConcreteStrategy`** is best in a given **`Context`**



Clues for the use of design patterns

- **Text:** “complex structure”, “must have variable depth and width”
  → **Composite pattern**
- **Text:** “must provide a policy independent from the mechanism”, “must allow to change algorithms at runtime”
  → **Strategy pattern**
- **Text:** “must be location transparent”
  → **Proxy pattern**
- **Text:** “states must be synchronized”, “many systems must be notified”
  → **Observer pattern** (part of the MVC architectural pattern)

Clues for the use of design patterns

- **Text:** “must interface with an existing object”
  → **Adapter pattern**
- **Text:** “must interface to several systems, some of them to be developed in the future”, “an early prototype must be demonstrated”, “must provide backward compatibility”
  → **Bridge pattern**
- **Text:** “must interface to an existing set of objects”, “must interface to an existing API”, “must interface to an existing service”
  → **Façade pattern**

State pattern

State:

- An abstraction of the attributes of a class (aggregation of several attributes of a class)
- An equivalence class of all those attribute values that do not need to be distinguished
- Has a certain duration
- Example: state of a bank
  - State 1: bank has money
  - State 2: bank is bankrupt

Motivation:

- There is an object with interesting dynamic behavior that can be in several states with different behavior in each of these states.
- The object changes its behavior when its state changes at runtime
- The source code contains large, multipart conditional statements that depend on the object's state

Properties:

- Useful for objects which change their state at runtime
- Uses polymorphism to define behaviors for different states of an object
- The selection of the subclass depends on the state of the object

Benefits:

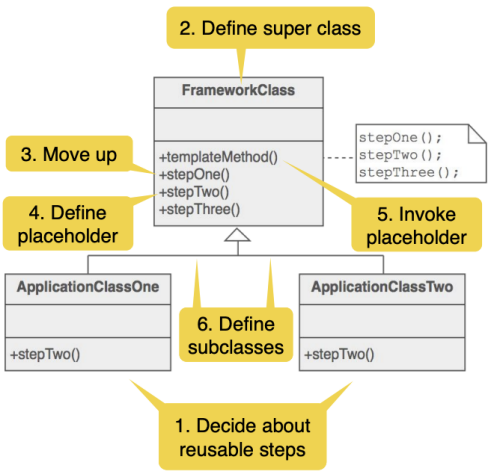
- Localizes state specific behavior
- Extensibility and flexibility: easy to add more states for additional behavior
- Avoids cluttered if-else or switch-case statements
- Makes state transitions explicit

Template method pattern

Problems:

- two different components (e.g. classes) have significant similarities, but demonstrate no reuse of a common interface or implementation
- if a change common to both components becomes necessary, duplicate effort is necessary

1. Decide which steps are standard and which steps are concrete to each of the current classes
2. Define a new abstract super class to host the "don't call us, we'll call you" framework
3. Move the body of the algorithm and the definition of all standard steps to the new super class
4. Define placeholder methods in the super class for each step that requires different implementations
5. Invoke the placeholder method(s) from the template method
6. Define all existing classes as subclasses of the new abstract super class



Command pattern

5 steps to realize a command pattern:

1. Create the Command interface (the key to the pattern: declares the interface for executing operations, includes an abstract operation execute())
2. Create the ConcreteCommand classes (control objects), each concrete class specifies a receiver-action pair by storing the receiver as an instance variable and provides a different implementation of the execute() method to invoke the specific request.
3. Create the Receiver (entity objects), has the knowledge to carry out the specific request.
4. Create the Invoker (boundary objects).
5. Create the Client

Benefits:

- Complexity reduction.
- Dealing with change.

Mediator pattern

- A class should only have one responsibility
- It should only have one reason to change

Benefits:

- Easier testing.
- Lower coupling.
- Better organization.

Problem: design reusable components, but dependencies between potentially reusable pieces demonstrate the spaghetti code phenomenon.
Motivation: reduce coupling between classes that communicate with each other.

Factory method pattern

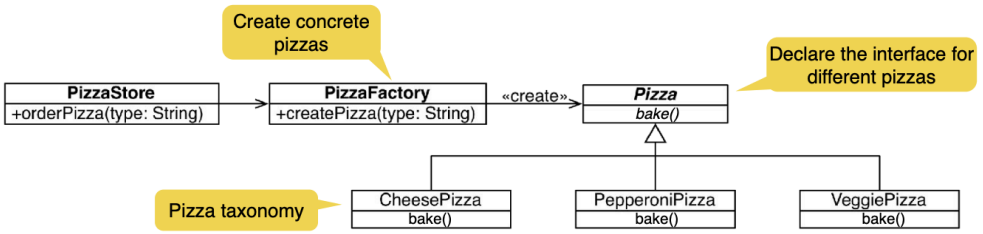
Problem: instantiating an object with the new operator creates a dependency on a concrete class, which means high coupling. The new operator is considered harmful.
Object oriented design goals: low coupling and high cohesion.
Use a factory class -> a class responsible for the instantiation of objects:

- define an interface for creating an object, but let subclasses decide which class to instantiate (defer instantiation to subclasses).
- defining a virtual constructor.
- First steps towards dependency injection.

```
public class PizzaStore {
    PizzaFactory factory;
    public PizzaStore(PizzaFactory factory) {
        this.factory = factory;
    }
    public Pizza orderPizza(String type) {
        Pizza pizza = factory.createPizza(type);
        pizza.bake();
        return pizza;
    }
}
```

Delegation

```
public class PizzaFactory {
    public Pizza createPizza(String type) {
        Pizza pizza = null;
        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza;
    }
}
```



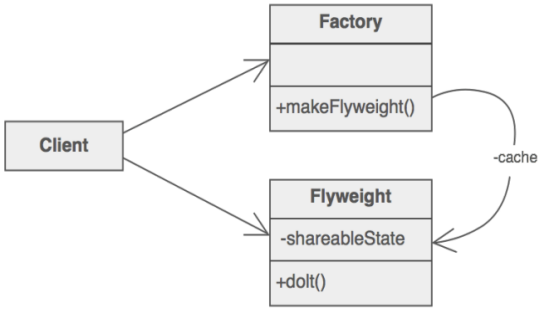
Abstract factory pattern

+ compatibility of products produced by the factory
+ low coupling between concrete products and client code
+ single responsibility principle: you can extract the product creation code into one place, making the code easier to maintain
+ open/closed principle: you can introduce new variants of products without breaking existing client code
- complex code: a lot of new interfaces and classes are introduced along with the pattern

Flyweight pattern

Problem: designing objects down to the lowest levels of system granularity provides optimal flexibility, but it can be unacceptably expensive in terms of performance and memory usage.
Purpose: share objects to allow their use at fine granularity without prohibitive cost
Each flyweight object is divided into two pieces:

- extrinsic / mutable: the state dependent part: stored or computed by client objects, and passed to the flyweight when its operations are invoked.
- intrinsic / immutable: the state independent part: stored (shared) in the flyweight object



The Flyweight pattern is used to reduce the number of objects created, decrease memory footprint and increase performance. Imagine you need 1000 car objects in an online car racing game, and the car objects vary only in their current position on the race track. Instead of creating 1000 car objects and adding more as users join in, you can create a single car object with common data and a client object to maintain the state (position of a car).

Builder pattern

Problem:

- an application needs to create the elements of a complex aggregate
- the specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage
- Intent:
    - separate the construction of a complex object from its representation so that the same construction process can create different representations
    - parse a complex representation, create one of several targets

The builder pattern is useful when it comes to the instantiation of objects with many attributes, especially if many of them are optional. Using the Builder pattern improves the readability of the code and makes it easier to add new attributes.

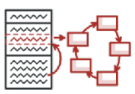
Summary

Clues for the use of design patterns

- **Text:** “complex structure”, “must have variable depth and width”
  → **Composite pattern**
- **Text:** “must provide a policy independent from the mechanism”, “must allow to change algorithms at runtime”
  → **Strategy pattern**
- **Text:** “must be location transparent”
  → **Proxy pattern**
- **Text:** “states must be synchronized”, “many systems must be notified”
  → **Observer pattern** (part of the MVC architectural pattern)

- **Text:** “must interface with an existing object”
  → **Adapter pattern**
- **Text:** “must interface to several systems, some of them to be developed in the future”, “an early prototype must be demonstrated”, “must provide backward compatibility”
  → **Bridge pattern**
- **Text:** “must interface to an existing set of objects”, “must interface to an existing API”, “must interface to an existing service”
  → **Façade pattern**

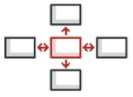
Summary



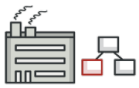
The **state pattern** lets one object alter its behavior when its internal state changes by turning the states into objects

The **template method pattern** defines the skeleton of an algorithm in the superclass and lets subclasses override specific steps of the algorithm without changing its structure

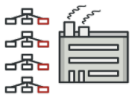
The **command pattern** turns a request into an object containing all information, allows to delay or queue requests, and supports undoable operations



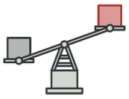
The **mediator pattern** reduces dependencies between objects, restricts the communications between them and forces them to collaborate only via a mediator object



The **factory method pattern** provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created



The **abstract factory method** produces families of related objects without specifying their concrete classes



With the **flyweight pattern**, more objects fit into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object

The **builder pattern** constructs complex objects step by step and allows to produce different types and representations of an object using the same construction code

Architectural styles

Subsystem decomposition -> identification of subsystems, services and their relationships to each other.
Architectural style -> a pattern for a subsystem decomposition.
Software architecture -> instance of an architectural style.

