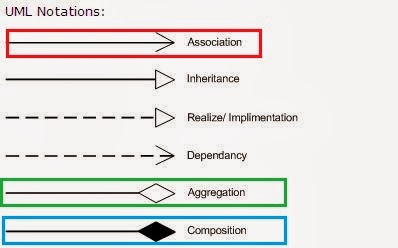
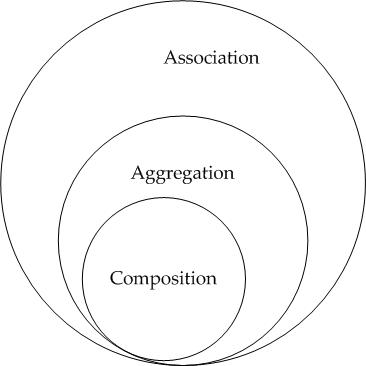
Question styles

* 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;

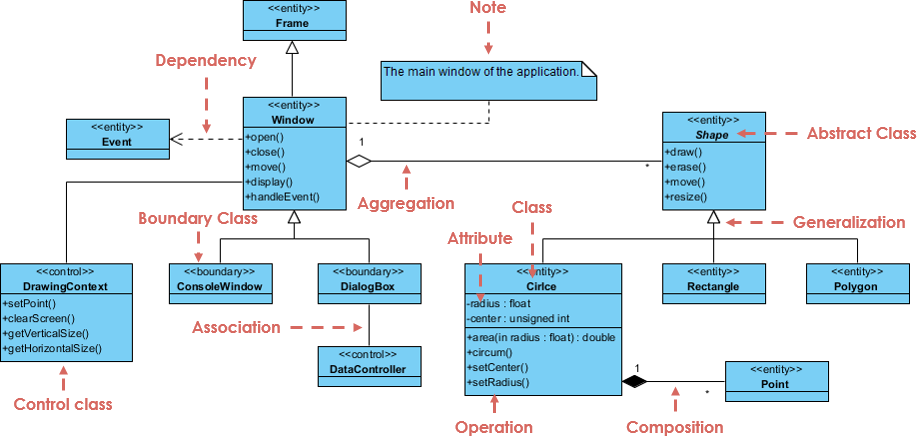
Foo(Bar bar) {

this.bar = bar;

}

}

|  |  |
| --- | --- |
| **Relationship Type** | Graphical Representation |
| **Inheritance (or Generalization):**   * **Represents an "is-a" relationship.** * **An abstract class name is shown in italics.** * **SubClass1 and SubClass2 are specializations of Super Class.** * **A solid line with a hollow arrowhead that point from the child to the parent class** | Inheritance |
| **Simple Association:**   * **A structural link between two peer classes.** * **There is an association between Class1 and Class2** * **A solid line connecting two classes** | Simple association |
| **Aggregation:**  **A special type of association. It represents a "part of" relationship.**   * **Class2 is part of Class1.** * **Many instances (denoted by the \*) of Class2 can be associated with Class1.** * **Objects of Class1 and Class2 have separate lifetimes.** * **A solid line with an unfilled diamond at the association end connected to the class of composite** | Aggregation |
| **Composition**:  A special type of aggregation where parts are destroyed when the whole is destroyed.   * Objects of Class2 live and die with Class1. * Class2 cannot stand by itself. * A solid line with a filled diamond at the association connected to the class of composite | Composition |
| **Dependency:**   * **Exists between two classes if the changes to the definition of one may cause changes to the other (but not the other way around).** * **Class1 depends on Class2** * **A dashed line with an open arrow** | Dependency |

