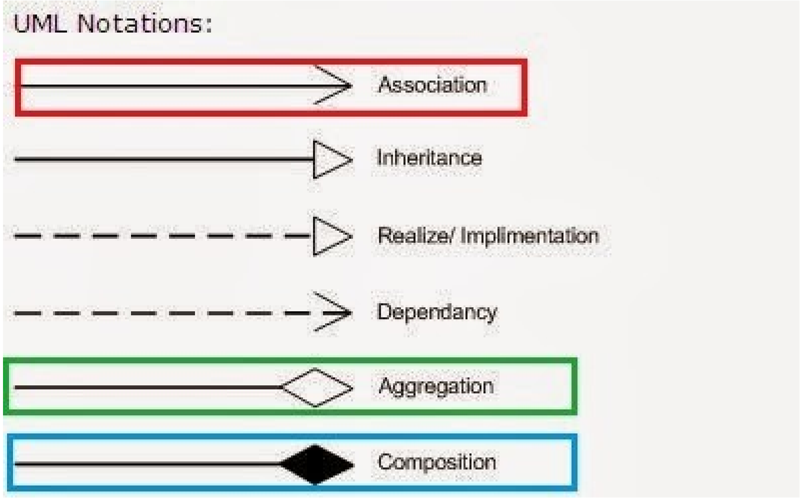
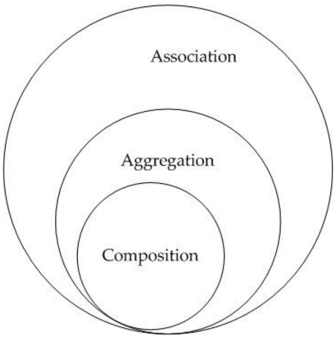
# Question styles

- Definition and why we need, what steps to implement
- UML descriptions
- Comparison of each other advantages/disadvantages
- Refactor with antipattern style
- Given case implement pattern/antipattern/test cases (junit and juice)

# UML DIAGRAMS





**Association** - I have a relationship with an object. `Foo` uses `Bar`

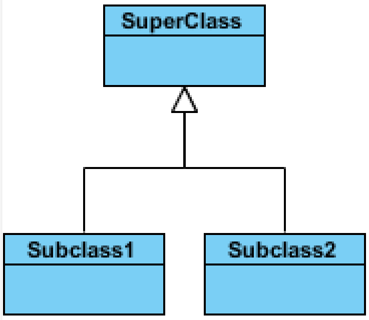
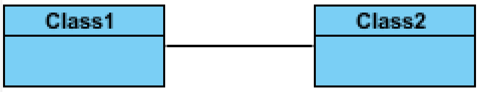
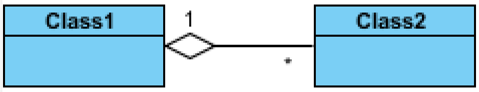
```
public class Foo {
    void Baz(Bar bar) {
    }
};
```

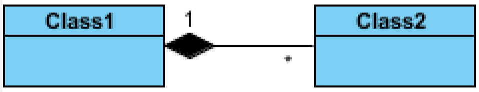
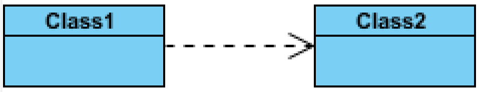
**Composition** - I own an object and I am responsible for its lifetime. When `Foo` dies, so does `Bar`

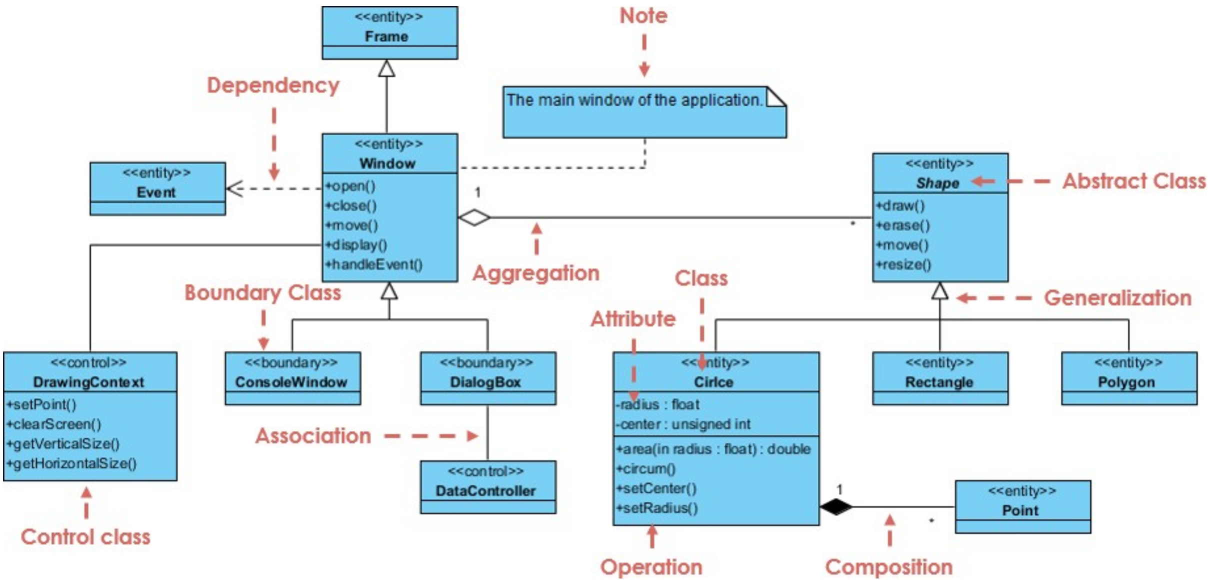
```
public class Foo {
    private Bar bar = new Bar();
}
```

**Aggregation** - I have an object which I've borrowed from someone else. When `Foo` dies, `Bar` may live on.

```
public class Foo {
    private Bar bar;
    Foo(Bar bar) {
       this.bar = bar;
    }
}
```

| Relationship Type | Graphical Representation |
|---|---|
| Inheritance (or Generalization):<br>• Represents an "is-a" relationship.<br>• An abstract class name is shown in italics.<br>• SubClass1 and SubClass2 are specializations of Super Class.<br>• A solid line with a hollow arrowhead that point from the child to the parent class |  |
| Simple Association:<br>• A structural link between two peer classes.<br>• There is an association between Class1 and Class2<br>• A solid line connecting two classes |  |
| Aggregation:<br>A special type of association. It represents a "part of" relationship.<br>• Class2 is part of Class1.<br>• Many instances (denoted by the *) of Class2 can be associated with Class1.<br>• Objects of Class1 and Class2 have separate |  |

| | |
|---|---|
| **lifetimes.**<br>• **A solid line with an unfilled diamond at the association end connected to the class of composite** | |
| **Composition:**<br>A special type of aggregation where parts are destroyed when the whole is destroyed.<br>• Objects of Class2 live and die with Class1.<br>• Class2 cannot stand by itself.<br>• A solid line with a filled diamond at the association connected to the class of composite |  |
| **Dependency:**<br>• **Exists between two classes if the changes to the definition of one may cause changes to the other (but not the other way around).**<br>• **Class1 depends on Class2**<br>• **A dashed line with an open arrow** |  |



# Foundations

- Abstractions and Types
    - Abstraction can be of two types, namely, **data abstraction** and **control abstraction**. Data abstraction means hiding the details about the data and control abstraction means hiding the implementation details.
- Inheritance
    - In object-oriented programming, inheritance enables new objects to take on the properties of existing objects. A class that is used as the basis for inheritance is called a superclass or base class. A class that inherits from a superclass is called a subclass or derived class. The terms parent class and child class are also acceptable terms to use respectively. A child inherits visible properties and methods from its parent while adding additional properties and methods of its own.
- Information hiding
    - Private/protected/public
- Coupling and Cohesion
    - As for **coupling**, it refers to how related or dependent two classes/modules are toward each other. For low coupled classes, changing something major in one class should not affect the other. High coupling would make it difficult to change and

maintain your code; since classes are closely knit together, making a change could require an entire system revamp.
    - Good software design has **high cohesion** and **low coupling**.

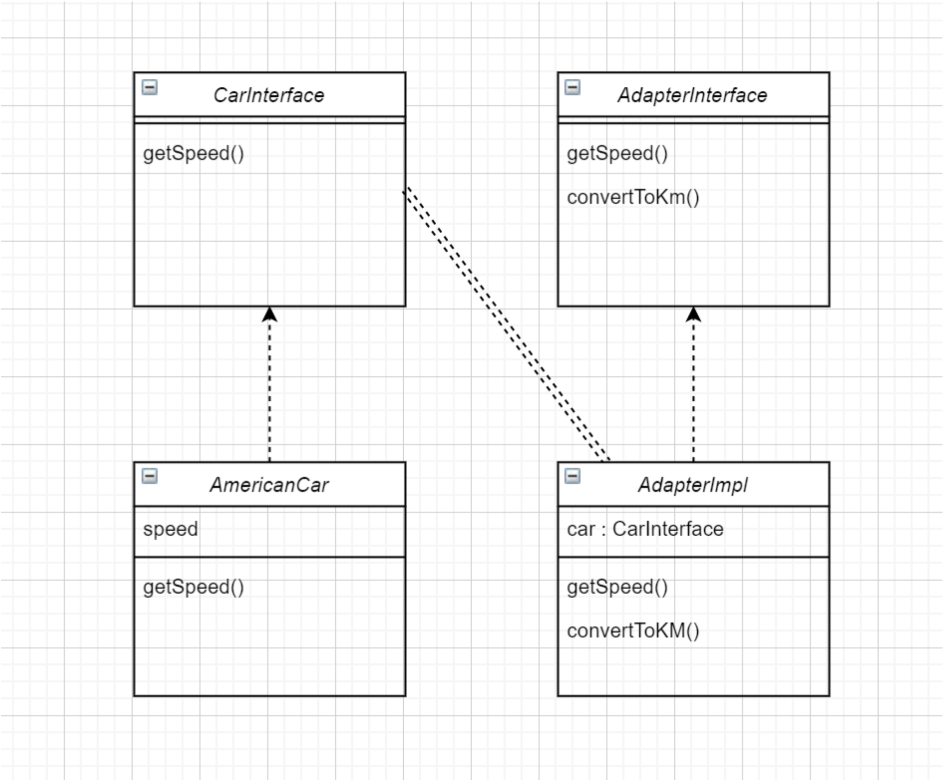
- Polymorphism
    - Method overloading/overriding

# Design Patterns

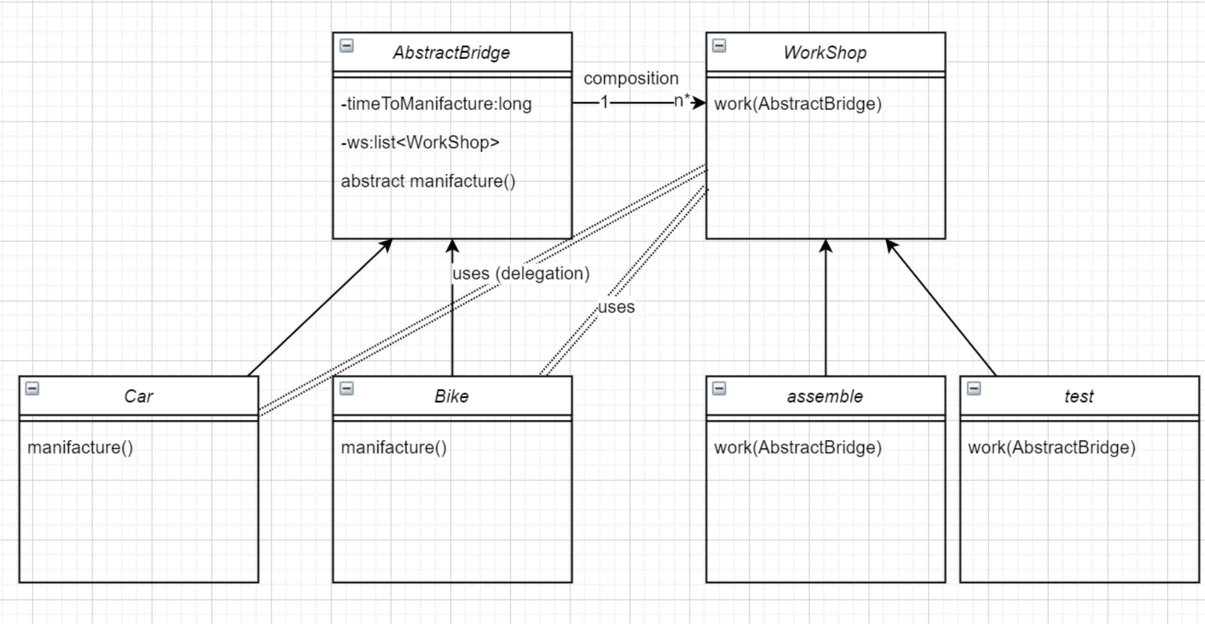
## Structural Patterns

- Adapter
    - Why ?
        - Use the Adapter class when you want to use some existing class, but its interface isn’t compatible with the rest of yourcode
    - How ?
        - Create a adapter class has(composition) of existing class. Override the same method you want to change, get result from the existing class change it according your needs and return.
    - Advantages
        - Single responsibility, no need to change existing code
    - Disadvantages
        - Increase complexity (new interface and classes)
    - Example
        - American Car (uses mph speed) / Adapter Class(has American car as composition and overrides its getSpeed()) method by converting it to km.