Client server architecture

Often used in the design of database systems:

- Client -> user application
- Server -> database access and manipulation
- Client -> requests a service from the server

Functions performed by the client:

- input by the user (customized user interface)
- sanity checks of input data

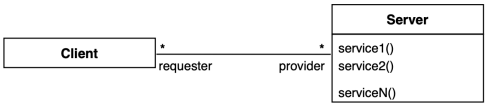
Functions performed by the server:

- centralized data management
- provision of data integrity and database consistency
- provision of database security

One or more servers provide services to clients
Each client calls a service offered by the server:

- server performs service and returns result to client
- client knows interface of the server
- server does not know the interface of the client

Response is typically immediate
End users interact only with the client




```
@GetMapping("/employees")
 List<Employee> all() {
   return repository.findAll();
 }
@PostMapping("/employees")
 Employee newEmployee(@RequestBody Employee newEmployee) {
   return repository.save(newEmployee);
 }
@DeleteMapping("/employees/{id}")
 void deleteEmployee(@PathVariable Long id) {
   repository.deleteById(id);
 }
```

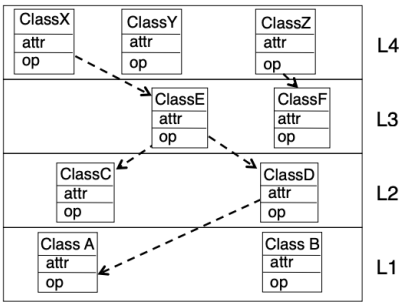
Layered architecture

A layer is a subsystem that provides a service to another subsystem with the following restrictions:

- a layer only depends on services from lower layers.
- a layer has no knowledge of higher layers.

A layered architecture is **closed** if each layer can only call operations from the layer directly below (also called “direct addressing”).

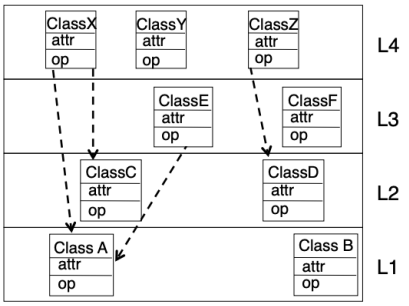
Design goals: maintainability, flexibility, portability



more portable ➡ **low coupling** 😄, but potentially a **bottleneck**

A layered architecture is **open** if a layer can call operations from any layer below (also called “indirect addressing”).

Design goals: high performance, real-time operations support



more **efficient** ➡ **high coupling** 😒

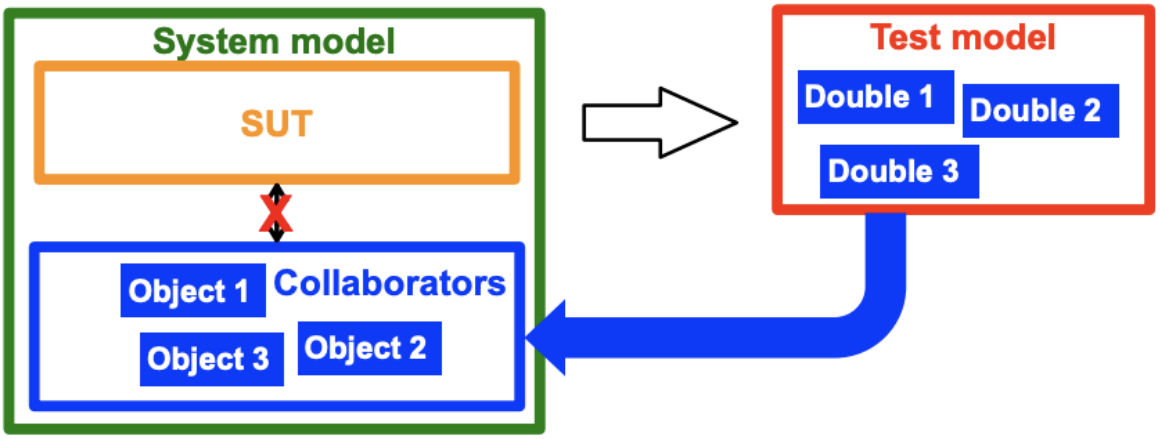
3 layered architectural style (often used for the development of web applications):

- the web browser implements the user interface.
- the web server serves requests from the web browser.
- the database manages and provides access to the persistent data.

Architectural style where an application consists of 3 hierarchical ordered layers.
Also there is a 4 layered architectural style, 7 layered architectural style…

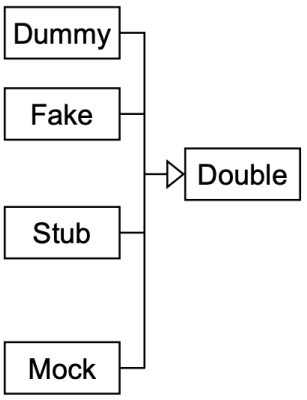
Object oriented test modeling

- Start with the system model
- The system contains the SUT (system under test)
- The SUT does not exist in isolation, it interacts with other participating objects in the system model that are not yet implemented: collaborators
- The test model is derived from the SUT
- To be able to interact with collaborators, we add objects to the test model
- These are called test doubles

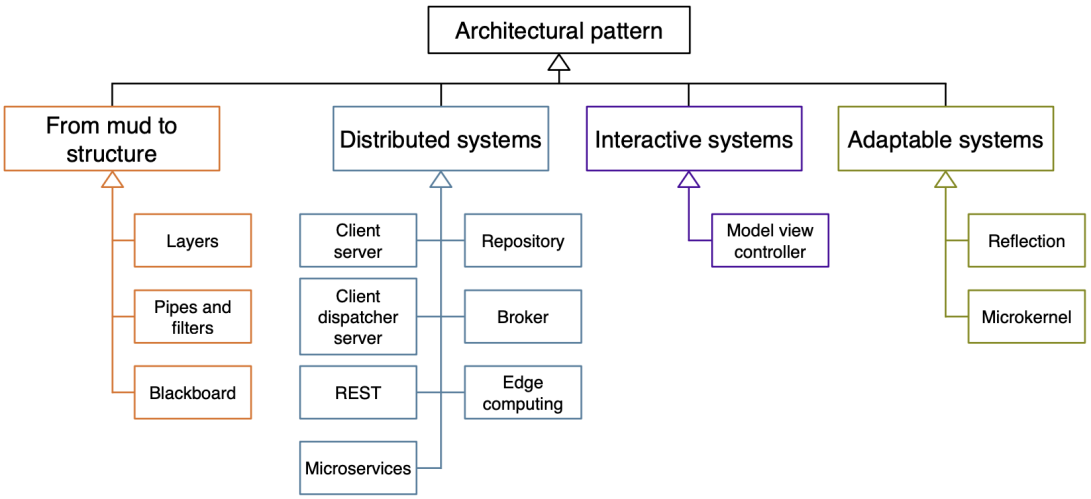


Taxonomy of test doubles

- **Dummy:** often used to fill parameter lists, passed around but never actually used
- **Fake:** a working implementation that contains a “shortcut” which makes it not suitable for production code
  - **Example:** a database stored in memory instead of on a disk
- **Stub:** provides canned answers (e.g. always the same) to calls made during the test
  - **Example:** random number generator that always return 3.14
- **Mock:** mimic the behavior of the real object and know how to deal with a specific sequence of calls they are expected to receive



2. Architectural pattern



Introduction

What is a **good architecture**:

- result of a consistent set of principles and techniques, applied consistently through all phases of a project
- resilient in the face of (inevitable) changes
- source of guidance throughout the product lifetime
- reuse of established engineering knowledge

A software architecture consists of **components** and **connectors**:

- Components (subsystems): computational units with specified interfaces
- Connectors (communication): interactions between the components

Overview

From mud to structure: subsystem decomposition

- Layers -> each subsystem presents a layer of abstraction
- Pipes and filters -> each subsystem is a processing step formulated in a filter connected by pipes
- Blackboard -> subsystems are knowledge experts working together to solve a problem without a solution strategy

Distributed systems: collaborating systems on different nodes

- Repository: exchange of persistent data between multiple clients
- Client server: clients interact directly with servers
- Client dispatcher server: clients interact with servers via an additional name server
- Broker: systems interact via remote service invocations
- REST: representational state transfer, often used for web services, based on client server, layers and proxy
- Edge computing: synchronization of data, real-time access, and availability are realized simultaneously
- Microservices: system consists of many small services

Interactive systems: systems interacting with a user

- Model view controller: subsystem decomposition with 3 components: model (entity objects), view (boundary objects), controller (controller objects)

Adaptable systems: systems evolving over time

- Reflection: self awareness provides information about system properties
- Microkernel: extensible minimal functional core

Model view controller (MVC)

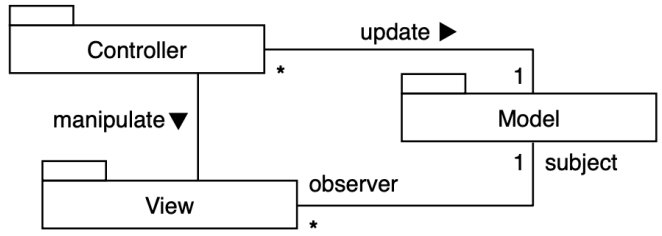
Problem: in systems with high coupling any change to the boundary objects (user interface) often forces changes to the entity objects (data)

- the user interface cannot be re-implemented without changing the representation of the entity objects
- the entity objects cannot be reorganized without changing the user interface

Solution: decouple data access (entity objects) and data presentation (boundary objects)

- separation of concerns
- improved testability and extensibility

Model: process and store application domain data (entity objects)
View: display information to the user (boundary objects)
Controller: interact with the user and update the model (which notifies the view)



There are two choices for the notification:

- **Pull notification variant**: view and controller obtain the data from the model
- **Push notification variant**: the model sends the changed state to view and controller

Benefits:

- multiple synchronized views of the same model
- pluggable views and controllers
- exchangeability of look and feel
- framework potential

Challenges:

- increased complexity
- potential for excessive number of updates
- close connection between view and controller
- close coupling of view and controllers to model

Client dispatcher server pattern

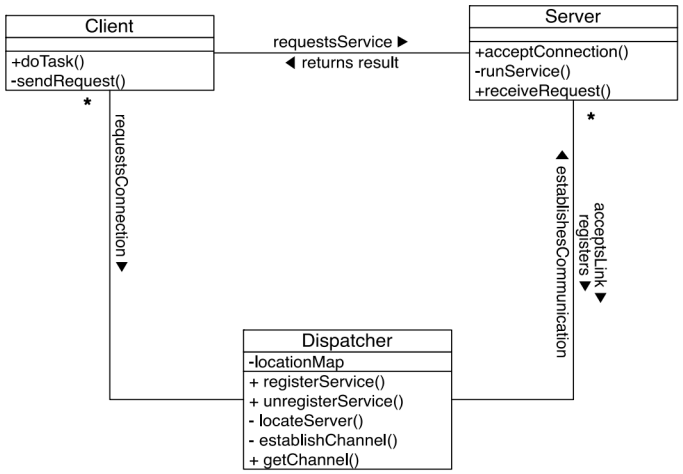
Context: When a client uses a remote server over a network it needs to establish a connection before the client can communicate with the server

Problem:

- client and server needs are not separated in many applications, causing unnecessary code complexity in service invocations
- it would better to separate the core functionality provided by the server from the details of the communication mechanism between client and server
- allow servers to dynamically change their location without impacting client code

Solution:

- insert a dispatcher component between client and server that provides the connection
- allow the client to refer the server by name instead of the physical location (location transparency)
- establish a channel between the client and server, reducing a possible communication bottleneck



Benefits:

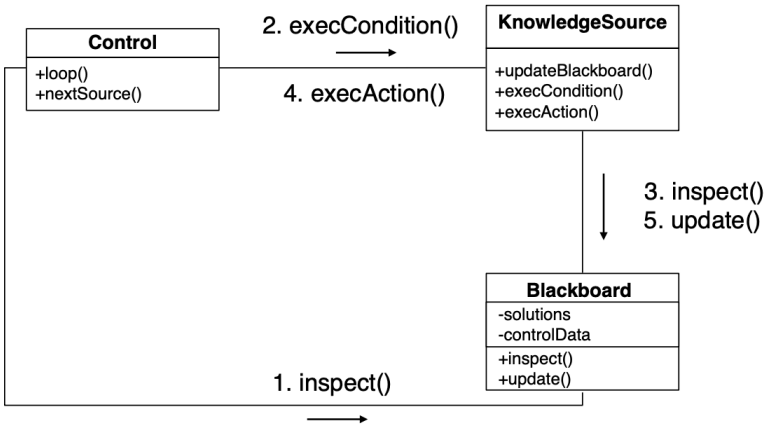
- exchangeability of servers
- location and migration transparency
- reconfiguration
- fault tolerance

Protocols:

- CDprotocol: specifies how a client must look for a particular server, deals with communication errors (time out, server does not exist, …)
- DSprotocol: specifies how a server registers with the dispatcher and determines the activities needed to establish a communication channel between client and server
- CSprotocol: specifies how client and server communicate with each other

Blackboard pattern

Blackboard is the repository for the problem, partial solutions and new information (hypotheses, …)
Knowledge sources read anything that is placed on the blackboard and place new information created by them on the blackboard
Control governs the flow of the problem solving activity in the system: how knowledge sources are notified of new information on the blackboard



Steps:

1. define the problem
2. define the solution space
3. identify the knowledge sources
4. define the blackboard
5. define the control
6. implement the knowledge sources

Advantages:

- problem solving support
- changeability and maintainability
- fault tolerance and robustness

Limitations:

- difficulty of testing
- no solution guaranteed
- difficulty to establish a good control strategy
- high development effort

Broker

Goals:

- low coupling
- location transparency
- runtime extensibility

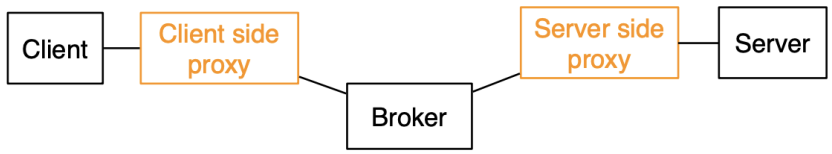
Functionality:

- provides the name service
- transmission of request to services (client -> server)
- transmission of responses and exceptions (server -> client)

There is no direct connection between client and server

Nonfunctional requirement achieved: platform independence

With proxies:

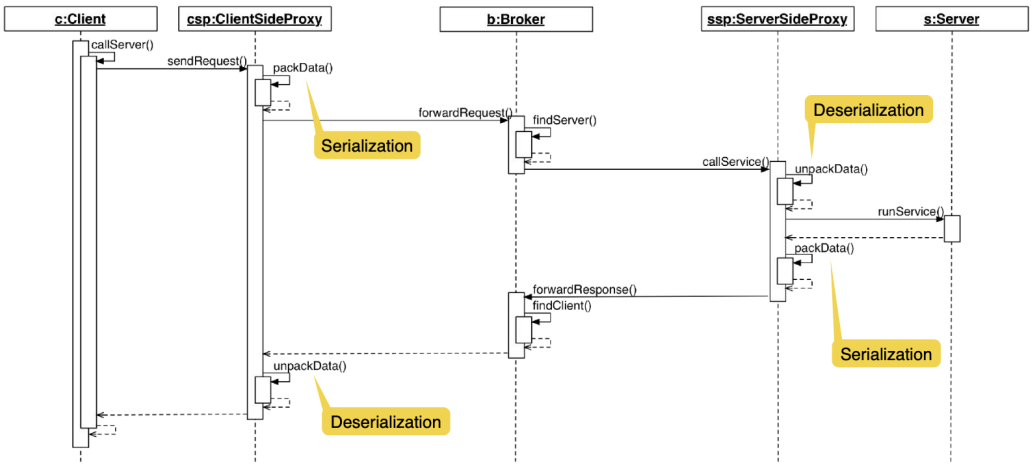


Client side proxy: layer between client and broker

- remote objects (in the server) appear as local ones
- translates the object model specified by the server into an object model in the client
- hides interprocess communication details between client and broker

Server side proxy: layer between broker and server

- receives requests from the broker
- hides interprocess communication details between broker and server
- calls the services in the server



Synchronous:

- The client issues the method call and waits (blocks) until the result is returned

Asynchronous:

- The client issues the method call, and continues (a non-blocking method call)
- It gets notified by broker when the result is ready
- This is usually implemented using callbacks

REST

Context:

- shared resources and services with large numbers of distributed clients
- access control and quality of service are important

Problem:

- manage a set of shared web resources and services
- make them modifiable and reusable
- support scalability and availability while spreading the resources across multiple distributed components

Solution:

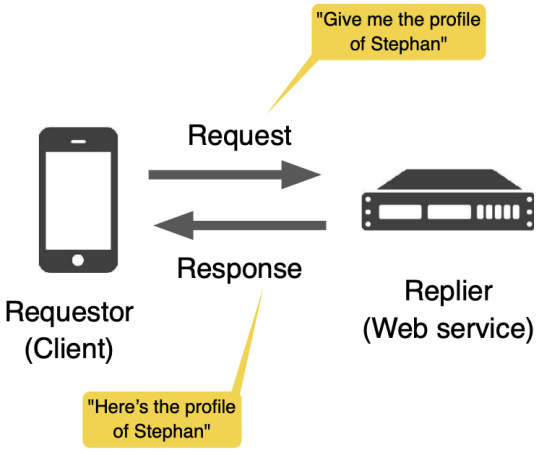
- provide an abstraction that models the structure and behavior of the world wide web
- REpresentational State Transfer (REST)

Requirements:

- Client server (CS): use the client server architectural style
- Stateless (CSS): communication must be stateless
- Cacheable (C$SS): data within a response to a request must be implicitly or explicitly labeled as cacheable or non-cacheable
- Uniform interface (U): introduction of a uniform interface between components
- Layered system (LS): the REST architecture should be composed of hierarchical layers
- Code on demand (COD): allows client functionality to be extended by downloading and executing code

Resources:

- request response is a stateless exchange protocol (the Replier does not keep a history of old requests)



HTTP is an application layer protocol used to communicate with web services
HTTP messages consist of a URI, a HTTP method, a header that contains key value pairs and a message body that can contain arbitrary data

Summary

- Software architectures are instances of architectural styles, which are patterns for a subsystem decomposition based on specific design goals

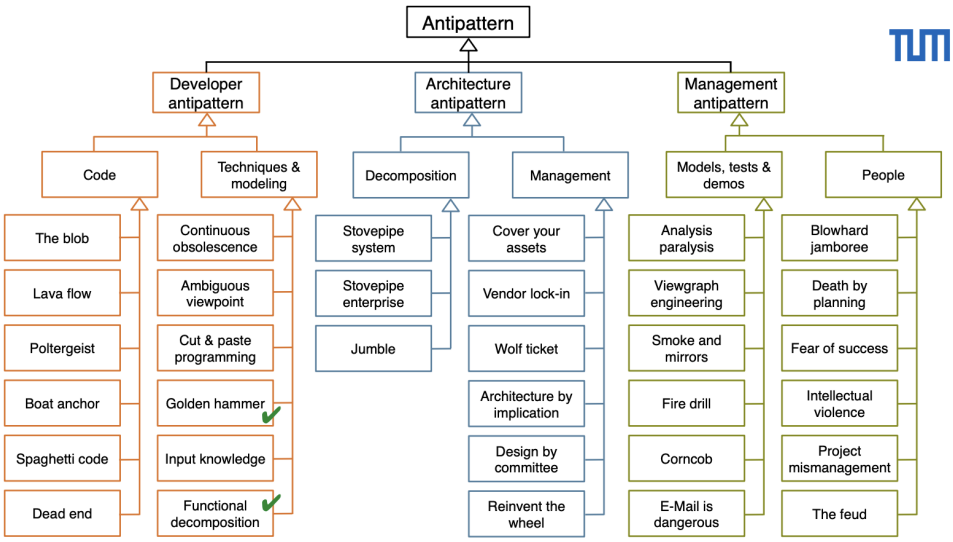
- **Model view controller:** subsystem decomposition with 3 components: model (entity objects), view (boundary objects), controller (controller objects)

- **Client dispatcher server:** clients interact with servers via an additional name server

- **Blackboard:** subsystems are knowledge experts working together to solve a problem without a known solution strategy

- Broker: systems interact via remote service invocations
- REST is an architectural style based on the layered architecture pattern
- REST typically use **HTTP** and **HATEOAS**
- REST clients and web services are easy to prototype
  - However: there are **tricky details** when the system becomes more complex
- Avoid REST bad practices
  - Upcoming lectures about anti-patterns and code smells
- GraphQL addresses some of the problems and enables querying across multiple resources using a type based schema definition language
- gRPC can include code generators and modern and efficient communication mechanisms

3. Antipatterns

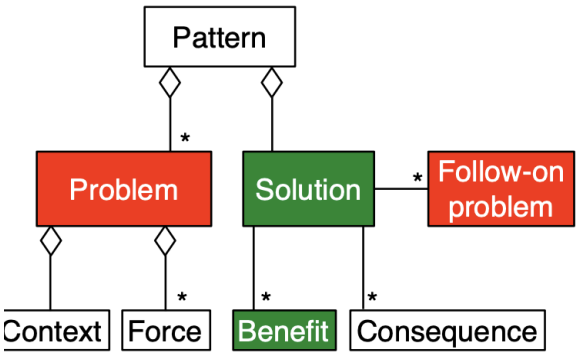


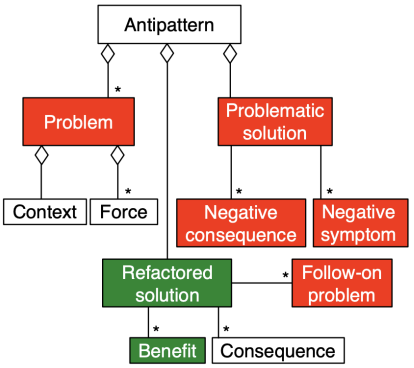
A pattern has two parts: problem and solution:

- The problem is elaborated in terms of a **context** and a set of **forces**
- The solution resolves these forces with **benefits** and **consequences**
- To be considered as a pattern, the solution must be applicable to **more than one** specific problem

Solutions usually generate **follow-on problems**

- Follow-on problems can again be elaborated in terms of context and forces which may lead to the applicability of other patterns





Patterns can evolve into antipatterns **when change occurs** (change of requirements, change of project parameters, change of methodology…).