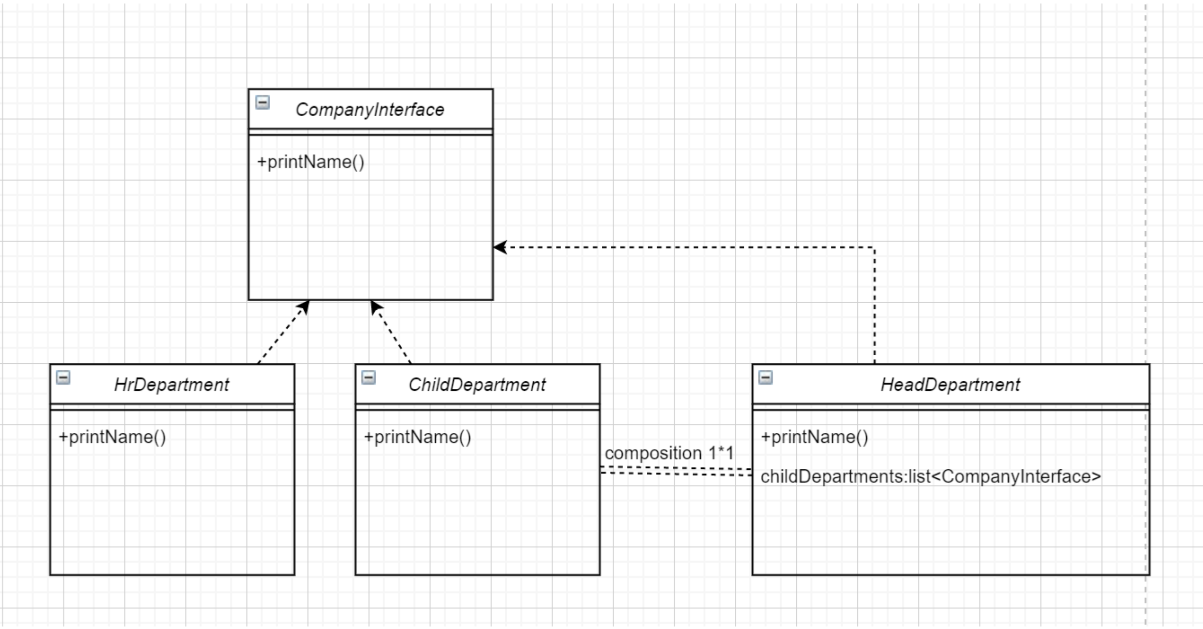
        - UML:



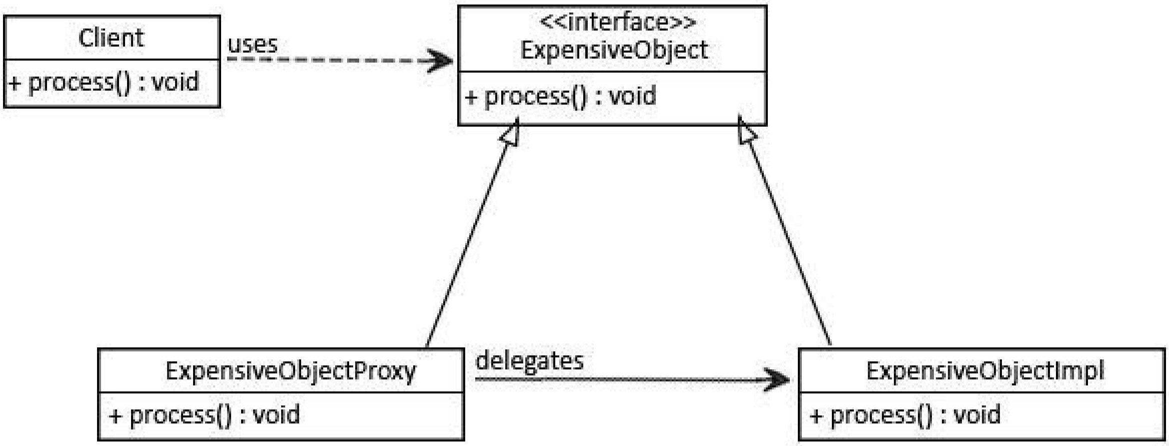
- Bridge
    - Why?
        - You split the monolithic class into several class hierarchies. After this, you abstract a bridge that makes delegation between those hierarchies.
    - How
        - Create a bridge abstract and store as composition other class inside then concrete classes of bridge uses stored composition to delegate specific task.
    - Adv
    - Disadvantages
    - Example
        - Car/bike/abstract bridge (manufacture method) stores manufacturer abstract contrece classes and delegates to it with manufacture method.



- Composite
  - Why
    - Use the Composite pattern when you have to implement a tree-like objectstructure. Like a companies departments which might be standalone or contain other departments inside.
  - How
    - Create a interface according to concrete classes needs, create concrete classes either a leaf(only implements base) or a composite(implements base and has other leafs inside)
  - Adv/Dis adv
  - Example
    - Company department interface with printName method, finance print its name, hr prints its name, head department prints its name and has other leafs inside as list

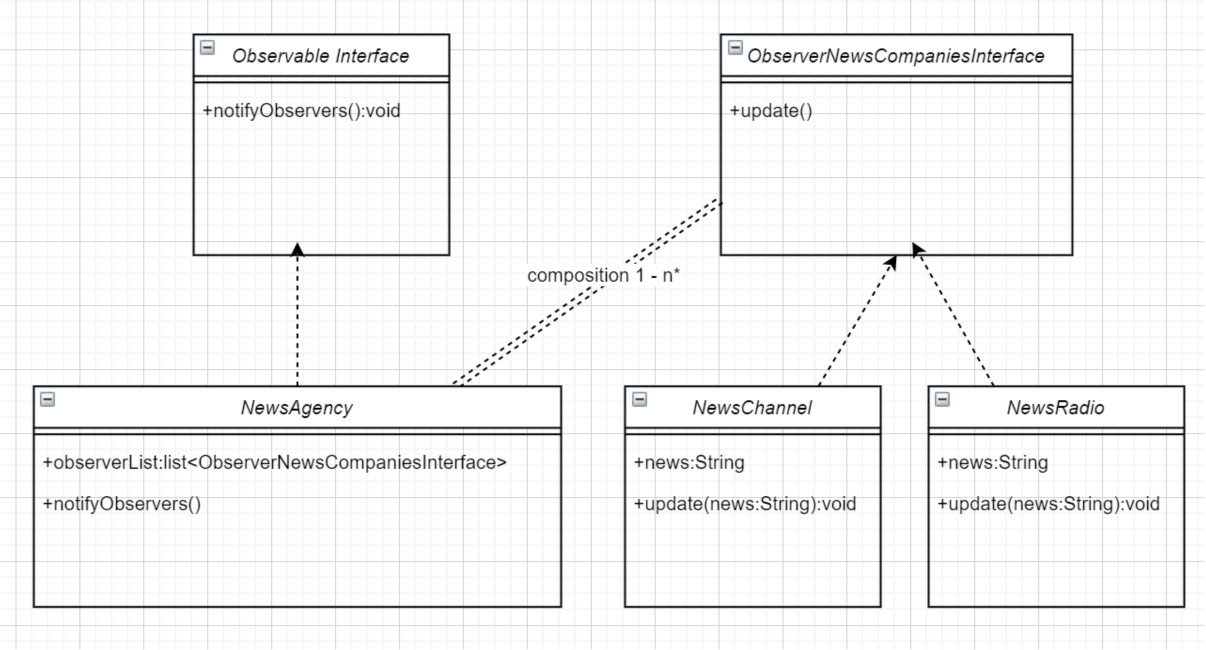


- Proxy
  - Why ?
    - When we want a simplified version of a complex or heavy object.
    - When we want to change a behavior of a class.
  - How ?
    - Have an interface and concrete actual and proxy objects. Replace it with a lightweight version and keep it that heavy weight concrete class inside lightweight and delegate regarding process whenever required.
  - Adv/dis
  - Example



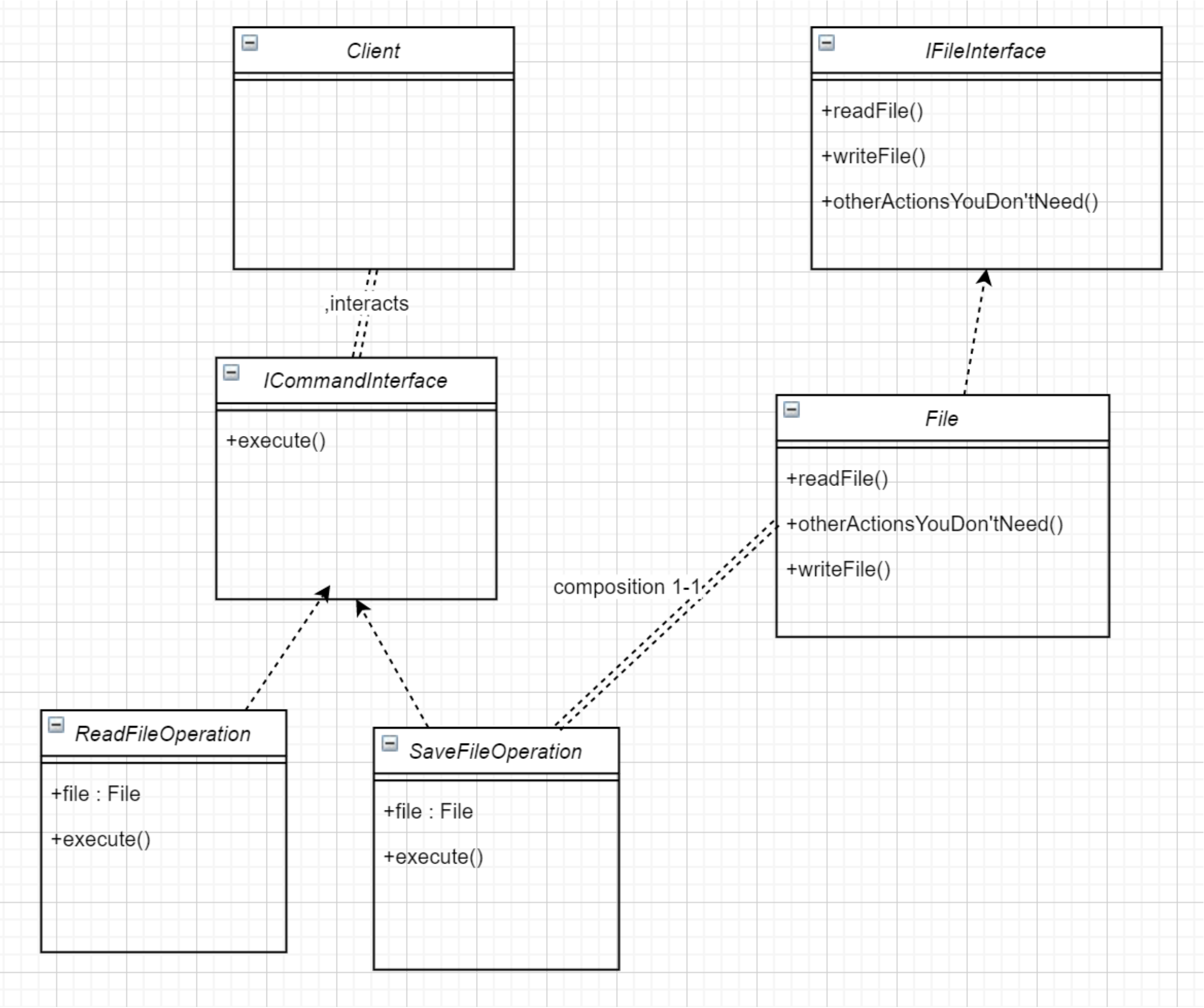
# Behavioral Pattern

- Observer
  - Why ?
    - use Observer pattern when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically.
  - How ?
    - Create Observable interface put “notifyObservers” method, create a concrete class(the one who has observers inside as list) from it. Create observer interface put update method. Concrete observers will be updated when change happens.
  - Adv / dis
  - Example
    1. News agency when it retrieves a news it passes the observers that it keeps inside.