Antipatterns identify and categorize common mistakes in software practice.

“An antipattern is something that looks like a good idea, but which backfires badly when applied”

An antipattern consists of 1 type of problem(s) and 2 solutions:

- the **problematic solution** describes a commonly occurring solution that generates overwhelming **negative consequences** and **negative symptoms**
- the **refactored solution** describes how the **problematic solution** can be reengineered to avoid these negative consequences and lead to **benefits** again
- the **refactored solution** can lead again to **follow-on problems**

Golden hammer

“When the only tool you have is a hammer it is tempting to treat everything as if it were a nail”

General form:

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

Symptoms and consequences:

- identical tools are used for a wide array of diverse products
- system architecture depends on a particular application suite and a specific vendor tool set

Typical causes:

- large investment has been made in products and specific technologies
- reliance on proprietary product features that are not available from other vendors or products

Variations: obsessive use of a favorite software concept or design pattern

Refactored solution:

- project organization develops commitment to explore new technologies
- software system is designed with well defined boundaries
- software developers keep up to date on technology trends
- management adopts a commitment to open systems and architectures
- management encourages hiring of people with different backgrounds

Identical tools are used for a wide of diverse problems

Functional decomposition

Also known as No OO

General form: everything is a function, lots of files called misc, util, util1, aux1, aux2…

Unbalanced forces: management of complexity, change management is difficult

Symptoms:

- the functionality is spread all over the system
- maintainer must understand the whole system to make a single change to the system

Consequences:

- source code is hard to understand
- source code is complex and impossible to maintain
- user interface is often awkward and non-intuitive

Typical causes:

- programmers have been trained only in an imperative / functional language
- designers have been trained with a functional decomposition method (DeMarco)

Refactored solutions:

- object oriented analysis
- object oriented reengineering (process)

Best practice: first identify use cases, then identify objects

Changing and adding new functionality is difficult, because the functionality is spread all over the system

Lava flow

Also known as: dead code
General form: lava like “flows” of previous development hardened into a basalt like mass of code, difficult to remove once it has solidified
Systems and consequences:

- unused or commented out code; undocumented complex, important looking code
- functions or classes that don’t relate to the system architecture
- “evolving” architecture

Typical causes:

- R&D code placed into production
- implementation of several trial approaches toward implementing some functionality
- high programmer turnover rate
- fear of breaking something and not knowing how to fix it
- architectural scars; unclear, repeatedly changing project goals

Refactored solution:

- architecture centric management
- ensure that sound architecture precedes production code development
- architecture must be backed up by configuration management process
- avoid architecture changes during active development

Lots of unused or commented out code, undocumented complex, important looking code, evolving architecture

The blob

Also known as: god class
General form:

- majority of responsibilities are allocated to single complex controller
- associated with simple data classes

Systems and consequences:

- a single class with a huge number of unrelated attributes and operations encapsulated the class
- single controller encapsulating the functionality (ex.: a procedural main program)
- the blob class is typically too complex for reuse and testing

Typical causes:

- lack of an object oriented architecture
- lack of (any) architecture
- lack of architecture enforcement
- too limited intervention in iterative projects

Refactored solution:

- distribution of responsibilities into smaller classes
- identify or categorize related attributes and operations
- move them into the classes they belong to
- remove redundant, indirect associations

A single (god) class with a huge number of unrelated attributes and operations

Spaghetti code

1. Spaghetti code -> complicated, difficult to understand and impossible to maintain software
2. Lasagna code -> simple, understandable and layered structure. However: monolithic and not easy to modify
3. Ravioli code -> ideal structure - small and loosely coupled components can be modified or replaced without significantly affecting other components

General form:

- the software has very little structure
- object methods are invoked in a single, multistage process flow

Systems and consequences:

- methods are process oriented, objects are named as processes
- execution flow is dictated by the class implementation of the objects, not by the class users
- inheritance is not used to provide for extension of the system; polymorphism is not used
- the source code is difficult to reuse: point of diminishing returns: the software maintenance effort is greater than a complete reengineering effort

Typical causes:

- no design prior to implementation
- inexperience with object oriented design technologies

Refactored solution:

- software refactoring (nicer term for managers: maintenance of software investment)
- incremental (step by step refactoring): also called incremental engineering
- code cleanup should be a natural part of the development process

Ad hoc structure which makes it hard to extend or optimize a model or code

Cut and paste programming

Also known as: clipboard coding, software cloning, software propagation
Root causes: sloth
Unbalanced forces: management of resources, technology transfer

Systems and consequences:

- software defects are replicated through the system (the same software bug reoccurs despite many local fixes)
- lines of code increase without adding to overall productivity
- code reviews and inspections are needlessly extended
- it becomes difficult to locate and fix all instances of a particular mistake
- code can be reused with a minimum of effort
- excessive software maintenance costs

Typical causes:

- short term payoff more important than long term investment
- no reward of reusable components
- lack of abstraction
- reusable components, once created, are not sufficiently documented or made readily available to developers
- “not invented here” syndrome

- lack of forethought or forward thinking among the development teams
- inexperience with new technology or tools

Refactored solution:

- white box reuse (inheritance): a new subclass reuses the functionality of the superclass and may offer new functionality
- black box reuse (delegation): a new class offers the aggregated functionality of the existing classes

Excessive code duplication (clones) make the software expensive and difficult to maintain

Vendor lock-in

General form: a software project adopts a product technology and becomes completely dependent upon the vendor's implementation

Systems and consequences:

- commercial product upgrades drive the application software maintenance cycle
- promised product features are delayed or never delivered
- application programming requires in-depth product knowledge

Typical causes:

- the product is selected based entirely upon marketing and sales information, and not upon more detailed technical inspection
- the product varies from published open system standards because there is no effective conformance process for the standard

Refactored solution: isolation layer to the vendor software

- level of abstraction between application software and lower-level infrastructure
- provides software portability from underlying middleware and platform specific interfaces
- separation of infrastructures knowledge from application knowledge
- reduction of the risks and costs of infrastructure changes

Dependence on a proprietary architecture or tool set, making it hard to switch to another vendor

Analysis paralysis

General form:

- goal to achieve perfection and completeness of the analysis phase
- generation of very detailed models

Assumptions:

- everything about the problem can be known a priori
- detailed analysis can be successfully completed prior to coding
- analysis model will not be extended nor revisited during development

Symptoms and consequences:

- cost of analysis exceeds expectation without a predictable end point
- analysis documents no longer make sense to the domain experts

Typical causes:

- management (and often the customer) assumes a waterfall progression of phases
- goals in the analysis phase are not well defined

Refactored solution:

- assume that details of the problem are not all known a priori and its solution will be learned in the course of the development process
- use vertical prototyping -> agile methods
- allow incremental, iterative as well as adaptive development

Incremental means to “add onto something”
Iterative means to “re-do something”
Adaptive means to “react to changing requirements”
Spending excessive time in requirements elicitation and analysis

Code smell

Code smell pattern:

- problem in the source code (solution domain)
- solution: source code refactoring

Smell pattern:

- problem in the system model (application domain)
- solution: model refactoring

Examples of code smells: method too long, duplicated code, class too large, parameter list too long, feature envy, lazy class, speculative generality, refused bequest…

Replace inheritance with delegation

Problem: a subclass uses only part of a superclass interface or does not want to inherit data

- this results in source code that says one thing when your intention is something else (a confusion you should remove)

Refactored solution: replace inheritance with delegation

- make it clear that you are making only partial use of the delegated class

Replace conditional with polymorphism

Motivation: assume you are modeling the different speeds of a bird (in the code there will be different conditional cases for each type of bird). And if we want to add another case statement, we will have to add more conditions.

Solution:

- turn the class (Bird) into an abstract class
- create a subclass for each case label
- add an abstract method to the superclass for the method that contains the case statement
- create subclass methods that override the superclass method
- repeat for each branch of the case statement: move its body into the appropriate subclass method

Advantages of the refactored solution:

- easier to add new subclass
- no need to understand the other subclasses
- easier to understand the code

Replace error code with exception

Motivation:

- exceptions are better ways to deal with errors than error codes
- make the problem explicit instead of using a magic number
- using exceptions makes the error condition and exception handling easier to understand

Unchecked exception:

- unchecked exceptions do not need to be declared
- the class RuntimeException in Java and its subclasses are unchecked exceptions
- it is the responsibility of the caller to do any testing

Checked exception:

- checked exceptions need to be declared
- the class Exception in Java and any subclasses that are not subclasses of RuntimeException are checked exceptions
- it is the responsibility of the callee to do any testing

Good design:

- if you can choose, use unchecked exceptions because your code is more robust
- robustness is a good design goal

Steps to replace error code with exceptions:

1. decide if you want to place the responsibility of the exception handling with the caller or the callee
2. find all the callers of the method and adjust the method calls to deal with the exception
3. change the signature of the method to reflect the new usage, in particular remove the return code, as the error is now handled via the exception mechanism

Summary

- **Antipatterns** identify and categorize common mistakes in software practice and provide **refactored solution** to improve development, architecture and management
- **Golden hammer:** identical tools are used for a wide of diverse problems
  - **Refactored solution**: explore new technologies
- **Functional decomposition:** changing and adding new functionality is difficult, because the functionality is spread all over the system
  - **Refactored solution**: object oriented programming
- **Lava flow:** lots of unused or commented out code, undocumented complex, important looking code, evolving architecture
  - **Refactored solution**: ensure that system and object design (architecture) are implemented properly
- **The blob:** a single (god) class with a huge number of unrelated attributes and operations
  - **Refactored solution**: distribution of responsibilities into smaller classes based on object oriented analysis and design

- **Spaghetti code**: ad hoc structure which makes it hard to extend or optimize a model or code
- **Cut and paste programming**: excessive code duplication (clones) make the software expensive and difficult to maintain
- **Vendor lock-in**: dependence on a proprietary architecture or tool set, making it hard to switch to another vendor
- **Analysis paralysis**: spending excessive time in requirements elicitation and analysis
- **Code smells** are heuristics that indicates when to refactor, what specific technique to use and how the **refactored solution** can improve the problem

4. Testing Patterns

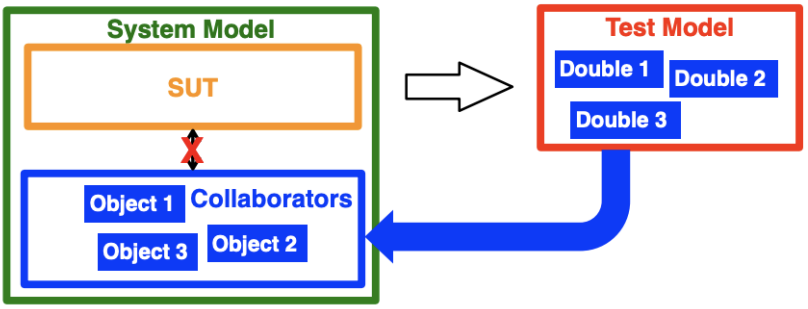
Introduction

The purpose of testing is the generation of failures, there are two ways to express the success of testing a component:

1. The test was successful because it did not generate a failure
2. The test was successful because it generated a failure

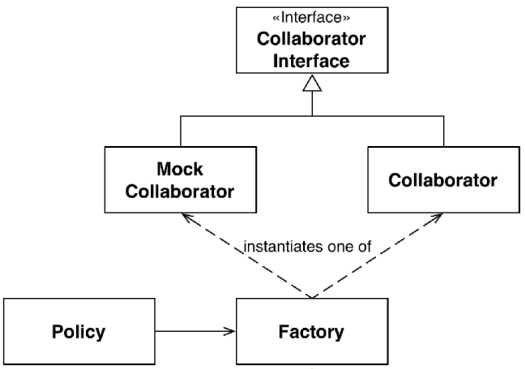
Object oriented test modeling

- Start with system model
- The system contains the **SUT** (system under test)
- The SUT does not exist in isolation, it interacts with other participating objects in the system model that are not yet implemented: **collaborators**
- The **test model** is derived from the **SUT**
- To be able to interact with **collaborators**, we add objects to the **test model**
- These are called **test doubles**

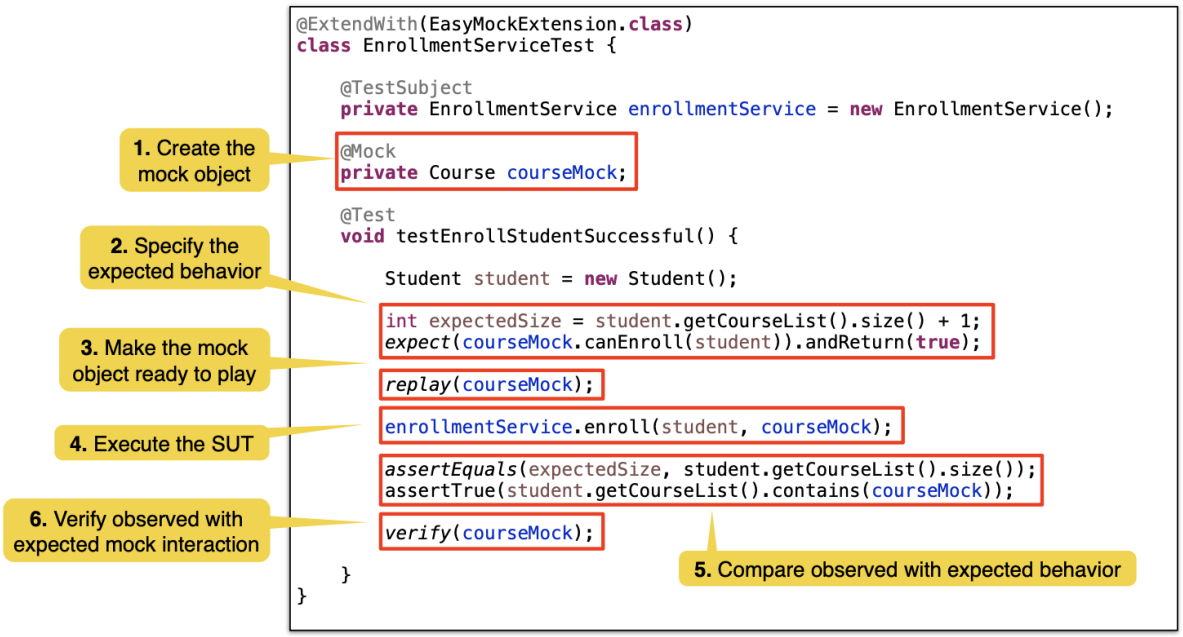


Mock object pattern

- A mock object replaces the behavior of a real object called the collaborator and returns hard-coded values
- A mock object can be created at startup time with the factory pattern
- Mock objects can be used for testing the state of individual objects as well as the interaction between objects
- The use of mock objects is based on the record play metaphor



EasyMock




```
@ExtendWith(EasyMockExtension.class)
class EnrollmentServiceTest {

    @TestSubject
    private EnrollmentService enrollmentService = new EnrollmentService();

    @Mock
    private Course courseMock;

    @Test
    void testEnrollStudentSuccessful() {

        Student student = new Student();

        int expectedSize = student.getCourseList().size() + 1;
        expect(courseMock.canEnroll(student)).andReturn(true);

        replay(courseMock);

        enrollmentService.enroll(student, courseMock);

        assertEquals(expectedSize, student.getCourseList().size());
        assertTrue(student.getCourseList().contains(courseMock));

        verify(courseMock);

    }
}
```

Nice vs default vs strict mocks

1. @Mock(MockType.NICE)
   Allows all method calls and returns appropriate empty values (0, null or false)

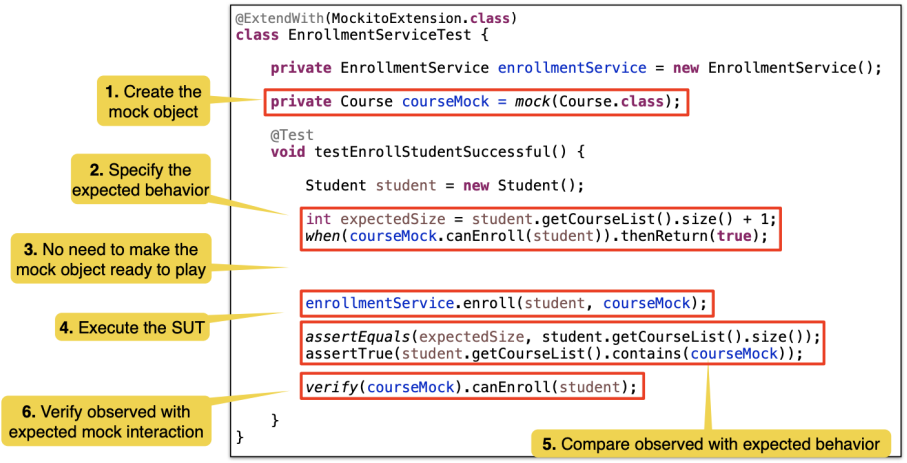
2. @Mock (default mock)
   Throws an AssertionError for unexpected method calls
   Does not check the order of the method calls

3. @Mock(MockType.STRICT)
   Checks the order of method calls

Mockito

Main features:

- mock() / @Mock -> full mocking
- spy() / @Spy -> partial mocking
- @InjectMocks -> automatically inject fields annotated with @Spy or @Mock
- verify() -> checks that methods were called with given arguments
    - can use flexible argument matching, for example any expression via the any()
    - capture what arguments were called using @Captor




```
@ExtendWith(MockitoExtension.class)
class EnrollmentServiceTest {

    private EnrollmentService enrollmentService = new EnrollmentService();

    private Course courseMock = mock(Course.class);

    @Test
    void testEnrollStudentSuccessful() {

        Student student = new Student();

        int expectedSize = student.getCourseList().size() + 1;
        when(courseMock.canEnroll(student)).thenReturn(true);


        enrollmentService.enroll(student, courseMock);

        assertEquals(expectedSize, student.getCourseList().size());
        assertTrue(student.getCourseList().contains(courseMock));

        verify(courseMock).canEnroll(student);

    }
}
```

Mock vs Spy

1. mock() / @Mock -> full mocking
    - optionally specify how it should behave via Answer/MockSettings
    - when() / given() to specify how a mock should behave
2. spy() / @Spy -> partial mocking
    - the behavior of single methods can be specified
    - real methods are invoked but still can be verified and stubbed

Argument matchers

- allow more flexible verification or mocking (e.g. any integer)
- if you are using argument matchers, all arguments have to be provided by matchers

Inject mocks into SUT code

@InjectMocks -> inject mock or spy fields into test objects automatically. Mockito will try to instantiate objects annotated with @Spy and @Mock and will instantiate @InjectMocks fields using constructor injection, setter injection or field injection.

Argument captors

Capture argument values for further assertions. Mockito verifies argument values in natural java style: by using equals() method. This is also a recommended way of matching arguments because it makes tests clean and simple. In some situations though, it is helpful to assert on certain arguments after the actual verification.