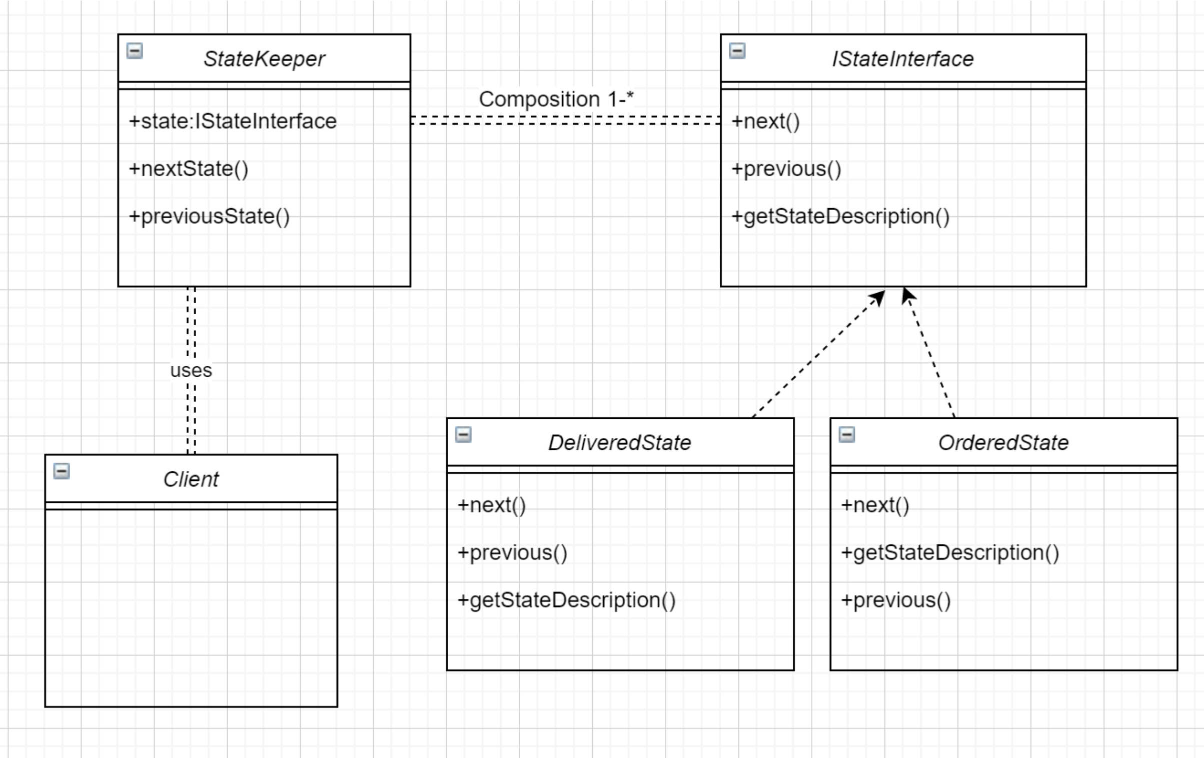
- Command
  - Why?
    - It decouples client and actual implementor. Delegates required actions the corresponding responsible. So it keeps an abstraction and encapsulation to command class.
  - How?
    - Create an interface which contains execute() method. Concrete classes of this interface are the interface which are open to client and they store actual request object. Client stores its request inside those concrete classes and only calls its execute method.
  - Adv/dis
  - Example

- There is a text file object. You provide it to command interface concrete classes according to your need, you don't need to know about how command implements details. You just pass it.

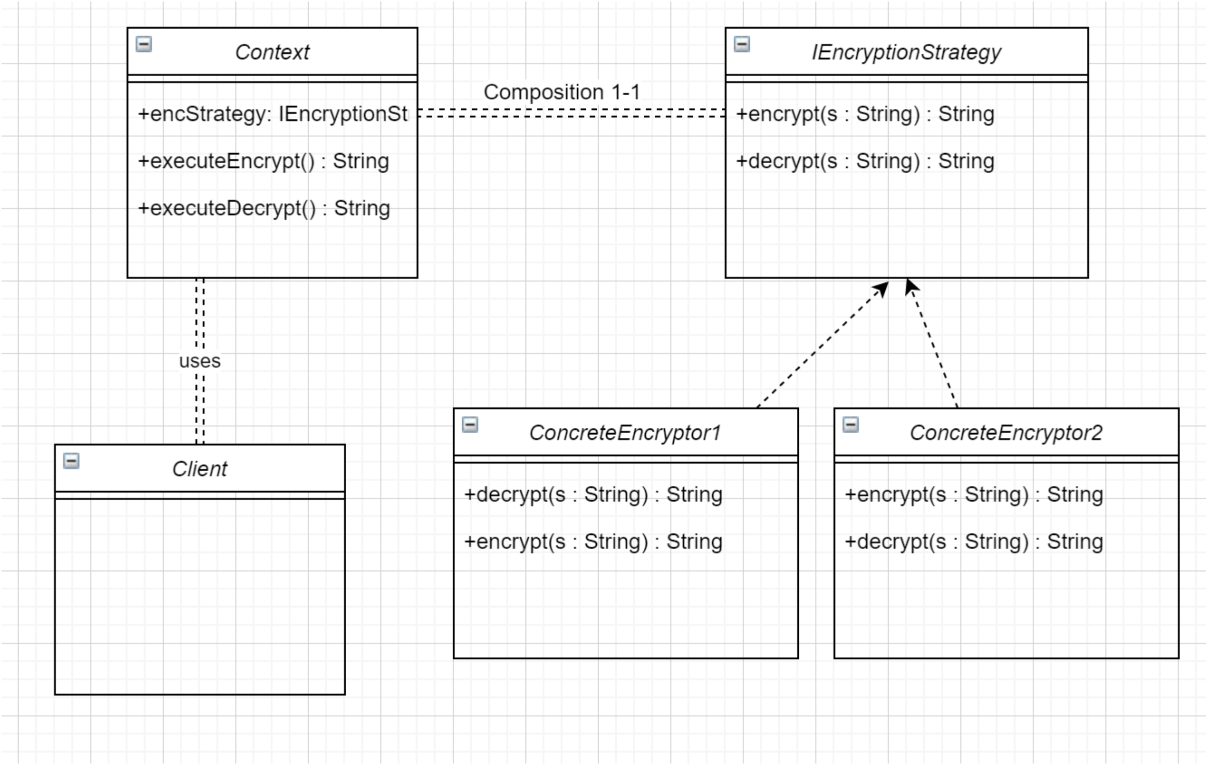


- State
  - What ?
    - Lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.
  - How?
    - Create an interface for states, and create concrete classes. Keep interface as composition in the class that will keep states and behave accordingly. Call that class from client.
  - When ?
    - Use the State pattern when you have an object that behaves differently depending on its current state, the number of states is enormous, and the state-specific code changes frequently.
    - To get rid of so many if/else statements
  - Adv/DisAdv
  - Example

        - Delivery system. StateKeeper class keeps composition of concrete classes of StateInterface which has methods next(StateKeeper),previous(StateKeeper) so that it changes StateKeeper StateInterface fields concrete class.



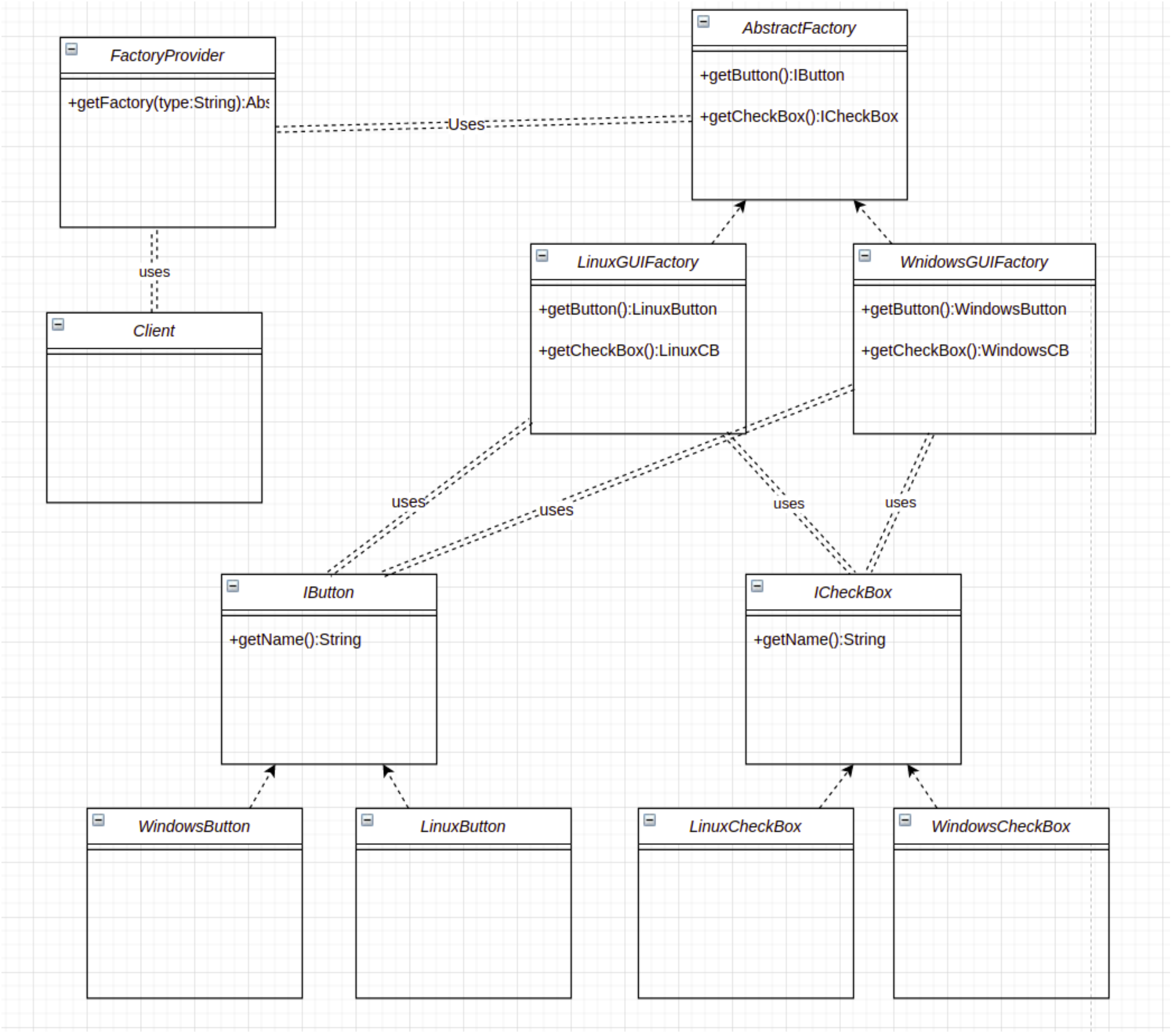
- Strategy
    - What
        - Essentially, the strategy pattern allows us to change the behavior of an algorithm at runtime.
    - When
        - Use the Strategy pattern when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.
    - How
        - Create a class that keeps strategy interface as composition. ConcreteStrategy classes of strategy interface will be used according to that composition interchangebly. By calling its execute method.
    - Adv/DisAdv
    - Example
        - You want to encrypt data by using IEncrypt interfaces concrete classes. You can create a Context class that keeps IEncrypt as composition, and there is a encrypt and decrypt method will delegate concrete classes of IEncrypt. Client only communicates with context.


Context
+encStrategy: IEncryptionSt
+executeEncrypt() : String
+executeDecrypt() : String
Composition 1-1
IEncryptionStrategy
+encrypt(s : String) : String
+decrypt(s : String) : String
uses
Client
ConcreteEncryptor1
+decrypt(s : String) : String
+encrypt(s : String) : String
ConcreteEncryptor2
+encrypt(s : String) : String
+decrypt(s : String) : String

# Creational Pattern

- Abstract factory
  - What?
    - Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.
  - When ?
    - Use the Abstract Factory when your code needs to work with various families of related products, but you don’t want it to depend on the concrete classes of those products—they might be unknown beforehand or you simply want to allow for future extensibility.
  - How ?
    - First Create AbstractFactory(createButton(),createCheckBox()) interface which forces actual factory classes to create concrete classes. Ask to FactoryProvider(getFactory(“button”)) to get actual factories which implement abstractFactories methods. Actual ones returns asked concrete class type.

- Example ?