EasyMock vs Mockito

- For simple cases, both offer similar functionality:
  - specify the desired behavior of the collaborators (stubbing / mocking)
  - verify that the methods have been called (optionally in a specific order)
- Mockito has a clear separation between specifying the behavior and verifying that certain methods are invoked
- mockito offers spies (partial mocks)
- EasyMock always requires the replay() method in order to verify
- EasyMock distinguishes between nice, default and strict mocks
- EasyMock has difficulties with stubbing void methods

Test driven development (TDD)

Cycle:

1. Write test (before writing any other code)
2. Get the test to pass (with the most basic solution)
3. Optimize the design (make code more readable, eliminate “code smells”, make it pretty)
   Repeat

Why TDD? -> If tests are well written they serve as a concise documentation of what the implementation can and cannot do

Benefits:

- many teams report significant reductions in defect rates, at the cost of a moderate increase increase in initial development effort
- the same teams tend to report that these overheads significantly reduce the effort in the final phases of a project (for testing and documentation)
- practitioners report that TDD leads to improved design qualities in the code
- higher degree of “internal” or technical quality, for instance improving the metrics of cohesion and coupling

Limitations:

- it can be very tedious in the beginning
- it can be frustrating
- no system design upfront can lead to issues in the architecture

Reflection test pattern

Problem: there are some cases when it is necessary to test a private attribute:

- legacy code
- API of open source library
- bad design: refractor, but before refactoring you need to have a test

Solution: use reflection (versatile way of dynamically linking components and manipulation without need to hardcode the target classes).

Problems:

- can obscure what is going on in code
- maintenance of reflection code is difficult
- results in complex code
- performance can be slow

Four stage testing pattern

1. Setup: create the so-called test fixture with state and behavior that is needed to observe the SUT (such as using a mock object)
2. Exercise (running the test): interact with the SUT (calling a method)
3. Validate: look at the results with respect to state and/or behavior of the test and determine whether the observed outcome is equal to the expected outcome
4. Teardown: put the SUT back into the state before the test was executed, in particular, tear down any object that was instantiated in the test fixture

```
@ExtendWith(EasyMockExtension.class)
class EnrollmentServiceTest {
    private static EnrollmentService enrollmentService;
    private Student student;

    private Course courseMock = mock(Course.class);

    @BeforeEach
    void setUp() {
        enrollmentService = new EnrollmentService();
    }

    @Test
    void testEnrollmentStudentSuccessful() {
        student = new Student();

        enrollmentService.enroll(student, courseMock);

        assertEquals(...);
        assertTrue(...);
    }

    //...

    @AfterEach
    void tearDown() {
        reset(courseMock);
    }
}
```

1a) Setup collaborating objects

1b) Continued setup: Create instance of SUT

2. Run the test

3. Evaluate the results

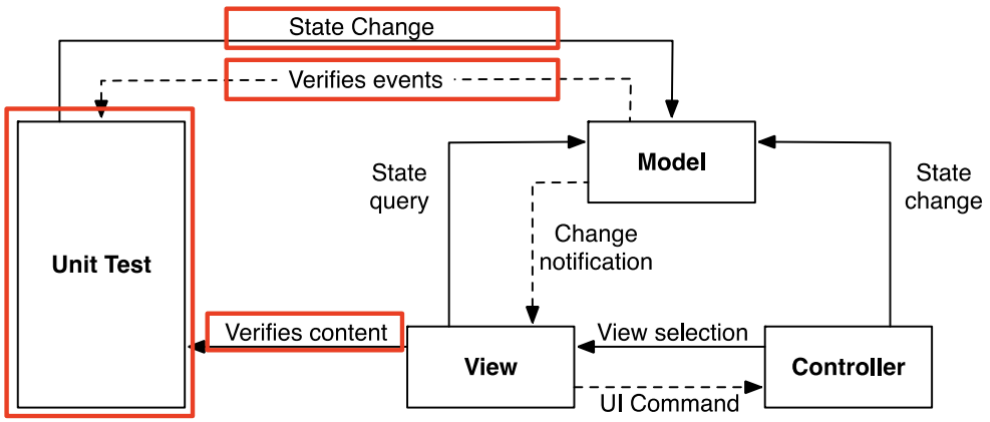
4. Tear down the test objects (e.g., reset all mocks)

Two testing patterns for MVC

View state test pattern

This pattern tests that whenever the model state changes, the view changes state appropriately (check if the view is updated when the model is changed).

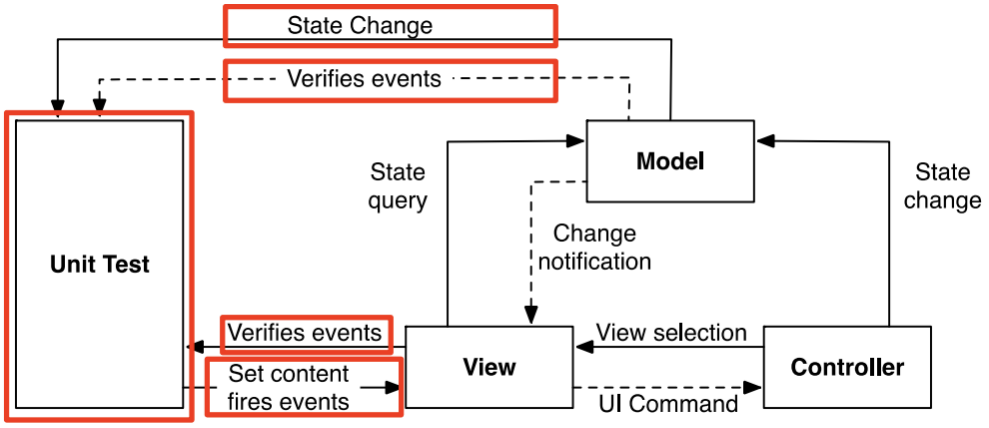
- This test exercises only half of the MVC pattern: the model change notifications to the view, and the view management of those events
- The controller is not tested in this test pattern



Model state test pattern

This pattern simulates user input by invoking a state change event such as “KeyUp”, “Click”, entering a test string “Max”, etc (checks if the model is updated correctly when the user tries to update the model via the view).
Validates that the model state is changed and the expected events fired correctly:

- may require some setup on the model itself, treats the controller as a black box
- model state can be inspected to determine if the controller is managing the model state correctly



Dependency injection

Goal: reduce the high coupling as much as possible

Guice

Uses the concept of a Module to bind a subclass implementation to an interface:

- it provides an abstract class AbstractModule with a protected method configure()
- AbstractModule can have subclasses called Guice modules

Modules overwrite the configure() method to bind an implementation to a specification
Two Guice modules are often used: ProductionModule and TestModule

```
public class InventorySystem {
    private static String TALISKER = "Talisker";
    private int totalStock;
    @Inject private Warehouse warehouse;
    public void addToWarehouse(String item, int amount) {
        warehouse.add(item, amount); totalStock += amount;
    }
    public int getTotalStock() {
        return totalStock;
    }
    public boolean processOrder(Order order) {
        order.fill(warehouse);
        return order.isFilled();
    }
    public static void main(String[] args) {
        Injector injector = Guice.createInjector(new ProductionModule());
        InventorySystem inventorySystem = injector.getInstance(InventorySystem.class);
        inventorySystem.addToWarehouse(TALISKER, 50);
        boolean order1success = inventorySystem.processOrder(new OrderImpl(TALISKER, 50));
        boolean order2success = inventorySystem.processOrder(new OrderImpl(TALISKER, 51));
        System.out.println("Order1 succeeded? " + order1success + " - Order2 succeeded?" + order2success);
    }
}
```

Notice there is no **new** here anymore!

An **injector** is created: it allows to specify bindings of implementations to interfaces → In this case it injects a **ProductionModule**

Two changes to the **Client**

Guice module for the production code

```
public class ProductionModule extends AbstractModule {

    @Override
    protected void configure() {
        bind(Warehouse.class).to(WarehouseImpl.class);
    }
}
```

Guice module for the unit test

```
public class TestModule extends AbstractModule {

    @Override
    protected void configure() {
        Warehouse warehouseMock = EasyMock.createMock(Warehouse.class);
        bind(Warehouse.class).toInstance(warehouseMock);
    }
}
```

4 steps to use Guice:

1. Tell GUice where to inject using @Inject annotation:
   - Constructor injection
   - Method injection
   - Field injection
2. Create a Module to define the Binding
3. Instantiate an Injector and tell it which Module to use
4. Instantiate an instance of the class needing the injection

```
class InventorySystemTest {
    private static String TALISKER = "Talisker";
    private InventorySystem inventorySystem;
    private Injector injector;

    @BeforeEach
    void setUp() throws Exception {
        injector = Guice.createInjector(new TestModule());
        inventorySystem = injector.getInstance(InventorySystem.class);
    }

    @Test
    void addToWarehouse() {
        Warehouse warehouseMock = injector.getInstance(Warehouse.class);
        warehouseMock.add(TALISKER, 50);
        expectLastCall().andVoid();
        replay(warehouseMock);
        inventorySystem.addToWarehouse(TALISKER, 50);
        assertEquals(inventorySystem.getTotalStock(), 50);
        verify(warehouseMock);
    }
}
```

Tell **Guice** to create an **injector** using **TestModule** that binds **Warehouse** to a **WarehouseMock**

Let **Guice** instantiate the **InventorySystem** and the **Warehouse** in it

Retrieve the **WarehouseMock** from the **injector** to use it during the test

Specified behavior in the test case

EasyMock validation

Inversion of control

- Control of objects or portions of a program is transferred to a container or framework
- The framework takes control of the flow of a program and makes calls to the program's custom code
- To enable this, frameworks use abstraction with additional behavior built in
- If you want to add your own behavior, you need to extend the classes of the framework or plugin your own classes

Advantages:

- Decoupling the execution of a task from its implementation
- Making it easier to switch between different implementations
- Greater modularity of a program
- Greater ease in testing a program by isolating a component or mocking its dependencies and allowing components to communicate through contracts

Design principle to make:

- application easier to develop
- code less coupled
- code easier to test

Spring

Uses an application context for inversion of control. 3 possibilities to use dependency injection:

1. Constructor injection
2. Method injection (setter)
3. Field injection (attribute)

Autowiring dependencies

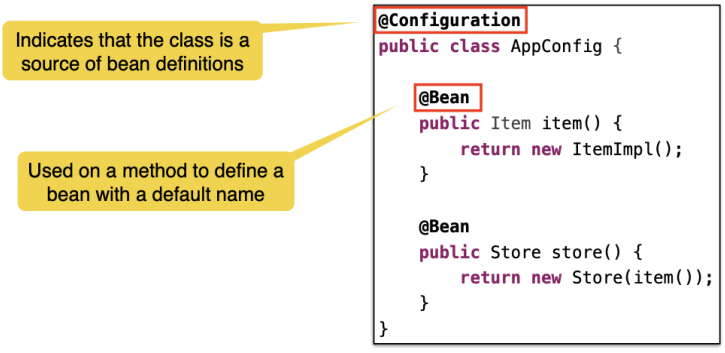
Lazy instantiations

Spring bean

- an object that the Spring container instantiates, assembles and manages
- define beans for Service layer objects, Data access objects, Presentation objects, etc
- should not configure fine grained domain (entity) objects in the container

Bean definition in the configuration

For a bean with the default **singleton scope**, Spring first checks if a cached instance of the bean already exists and only creates a new one if it doesn't. If we're using the **prototype scope**, the container returns a new bean instance for each method call



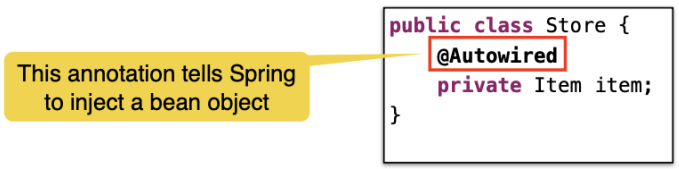
Constructor injection
In Spring, no annotations are needed

```
private final Item item;

public Store(Item item) {
    this.item = item;
}
```

Method injection
The container will call setter methods of our class, after instantiating the object (possibly with the default constructor with no arguments)

Field injection
Inject dependencies by marking them with an @Autowired annotation

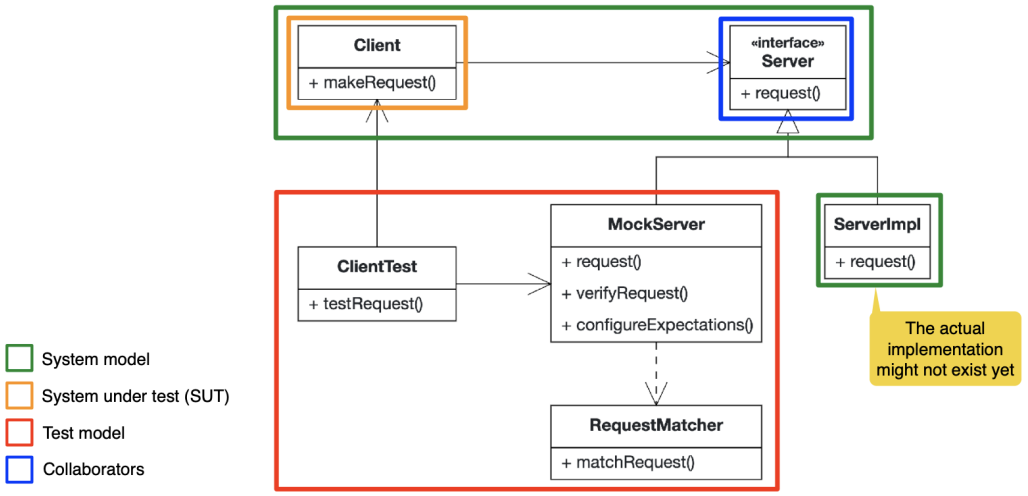


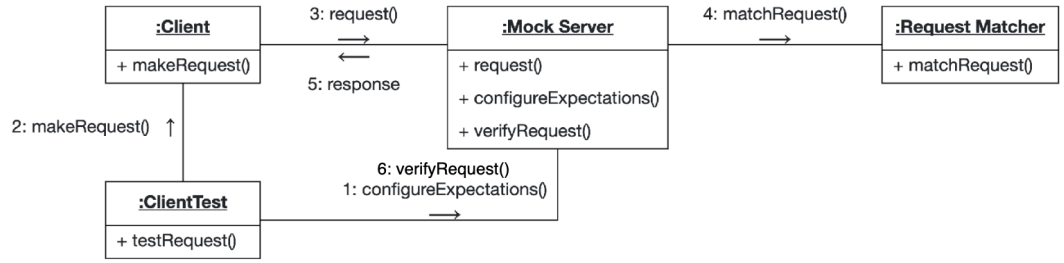
Autowiring dependencies
Allows the Spring container to automatically resolve dependencies between collaborating beans by inspecting the beans that have been defined. There are 4 modes of autowiring a bean:

1. no: the default value - this means no autowiring is used for the bean and we have to explicitly name the dependencies
2. byName - autowiring is done based on the name of the property, therefore Spring will look for a bean with the same name as the property that needs to be set
3. byType - similar to the byName autowiring, only based on the type of the property
4. constructor - autowiring is done based on constructor arguments

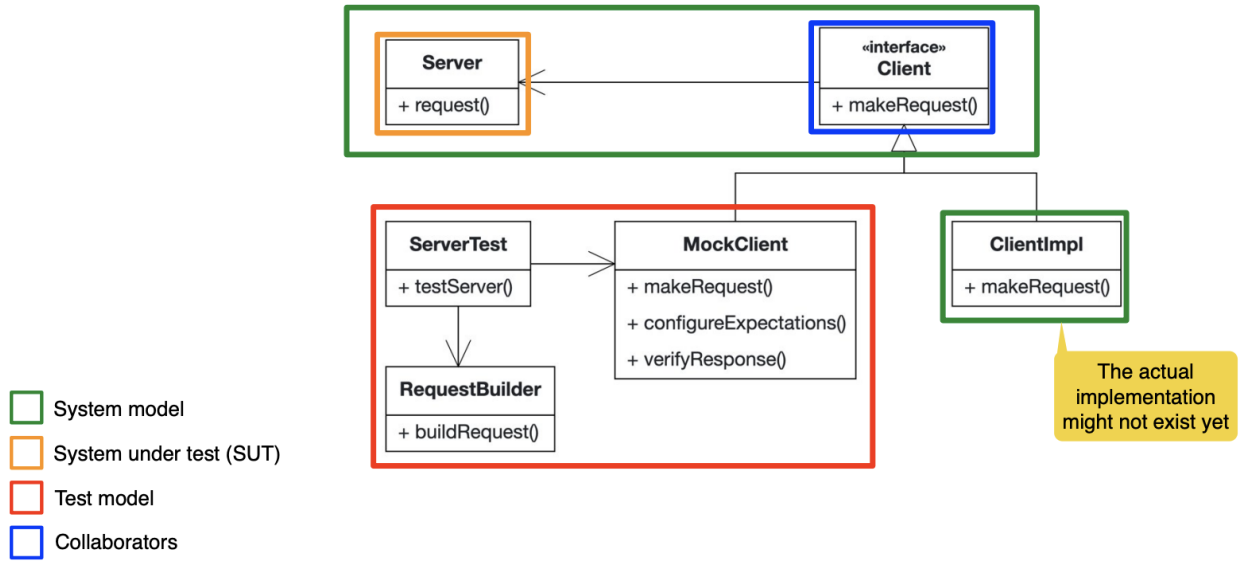
Test client server applications

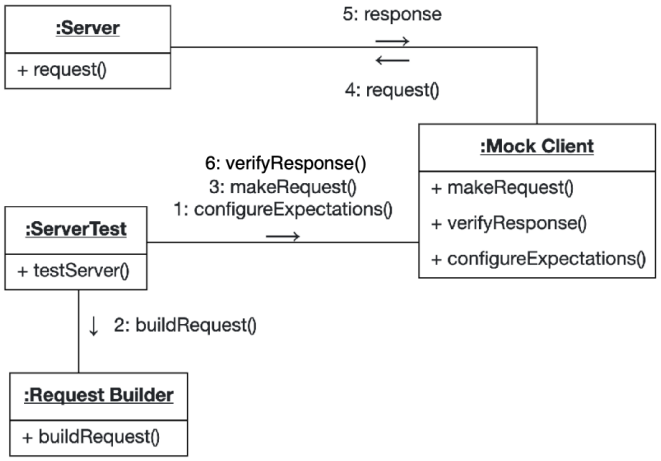
Test clients: In order to write integration client tests, we need to mock the remote service.





Test servers: In order to write integration server tests, we need to mock the client.





Example of server test with Spring MockMvc

```
@WebMvcTest
@ContextConfiguration(classes = RestController.class)
class RestControllerTests {
    @Autowired
    private MockMvc mockMvc;
    @Autowired
    private ObjectMapper objectMapper;

    @Test
    void testGetAllObjects() throws Exception {
        List<Object> expectedResultList = new ArrayList<Object>(); // add the expected objects here
        ResultActions request = mockMvc.perform(get("/objects")).andDo(print()).andExpect(status().isOk());

        String response = request.andReturn().getResponse().getContentAsString();


       List<Object> actualResultList =
                Arrays.stream(objectMapper.readValue(response, Object[].class)).collect(Collectors.toList());

        assertEquals(expectedResultList, actualResultList, "Not all objects have been returned");
    }
}
```

Allows to mock the client

Performs a GET request on the actual server and checks that the response code is 200 (OK)

Response as (json) string

Map response to Java objects

Compare expected and actual response

Summary

- Mock objects are test doubles that mimic the behavior of the real object
- The **mock object pattern** enables us to test state and behavior
  - **EasyMock**: simple framework with some limitations
  - **Mockito**: powerful and popular open source framework with many advanced features
- Mock objects are a way to create “self-contained” unit tests (i.e. without much interaction with the rest of the objects in the system model)
- Frameworks for mock objects are available in major programming languages (Java, C++, Perl, Ruby, …)
- **Reflection test pattern:** access or modify private values for testing purposes
- **Testing MVC**: check the view or model state

- Simple unit tests allow to test the state of an object or a subsystem
- The **mock object pattern** allows to unit test behavior
- We can achieve low coupling between the system and the tests with **dependency injection**
- Successful testing is difficult and there are many obstacles
- Follow the **best practices** and **modern test principles**
- There are many examples how to write integration tests for complex setups of distributed systems
- Are there other patterns applicable to testing?
  - Meszaros describes 68 testing patterns: Gerard Meszaros: xUnit Test Patterns – Refactoring Test Code. Martin Fowler Signature Series, Addison-Wesley, 2007

5. Microservice Patterns

Introduction

The microservice architecture is an architectural style that functionally decomposes an application into a set of services.