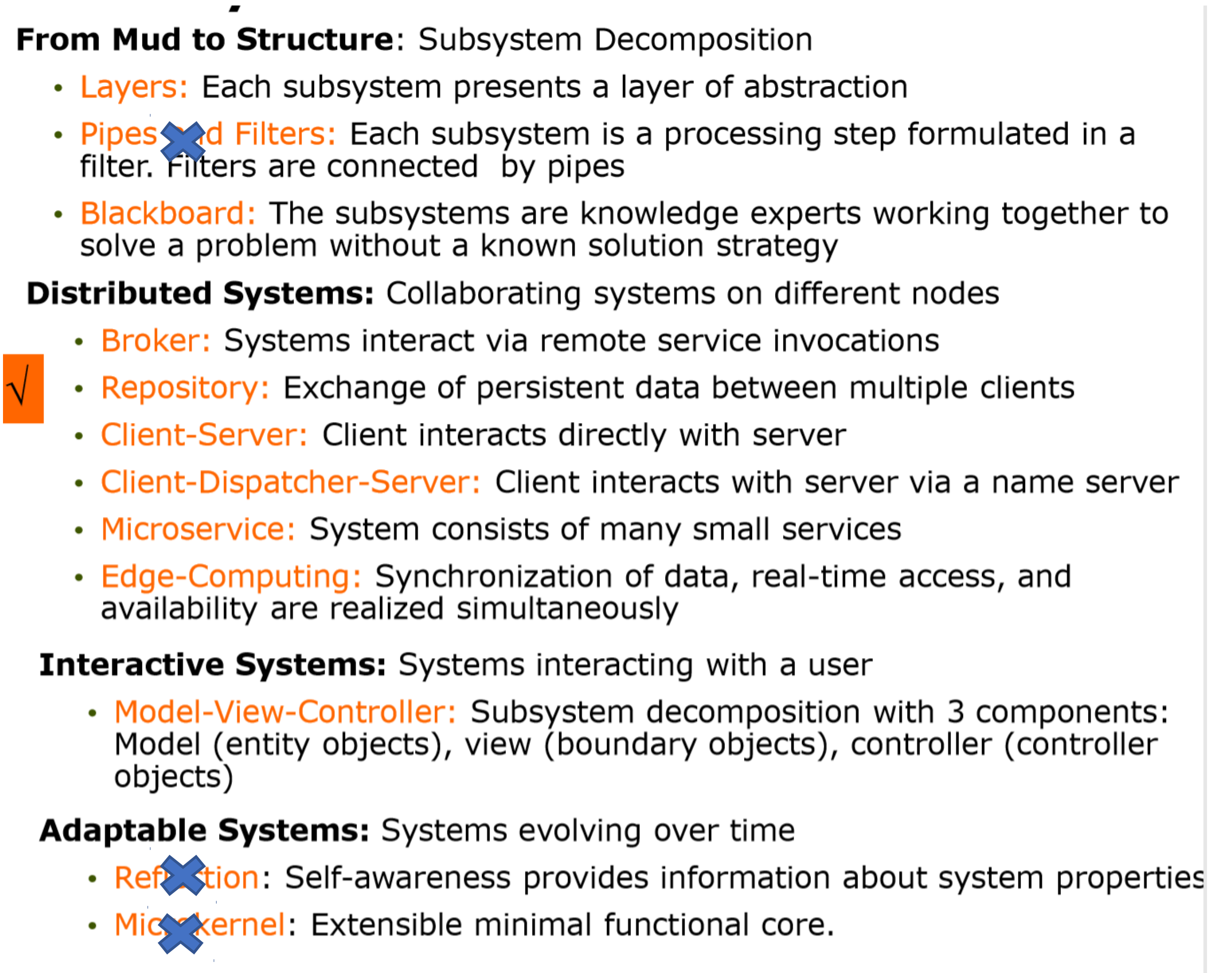


## Pattern Templates

- Alexander’s Template
  - ‘Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.’
  - Structures cannot be separated from the problems they are solving
  - A pattern is a rule that expresses a relation between
    - Context
    - a problem and
    - a solution
- GoF Template
  -
- Buschmann Template

- Antipattern Template (Brown)

# Architectural Patterns



**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. Filters are connected by pipes
- Blackboard: The subsystems are knowledge experts working together to solve a problem without a known solution strategy

**Distributed Systems:** Collaborating systems on different nodes

- Broker: Systems interact via remote service invocations
- √ Repository: Exchange of persistent data between multiple clients
- Client-Server: Client interacts directly with server
- Client-Dispatcher-Server: Client interacts with server via a name server
- Microservice: System consists of many small services
- Edge-Computing: Synchronization of data, real-time access, and availability are realized simultaneously

**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.

## From mud to structure

- Layers
  - What ?
    - Layered are the subsystems or the existing architecture. Closed if only one layer communicates the one below, open if it can communicate any layer bypassing the others.
  - Why ?
    - Reusability of Layers, Standardization, Low Coupling, Improved Testability
  - Example
    - Any spring boot mvc. View Controller Service DAO Repository layers.

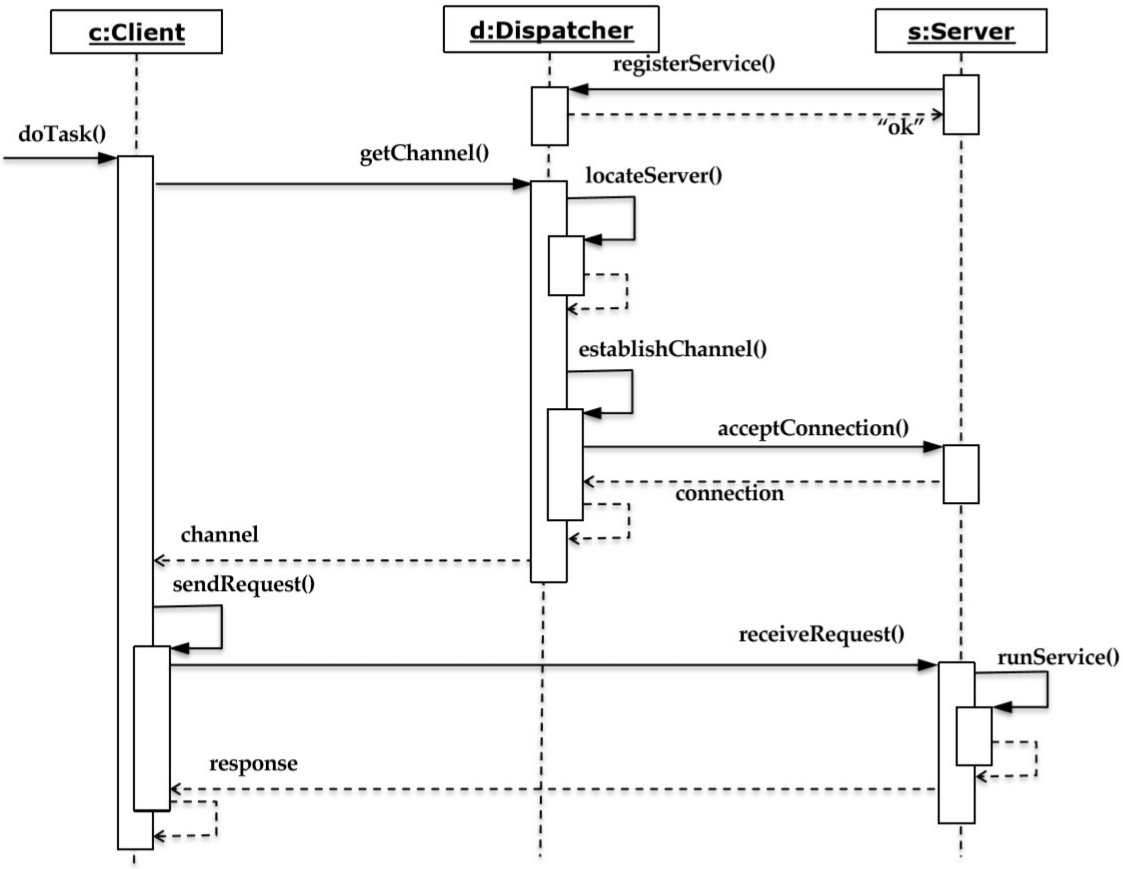
  - Steps :
    - 1. Identify subsystems with hierarchical structure
    - 2. Structure the individual layers
    - 3. Specify the communication protocol of layers
    - 4. Decouple layers
    - 5. Design error handling

- Blackboard ( to be done )
  - What ?
  - When ?
  - Example
  - Steps
    - 1. Define the Problem, no algorithmic solution, specify domain and actors
    - 2. Define solution space, possible solution candidates
    - 3. Identify knowledge sources, input / output
    - 4. Define blackboard, identify representation
    - 5. Define control, identify problem solving strategy
    - 6. Implement Knowledge

## Distributed Systems

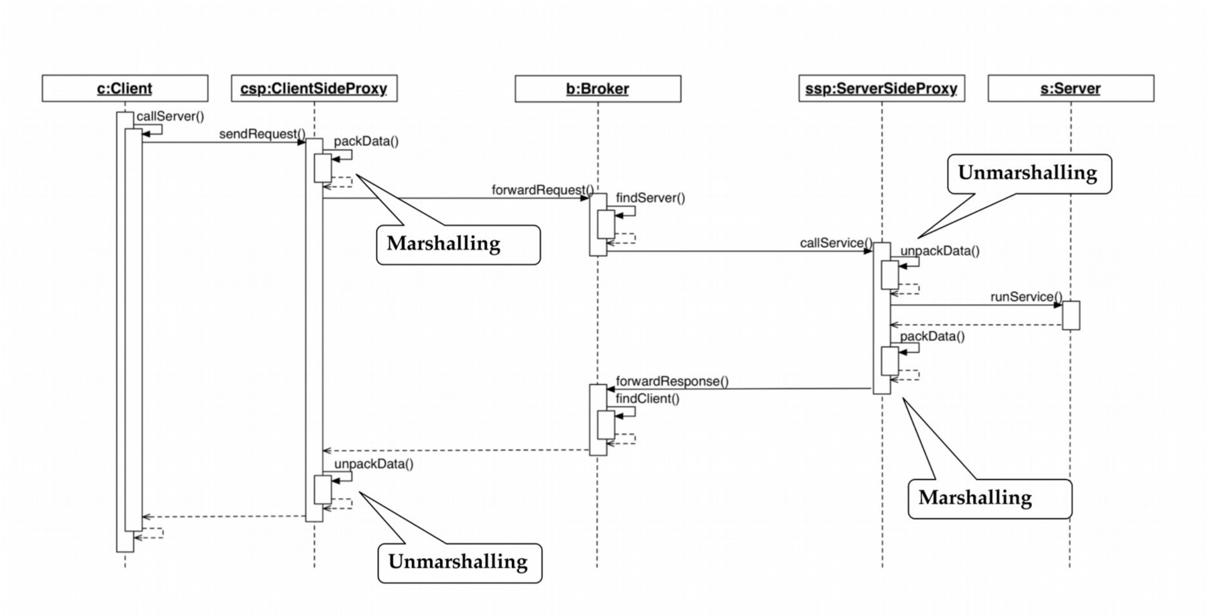
- Repository
- Client-server
  - Each Client calls a service provided by the Server; the Server performs the service and returns the result to the Client The Client knows the interface of the Server The Server does not know the interface of the Client
  - Components:
    - Client, a component that invokes services of a server component
    - Server: a component that provides services to clients. Servers have ports that describe the services they provide
  - Connector: a data connector employing a request/reply protocol, used by a client to invoke services on a server. Important characteristics include whether the calls are local or remote, and whether data is encrypted
  - Constraints:
    - Clients are connected to servers through request/reply connectors
    - Server components can be clients to other servers
  - Weaknesses:
    - The server can be a performance bottleneck
    - The server can be a single point of failure
    - Decisions about where to locate functionality (in the client or in the server) are often complex and costly to change after a system has been built.
- Client-dispatcher servlet
  - Context:

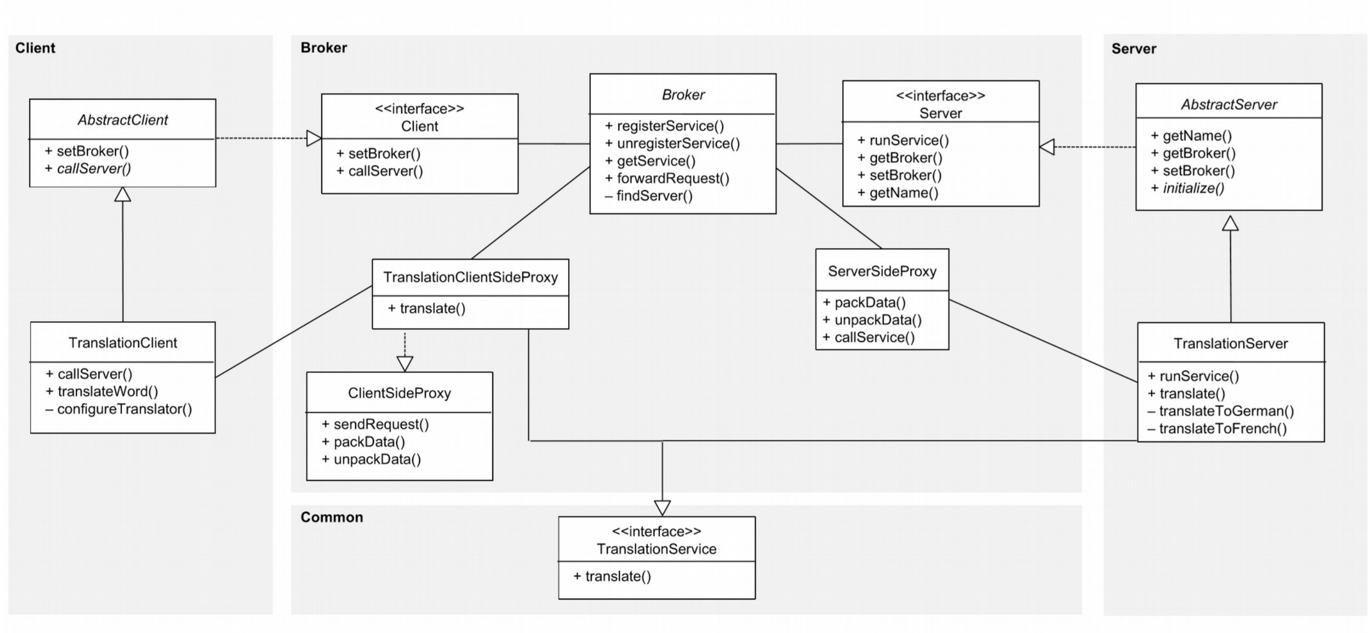
- When a client uses a remote server over a network it needs to establish a connection before the client can communicate with the server
- Problem: These two needs are not separated in many applications, causing unnecessary code complexity in service invocations.
- Separate the core functionality provided 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. Allows the client to refer to the server by name instead of the physical location (location transparency). Establishes a channel between client and server, reducing a possible communication bottleneck
- Weakness : No error handling, no support for communication between heterogeneous languages and platforms
- 1. Identify subsystems, 2. Decide on communication mechanism, 3. Specify protocol, 4. Configure dispatcher servlet, 5. Implement dispatcher, 6 implement client and server



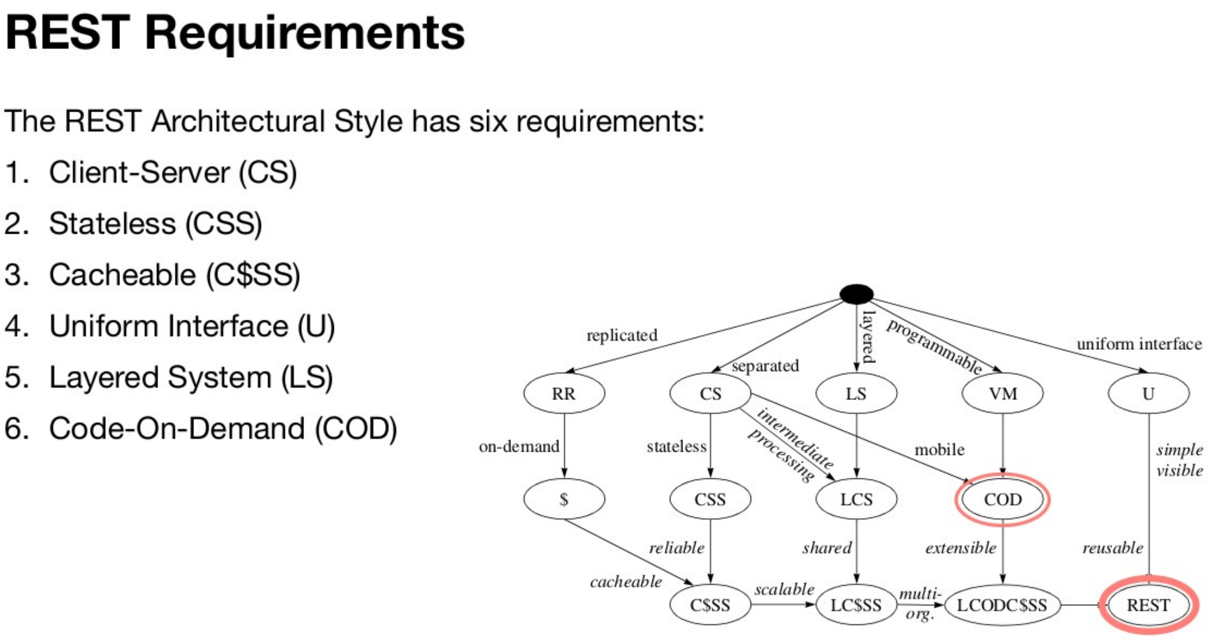
- Broker

- The broker coordinates the communication between heterogeneous nodes
- Forwarding requests, transmitting results, handling exceptions.
- Steps:
    - 1. Provide the Object Model and Service Definitions
        - Define client and server objects. The state of the server objects should be private. Clients may change or read the server’s state only by passing requests to the broker
        - Define the service interfaces with an existing interface definition language (IDL)
    - 2. Define the broker service Specify the services offered by the Broker (register, call, ...)
    - 3. Implement the Broker component and proxy objects at the client and server side
    - 4. Implement client and server





- REST

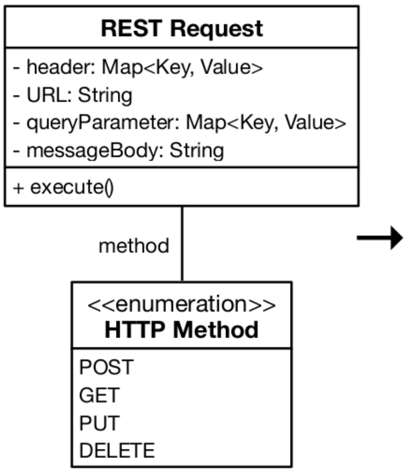


## REST Requirements

The REST Architectural Style has six requirements:

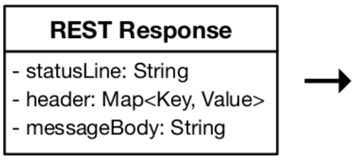
1. Client-Server (CS)
2. Stateless (CSS)
3. Cacheable (C$SS)
4. Uniform Interface (U)
5. Layered System (LS)
6. Code-On-Demand (COD)

# Implementation of REST Requests using HTTP



| Attribute | Description / Example |
|---|---|
| Header | Key-value information to configure the request<br>*Accept: */** |
| Method | *GET* |
| URL | *https://ase.in.tum.de/articles* |
| Query Parameter | Additional query parameter in the URL<br>*https://ase.in.tum.de/articles**?name=test*** |
| Message Body | Message to the server<br>"" |

Serialized Object



| Attribute | Description / Example |
|---|---|
| Status Line | Information about the result of the request, e.g.<br>*HTTP/1.1 200 OK* |
| Header | Key-value information about the response, e.g.<br>*Content-Type: application/json; charset=utf-8*<br>*Date: Mon, 18 Dec 2017 10:53:26 GMT*<br>*Content-Encoding: gzip* |
| Message Body | Message to the client, e.g. in JSON<br>*{ "author_id": 1,*<br>*"text": "I love this lecture – PSE rocks!"}* |

HTTP Status Code

## REST Best Practices - HTTP Body Data

- **GET**
  - → No data in the HTTP body Request
  - ← Resource or a collection of Resources serialized in the HTTP body Response
- **POST**
  - → Resource without an ID serialized in the HTTP body
  - ← Resource with an ID serialized in the HTTP body
- **PUT**
  - → New Resource in the HTTP body, ID encoded in the HTTP URI path
  - ← Updated Resource in the HTTP body including the same ID as before
- **DELETE**
  - → No data in the HTTP body, ID encoded in the HTTP URI path
  - ← Empty body with HTTP Status Code as an success indicator

## REST Bad Practices

Also called smells (Lecture on Antipatterns II)

1. **Breaking self-descriptiveness**
   - Own headers or formats break may break the self-descriptiveness of your API
2. **Ignoring caching capabilities**
   - Take advantage of caching specifications to reduce network load
3. **Ignoring response codes**
   - Use response codes in your application to increase reusability, interoperability of subsystems, and to enable looser coupling
4. **Handling authentication via parameters in the URL**
   - URLs and query parameters aren’t secure, use a token based authentication
5. **Tunneling everything through one HTTP Method or URI path**
   - Use GET, POST, PUT and DELETE and multiple Endpoints

Cons of REST

- Decisions where to locate functionality (in the Client or in the Web Service) are often complex and costly to change after a system has been built

- Larger, nested web resource exchanges often require multiple independent requests

- The stateless nature prohibits e.g. publisher subscriber use cases

## -GRAP QL-

GraphQL as the refactored solution

- Dynamic GraphQL queries offer more flexible APIs

- GraphQL queries can traverse related objects and their fields

- GraphQL subscriptions are a way to push data from the server

to the clients that listen to real time messages

- Also uses HTTP messages and encodes queries and responses

in the HTTP body

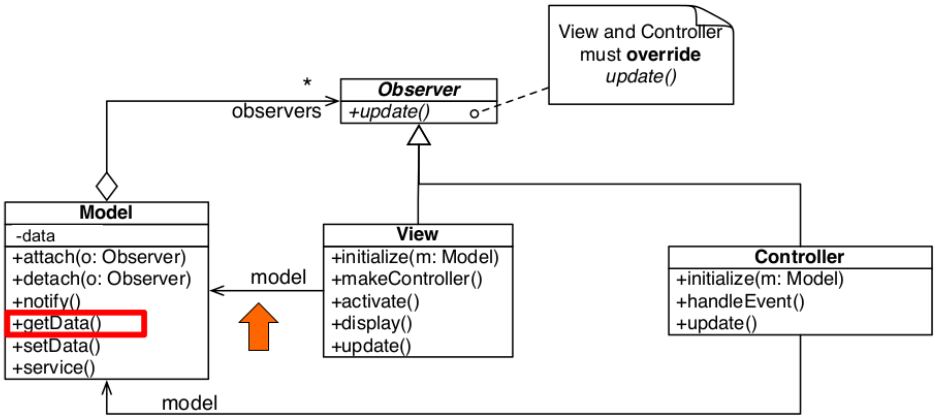
# Interactive systems

- MVC

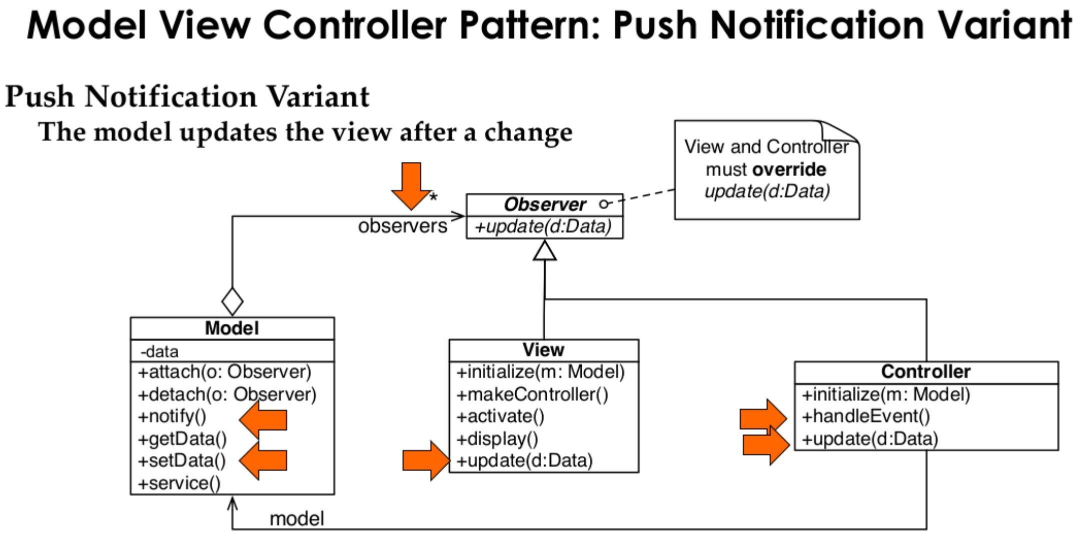
• Problem: In systems with high coupling changes to the user interface (boundary objects) often force changes to the entity objects (data)
• The user interface cannot be re-implemented without changing the representation of the
entity objects
• The entity objects cannot be reorganized without changing the user interface
• Solution: Decoupling! The model-view-controller architectural style decouples
data access (entity objects) and data presentation (boundary objects)
• The Data Presentation subsystem is called the View
• The Data Access subsystem is called the Model
• The Controller subsystem (control objects) mediates between View (data presentation) and Model (data access)
• Often called MVC.

When model is changed observers notified without data, when they notified after that they ask for newest data.

## Model View Controller Pattern: Pull Notification Variant



When model is changed observers notified with changed data.



## Model View Controller Pattern: Push Notification Variant

**Push Notification Variant**

**The model updates the view after a change**

View and Controller must **override** *update(d:Data)*

***Observer***
*+update(d:Data)*

**Model**
-data
+attach(o: Observer)
+detach(o: Observer)
+notify()
+getData()
+setData()
+service()

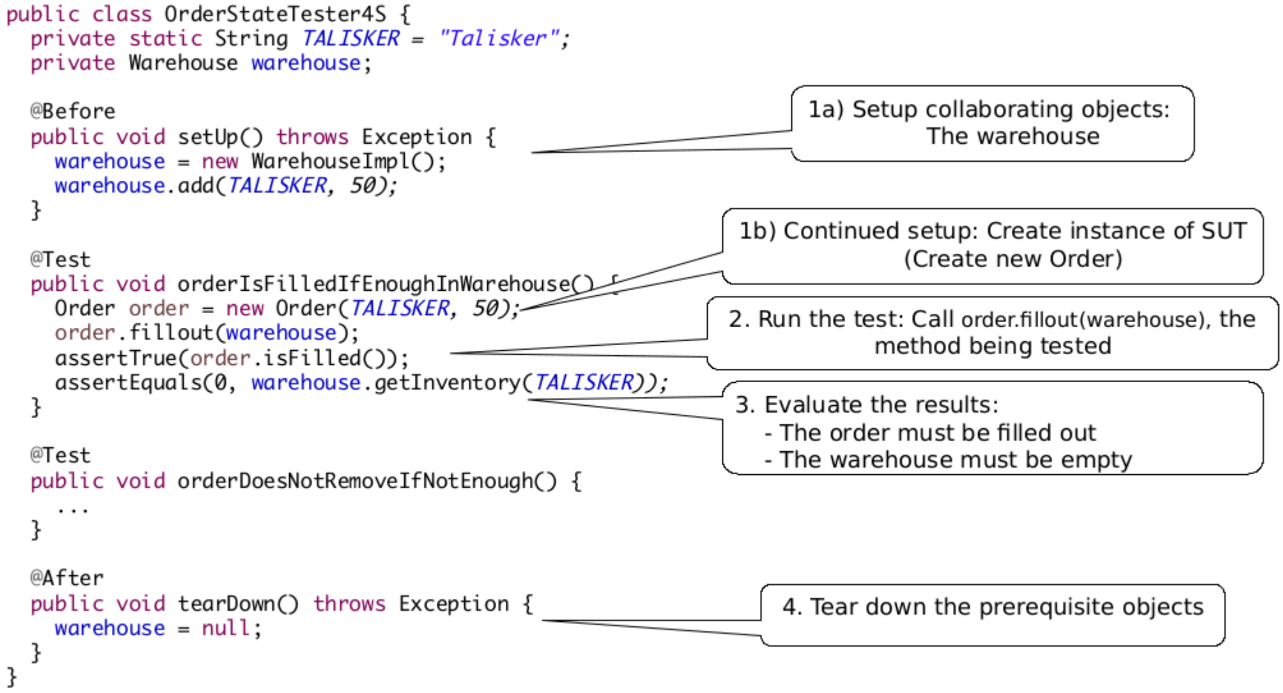
**View**
+initialize(m: Model)
+makeController()
+activate()
+display()
+update(d:Data)

**Controller**
+initialize(m: Model)
+handleEvent()
+update(d:Data)

# Testing Patterns

Slides very up to point.

# Example: A test suite for unit-testing the Order class using the Four-Stage Testing Pattern



```
public class OrderStateTester4S {
  private static String TALISKER = "Talisker";
  private Warehouse warehouse;

  @Before
  public void setUp() throws Exception {
    warehouse = new WarehouseImpl();
    warehouse.add(TALISKER, 50);
  }

  @Test
  public void orderIsFilledIfEnoughInWarehouse() {
    Order order = new Order(TALISKER, 50);
    order.fillout(warehouse);
    assertTrue(order.isFilled());
    assertEquals(0, warehouse.getInventory(TALISKER));
  }

  @Test
  public void orderDoesNotRemoveIfNotEnough() {
    ...
  }

  @After
  public void tearDown() throws Exception {
    warehouse = null;
  }
}
```

1a) Setup collaborating objects:
The warehouse

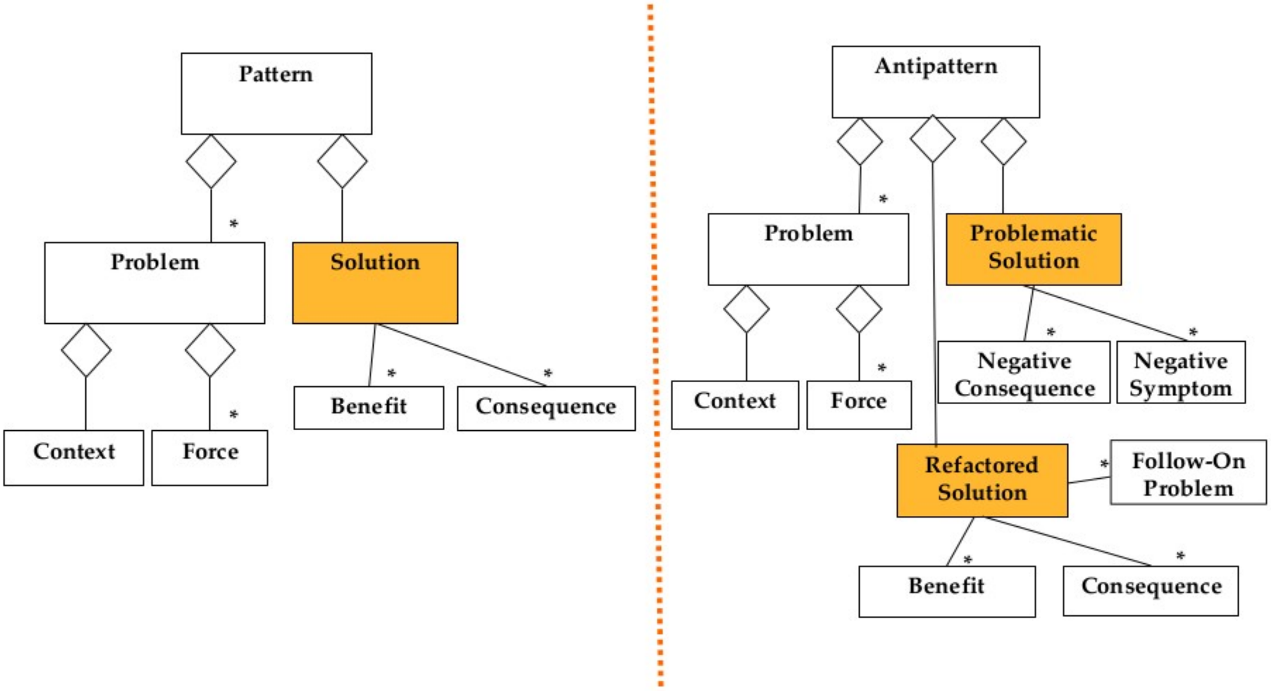
1b) Continued setup: Create instance of SUT
(Create new Order)

2. Run the test: Call order.fillout(warehouse), the method being tested

3. Evaluate the results:
- The order must be filled out
- The warehouse must be empty

4. Tear down the prerequisite objects

# Antipatterns



## 7 sins of software

Apathy : don't care about a problem, unwillingness find a solution

Haste : Being in hurry all the time

Narrow-Mindedness :The refusal to use solutions that are widely known (“Why reuse? I only have to solve one problem”)

Sloth : Making poor decisions based on “easy” answers

Avarice (Excessive Complexity) : No use of abstractions, excessive modeling of details

Ignorance : Failure to seek understanding

Pride  : Not invented here (NIH): Not willing to adopt anything from the outside.

# Developer Antipattern

- The Blob
  - General form : majority resp. integrated in one big controller
  - symptoms and consequences : unrelated attributes methods integrated in one class
  - resons: lack of oo skill, architectural skill, lack of architectural enforcement
  - refactored solution : categorize methods,attributes move them into seperate classes

  -

- Lava flow
  - General form : mass of code which is not important or not used at all. Once its started hard to stop like lava flow
  - symptoms and consequences : unused comment out, no documented, complex code
  - reasons : r&d code into prod, high turnover, fear of breaking something bc noone knows how to fix it, unclear project goals
  - refactored solution : Avoid arch. Changes, support arch with clear requirement analysis
- Spaghetti code
  - general form : software has little structure
  - symptoms and consequences : no oo (inheritance, abstraction, polymorp..), source is hard to reuse, reenginering takes more than refactor
  - cause : no design, inexperience in oo
  - refactored solution : incremental code refactor cleanup, and this should part of existing development cycle.
- Golden hammer ( also arch, manager antipattern )
  - General form : developer is highly skilled in a spesific field
  - Symptoms and consequences : Particular tool used in every aspect of applications
  - reasons : large invesment to some tool, limited ability of purchased product set
  - refactored solution : adapting new tech, dev team is motivated up to date tech, management adopts new tech and hires new people with diff. Backround

- Functional decomposition ( also arch. And management antipattern)
  - General form : Everything is a function, everystep stands in a standalone descriptor
  - symptoms and consequences : one change requires understanding whole system because its so complex there is no differentiation
  - reasons : developers familiar with imperative languages ( languages you construct everything step by step)
  - unbalanced sources : complexity
  - refactored solution name : object oriented reengineering

# Architectural antipattern

- Vendor locking

General form : a software adapts a tech product and becomes dependent to vendors implementation

symptoms and consequences : software requires in depth vendor tech knowledge, promised features late/never delivered.

Causes : decisions made just for marketing reasons not detailed technically evaluated.

Refactored solution : abstraction of existing layers(Adapter pattern), separation app and software related domain knowledge

# Management Antipatterns

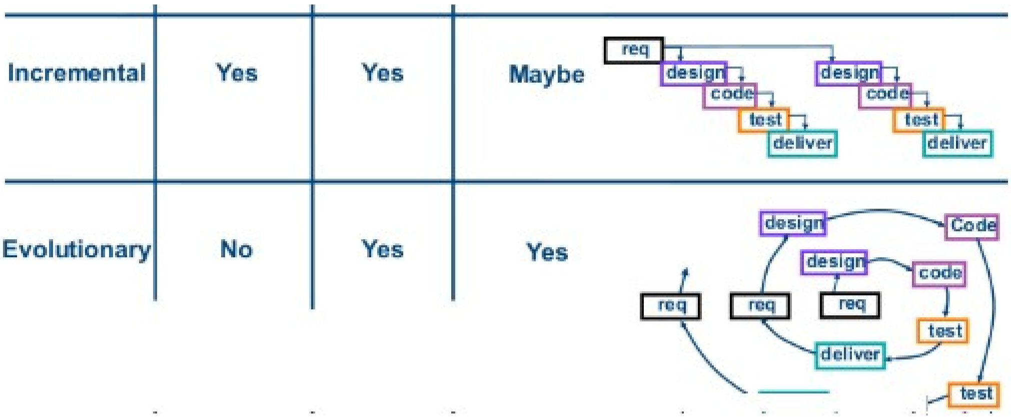
- ## Analysis paralysis

General form : Goal of achieving perfection of analysis phase, very detailed modelling

Cause : management decides on waterfall of phases, not clear analysis

symptoms and consequences : cost of analysis exceed expectations, analysis documents lost sync with domain experts

refactored solution : basically agile, adaptive incremental development.



| | | | | |
|---|---|---|---|---|
| Incremental | Yes | Yes | Maybe | |
| Evolutionary | No | Yes | Yes | |

req
design
code
test
deliver
design
code
test
deliver
design
Code
design
code
req
req
req
test
deliver
test

**Code Smell** : symptoms in a program that possibly indicates a bigger problem. Could be code level or arch. Level.

Examples of Business Smells

• Quality Delivered to Customer Is Unacceptable

Automated Developer Test, Functional Test

• Features Are Not Used by Customer

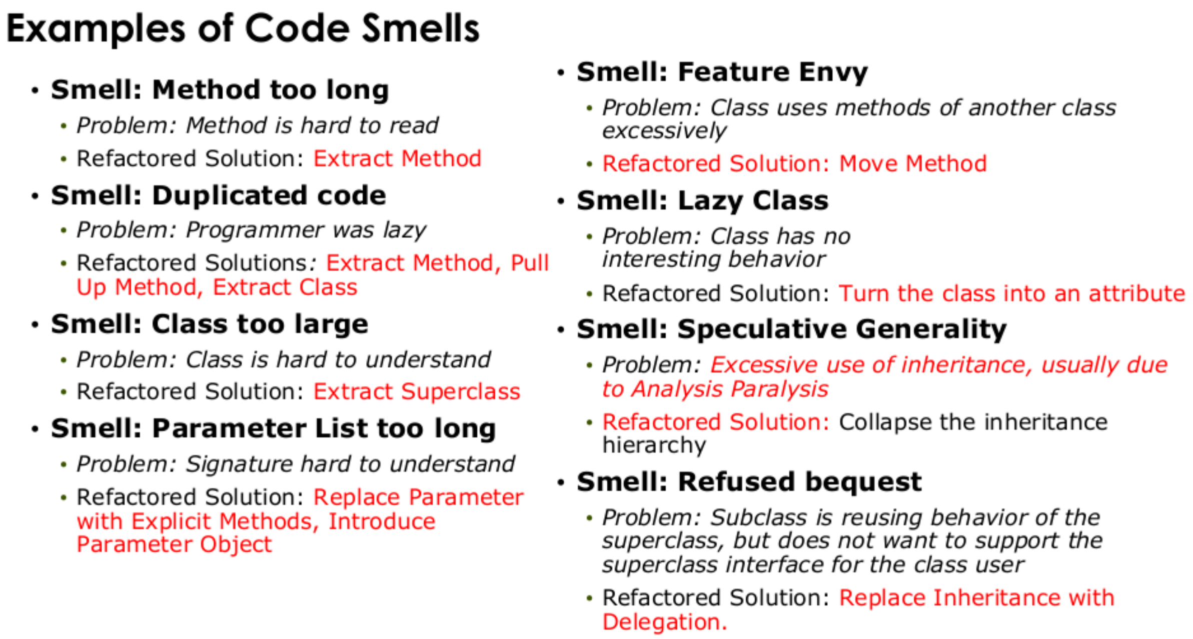
Prioritized Backlog

• Us Versus Them

Customer Part of Team, Release Often, Demo

• Customer Asks for Everything Including the Kitchen Sink

Kickoff-Meeting, Backlog, Planning Poker.