Benefits:

- Services are small and easily maintained
- Services are independently deployable
- Services are independently scalable
- They enable the continuous delivery and deployment of large, complex applications
- They enable teams to be autonomous
- Allows easy experimenting and adoption of new technologies
- Better fault isolation

Challenges:

- Deciding when to adopt the microservice architecture is difficult
- Finding the right set of services is challenging
- Distributed systems are complex, which makes development, testing and deployment difficult
- Deploying features that span multiple services requires careful coordination

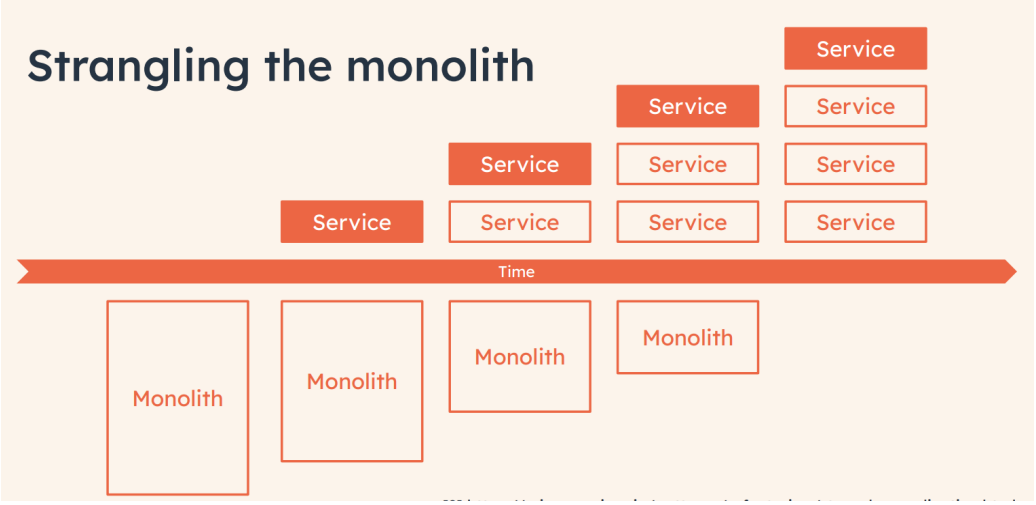
Strangler Pattern

Problem:

- Your application is growing (more traffic, more developers, …)
- Your code-base grows and becomes complex
- Developers are stepping on each other's toes
- So we want to transfer to microservices, without stopping operations or development

Solution:

- Break out small flows of monolith into a separate and new service
- Over time, there are more and more services and the monolith will shrink until it's obsolete



Decentralized ID Generation

Problem:

- ID generation is handled by our database's auto_increment
- We outgrew our single DB and now we need to split it up
- We still want to have unique IDs

Solution 1 - UUID

Pros:

- Generating UUID is simple. No coordination between servers is needed
- Fast

Cons:

- IDs are 128 bits long
- IDs do not go up with time
- IDs could be non-numeric

Solution 2 - Ticket Server

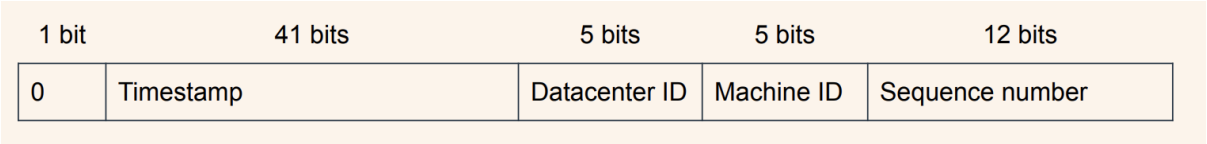
Pros:

- Numeric IDs
- Easy to implement
- Works for small to medium-scale applications

Cons:

- Single point of failure. To avoid that you can setup multiple ticket servers, but that requires synchronization

Solution 3 - Twitter Snowflake



1 bit	41 bits	5 bits	5 bits	12 bits
0	Timestamp	Datacenter ID	Machine ID	Sequence number

- Each ticket server can generate IDs independently
- Take the lowes 41 bits of a timestamp
- On startup, a ticket server receives a datacenter and machine ID
- Sequence number increments with every ID. It is reset every millisecond

Pros:

- Numeric IDs
- No single point of failure
- No synchronization

Cons:

- Harder to setup
- Timestamp only has 41 bits

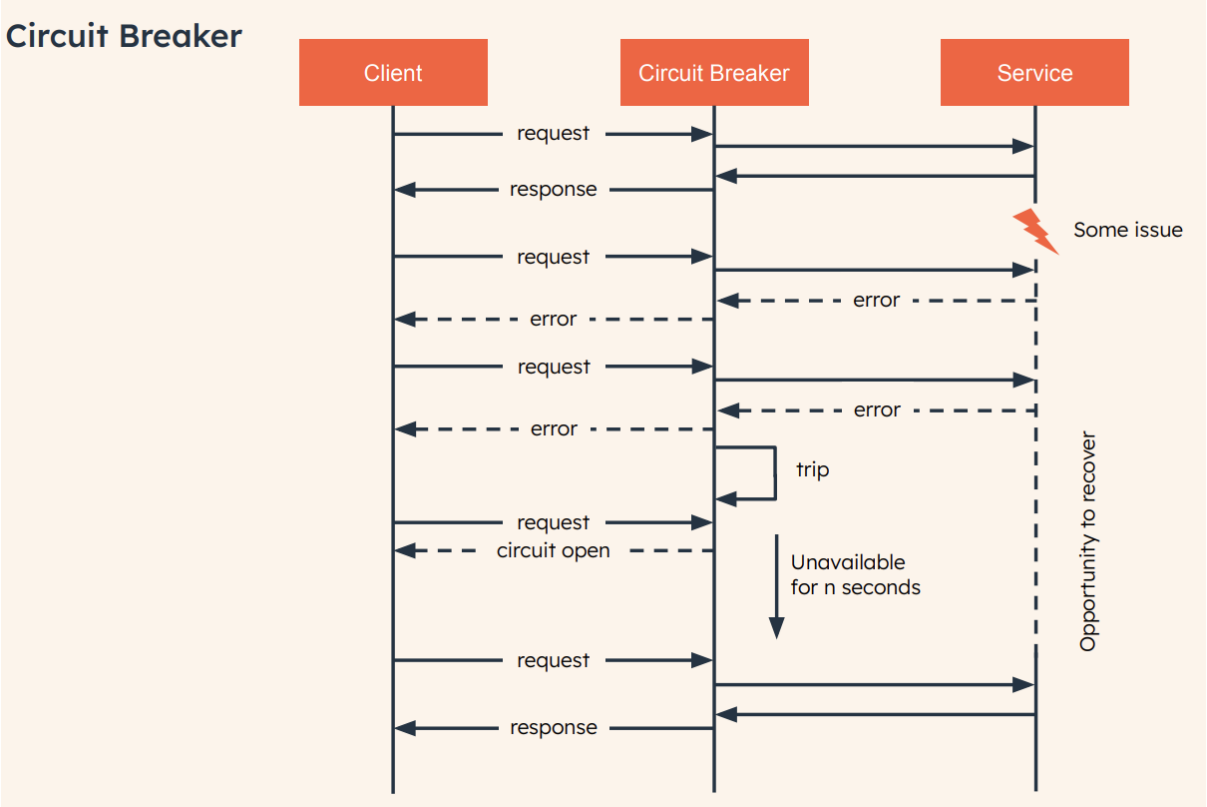
Circuit Breaker

Problem:

- Our service returns an error for 1 endpoint (e.g. we introduced a bug)
- There is a danger that consumers of our service drain our resources due to retries
- Other endpoints work just fine and they shouldn't get affected or crash

Solution:

- Proxy/monitor HTTP responses
- If to many errors appear consecutively, the circuit breaker trips and all subsequent requests will automatically fail for a period of time
- After the period, we retry if the service recovered again



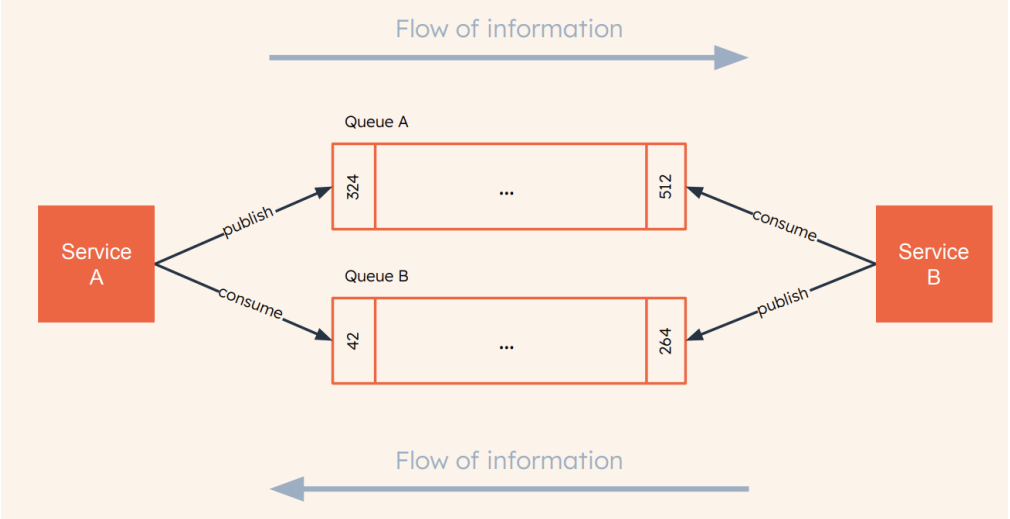
Asynchronous Messaging

Problem: Synchronous communication leads to tight runtime coupling

- Both services must be available for the duration of the request
- Services that have a high latency by design (e.g. neural networks) negatively impact your latency
- Peaks with high load saturate your services and threads

Solution:

- Asynchronous communication
- Usually achieved via message queues (e.g. Kafka, RabbitMQ) aka Observer Pattern
- You can also use a database



Summary