## Examples of Code Smells

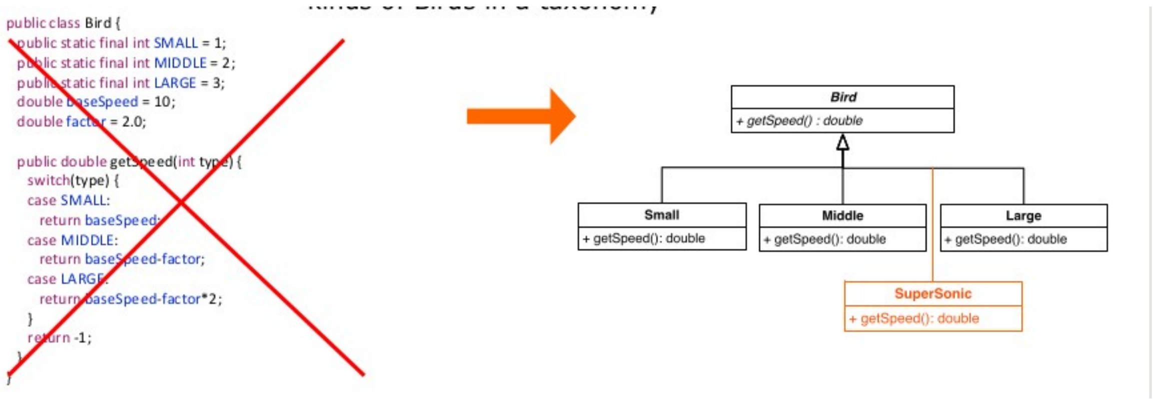
- **Smell: Method too long**
  - *Problem: Method is hard to read*
  - Refactored Solution: Extract Method
- **Smell: Duplicated code**
  - *Problem: Programmer was lazy*
  - Refactored Solutions*:* Extract Method, Pull Up Method, Extract Class
- **Smell: Class too large**
  - *Problem: Class is hard to understand*
  - Refactored Solution: Extract Superclass
- **Smell: Parameter List too long**
  - *Problem: Signature hard to understand*
  - Refactored Solution: Replace Parameter with Explicit Methods, Introduce Parameter Object
- **Smell: Feature Envy**
  - *Problem: Class uses methods of another class excessively*
  - Refactored Solution: Move Method
- **Smell: Lazy Class**
  - *Problem: Class has no interesting behavior*
  - Refactored Solution: Turn the class into an attribute
- **Smell: Speculative Generality**
  - *Problem: Excessive use of inheritance, usually due to Analysis Paralysis*
  - Refactored Solution: Collapse the inheritance hierarchy
- **Smell: Refused bequest**
  - *Problem: Subclass is reusing behavior of the superclass, but does not want to support the superclass interface for the class user*
  - Refactored Solution: Replace Inheritance with Delegation.

## Refactoring :

- Delegation instead inheritance(basically favor composition over inheritance)
- Extract method : extract functionality to method instead code repetation.
- Extract class : To preserve high cohesion, remove unrelated attributes to another class.

- Replace data value ot object : Assume Order keeps customerName, next time you need customerPhone and customerAddress and so on.. Instead composite the customer object inside Order then you are okay.

- Replace conditions to polymorphism :



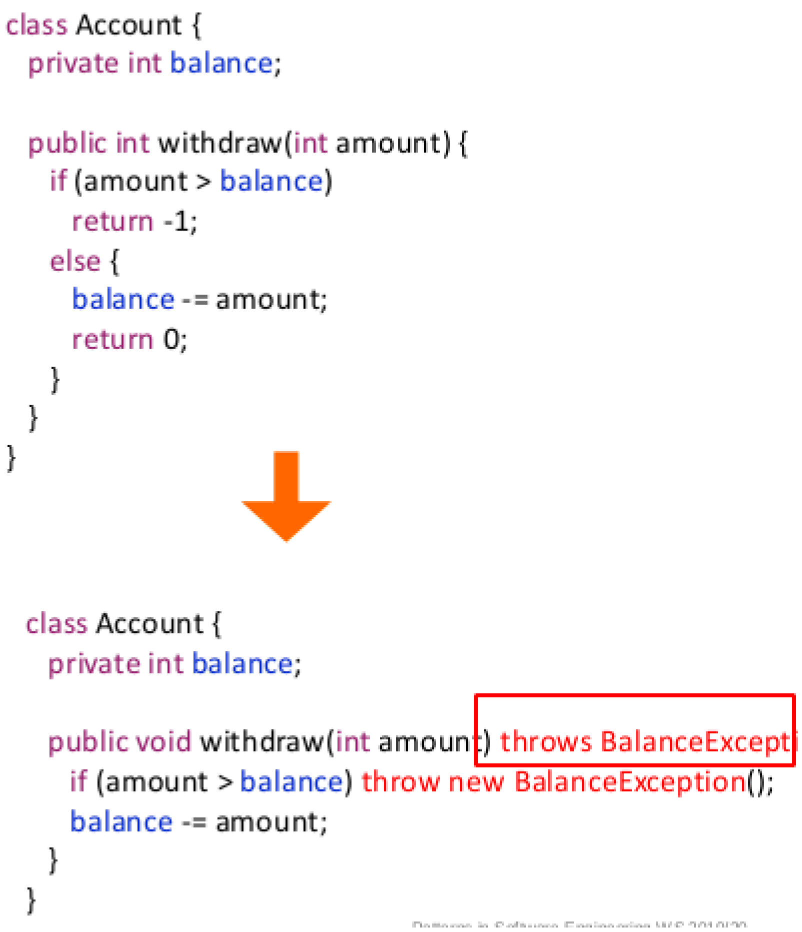
```
public class Bird {
  public static final int SMALL = 1;
  public static final int MIDDLE = 2;
  public static final int LARGE = 3;
  double baseSpeed = 10;
  double factor = 2.0;

  public double getSpeed(int type) {
    switch(type) {
    case SMALL:
      return baseSpeed;
    case MIDDLE:
      return baseSpeed-factor;
    case LARGE:
      return baseSpeed-factor*2;
    }
    return -1;
  }
}
```

Replace error code to exception : Handle checked exception callee method, handle unchecked exception in caller method. Favor unchecked exception handling, if caller checks exceptions its more robust.

checked(can be controlled compile time, try to open file compile forces you to check file exist)

unchecked(what if you open the file then someone deleted it at the same time, or there is no nth line you try to achieve, so compiler can’t see this case in compile time its runtime error)



```
class Account {
  private int balance;

  public int withdraw(int amount) {
    if (amount > balance)
      return -1;
    else {
      balance -= amount;
      return 0;
    }
  }
}
```

```
class Account {
  private int balance;

  public void withdraw(int amount) throws BalanceExcept
    if (amount > balance) throw new BalanceException();
    balance -= amount;
  }
}
```

# IOT

Internet of Things (IoT): A system of computing devices, mechanical and digital machines, objects, animals or people that are provided with unique identifiers (UIDs) and the ability to transfer data over a network without requiring human-to-human or human- to-computer interaction.

Cyber Physical systems are the same thing as IOT. IIOT (Industrial IOT)

Smart objects uses ,Ubiquitous Computing, : A paradigm where computing appears anytime and everywhere.

Edge computing : end devices computation happens inside there, might be connected to other central devices via network or not.

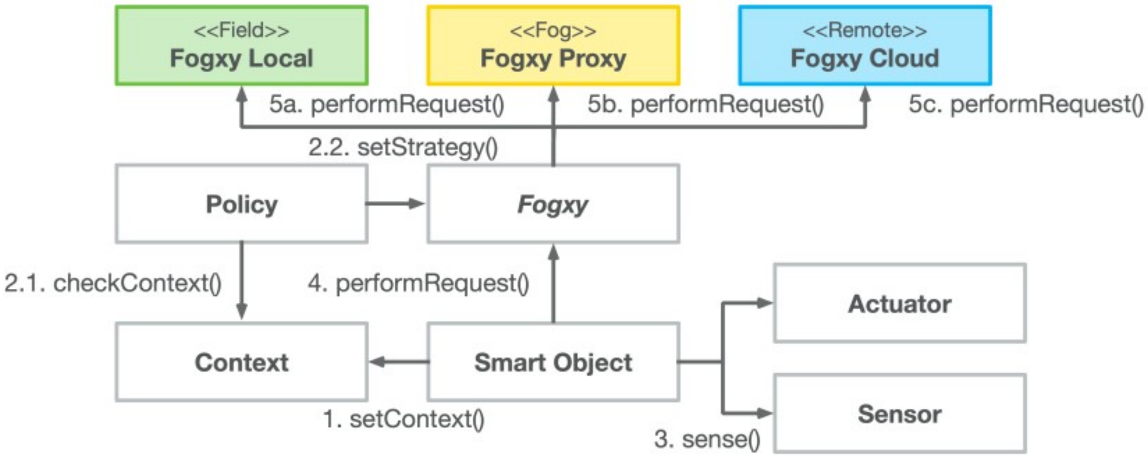
Fog computing : Distribution of compute services, near to the edge devices takes the information from them procceses and passes to its center.

Fogxy pattern is an example for a fog computing architecture.

# Fogxy uses Architectural Styles and Design Patterns

The Fogxy Architectural Style is composed of 2 architectural styles and 3 design patterns (*)

1. Layered Architectural Style with 3 layers: Fogxy Proxy, Fogxy Local and Fogxy Cloud
2. Peer-to-Peer Architectural Style: The Fogxy Proxy layer can act as a cache for the Fogxy Local layer as well as the Fogxy Cloud layer
3. Proxy Pattern: **Fogxy** is the abstract class, **Fogxy Proxy** the proxy, and the **Fogxy Cloud** the real subject. The proxy pattern enables availability regardless of cloud availability. The network load is reduced as requests only have to reach the Fogxy Proxy and not the cloud, thereby also reducing latency and enabling real-time applications.
4. Observer Pattern: The observer pattern facilitates communication between **Fogxy** and the **Smart Objects**, enabled by the two interfaces **Publisher** and **Subscriber**.
5. Strategy Pattern: The **Context** class describes the context of Smart Objects (battery power, connectivity, computational power). The **Policy** considers this information with respect to the nonfunctional requirements (real-time, availability, etc) to select a strategy on how to reach Fogxy Local, Fogxy Proxy or Fogxy Cloud.



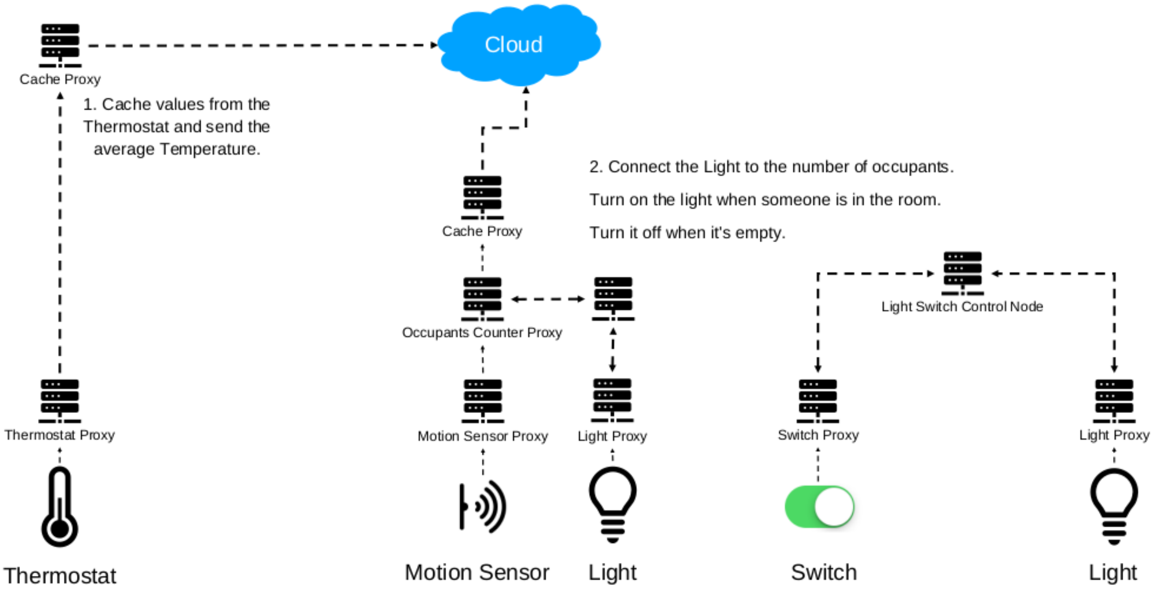
## Notes:

- Bridge,State,Strategy have very similar structures. Indeed, all of these patterns are based on composition, which is delegating work to other objects. However, they all solve different problems. A pattern isn't just a recipe for structuring your code in a specific way. It can also communicate to other developers the problem the pattern solves.

Caching fogxy: fog local may act like a cache before proceeding information directly to cloud.

Substitute fogxy : local smart component may not be applicable to compute a spesific task. Therefore fogxy proxy or remote cloud can provide functionality.

Access control fogxy : The fogxy component may act like authenti/authorization mechanism to decide which component allowed to pass its information to remote cloud.


Cloud
Cache Proxy
1. Cache values from the Thermostat and send the average Temperature.
Cache Proxy
2. Connect the Light to the number of occupants.
Turn on the light when someone is in the room.
Turn it off when it's empty.
Occupants Counter Proxy
Light Switch Control Node
Thermostat Proxy
Motion Sensor Proxy
Light Proxy
Switch Proxy
Light Proxy
Thermostat
Motion Sensor
Light
Switch
Light

Asdas

## -Memorizing Part-

Gang of four – writer of “Design Patterns. Elements of Reusable Object-Oriented Software”

Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides

**Algorithm :**

A method for solving a problem using a finite sequence of well-defined instructions for solving a problem

• Starting from an initial state, the algorithm proceeds through a series of successive states

• Terminating algorithm: Terminates in a final state

**Pattern:**

„A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice“

--- Christopher Alexander, A Pattern language.

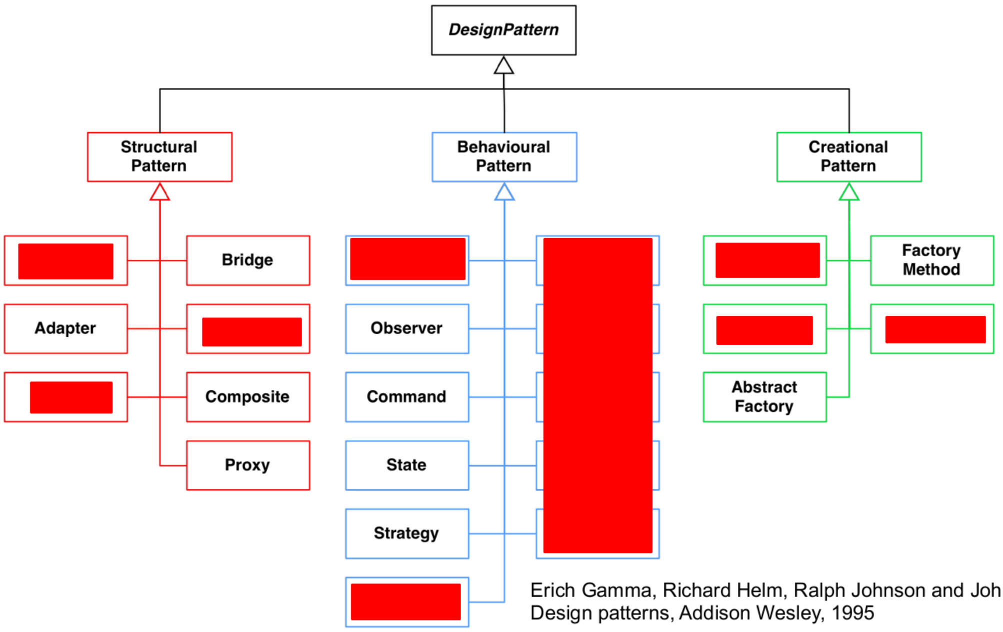
**Original definition (Christopher Alexander):**

A **pattern** is a three-part rule, which expresses a relation between a certain **context**, a **problem**, and a **solution**.

**His philosophy :**

- Users know more about what they need from buildings and towns than an architect
- Good buildings(softwares) are based on a set of design principles that can be described with a pattern language

Design Patterns of GOF (that we saw)

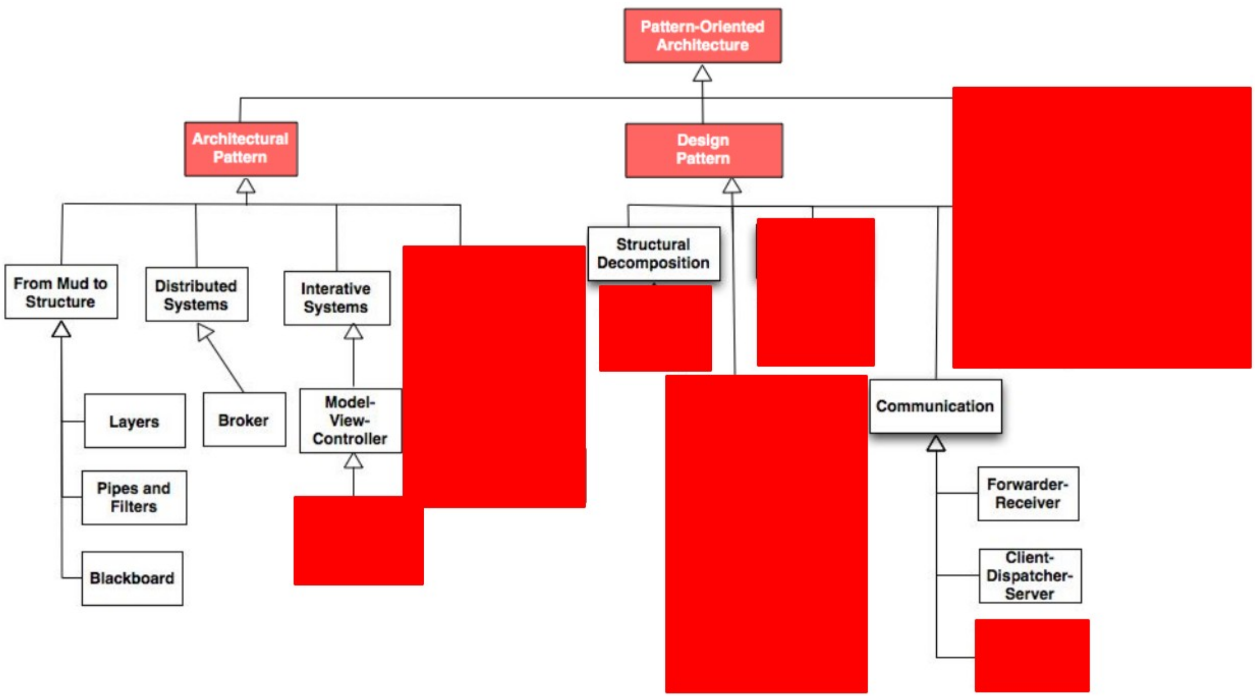


Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides
Design patterns, Addison Wesley, 1995

© 2019 Bernd Bruegge

Patterns in Software Engineering – WS 2019-2020

44



Pattern-Oriented Architecture

Architectural Pattern

Design Pattern

[redacted]

From Mud to Structure

Distributed Systems

Interative Systems

[redacted]

Structural Decomposition

[redacted]

[redacted]

Layers

Broker

Model-View-Controller

Pipes and Filters

Blackboard

[redacted]

[redacted]

Communication

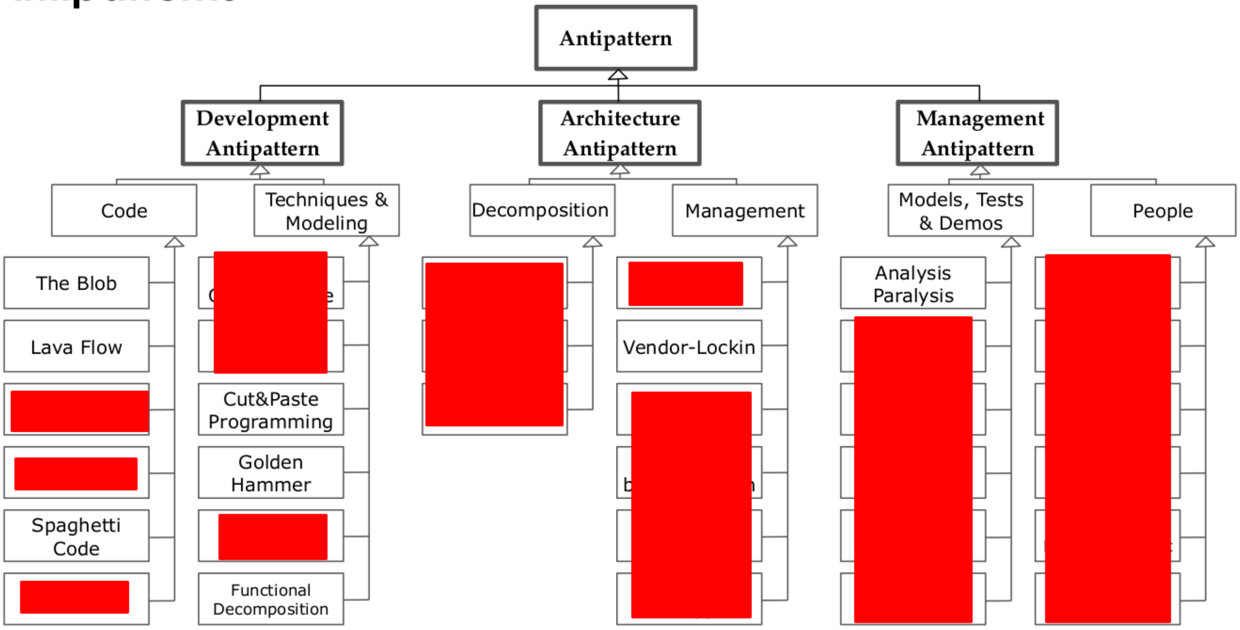
Forwarder-Receiver

Client-Dispatcher-Server

[redacted]

## Antipatterns

**William J. Brown Anti-Patterns and Patterns in Software Configuration Management, Wiley 1999**



Antipattern

Development Antipattern

Architecture Antipattern

Management Antipattern

Code

Techniques & Modeling

Decomposition

Management

Models, Tests & Demos

People

The Blob

Lava Flow

[redacted]

[redacted]

Spaghetti Code

[redacted]

[redacted]

Cut&Paste Programming

Golden Hammer

[redacted]

Functional Decomposition

[redacted]

[redacted]

Vendor-Lockin

[redacted]

Analysis Paralysis

[redacted]

[redacted]

**Alexander’s Schema (“Alexandrian Form”)**

**Name of the Pattern**

**Example**: Picture of an example for the pattern

**Context**: The range of situations in which the pattern can be used

**Problem**: Short description and elaborate description

**Solution**: Description and Diagram

**Conclusion**

Sections don‘t have explicit headings, sections are separated by symbols

3 diamonds between context and problem and after the solution

3 diamonds between solution and conclusion

Keyword “Therefore” to separate the problem from the solution.

**Gang of Four Schema**

- Pattern Name
- Intent
- Also Known As
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation

- Sample Code
- Known Uses
- Related Patterns

**Gang of Five Schema**

- Name
- Also Known As
- Example
- Context
- Problem
- Solution
- Structure
- Dynamics
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

**Creational Design Patterns**: Creational patterns deal with object creation i.e they look at ways to solve design issues arising out of creation of objects.

**Structural Design Patterns**: Structural patterns ease the design by identifying a simple way to realize relationships between entities.

**Behavioral Design Patterns**: Behavioral patterns identify common communication patterns between objects and realize these patterns.

Design Pattern

- Describes associations and collaborations of a reusable set of classes (used during Object Design)
- Architectural Style
- A pattern for a subsystem decomposition, i.e. describes relationships and collaborations of different subsystems (used during System Design)
- Software Architecture
- Instance of an