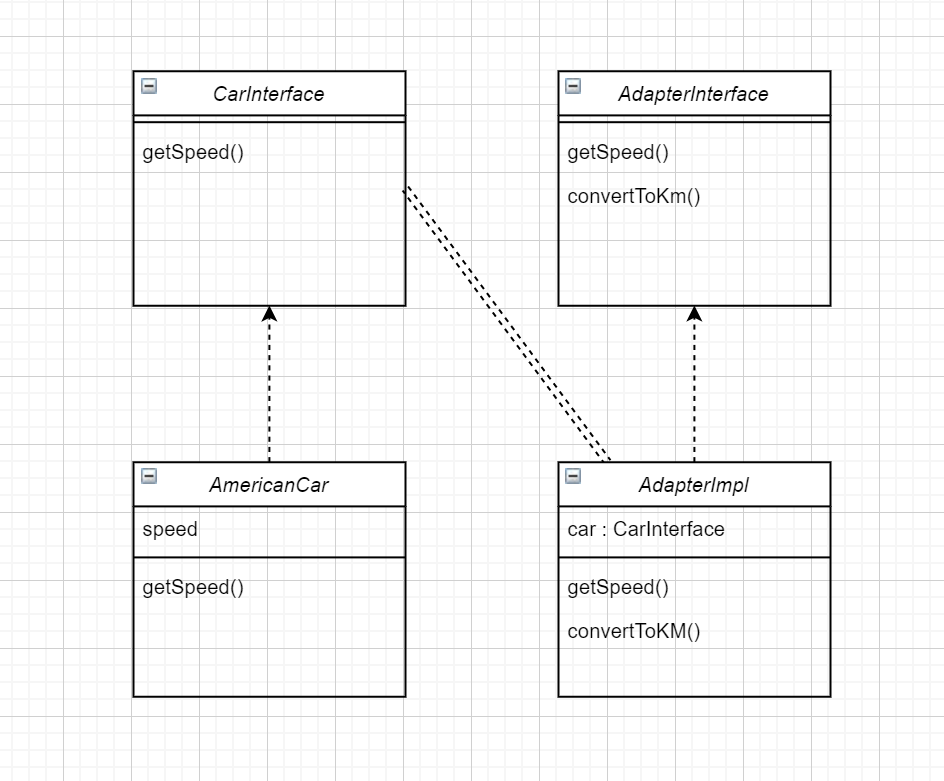


Foundations

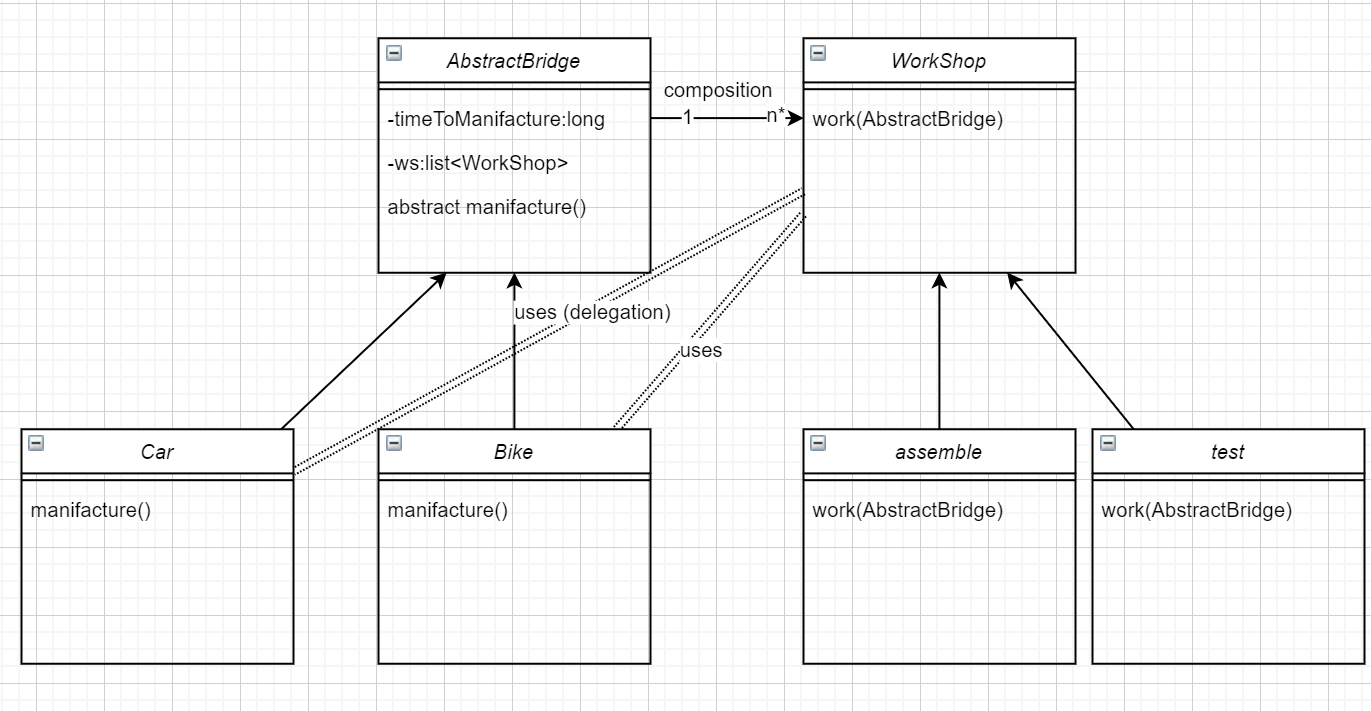
* Abstractions and Types
  + Abstraction can be of two types, namely, **data abstraction** and **control abstraction**. Data abstraction means hiding the details about the data and control abstraction means hiding the implementation details.
* Inheritance
  + In object-oriented programming, inheritance 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

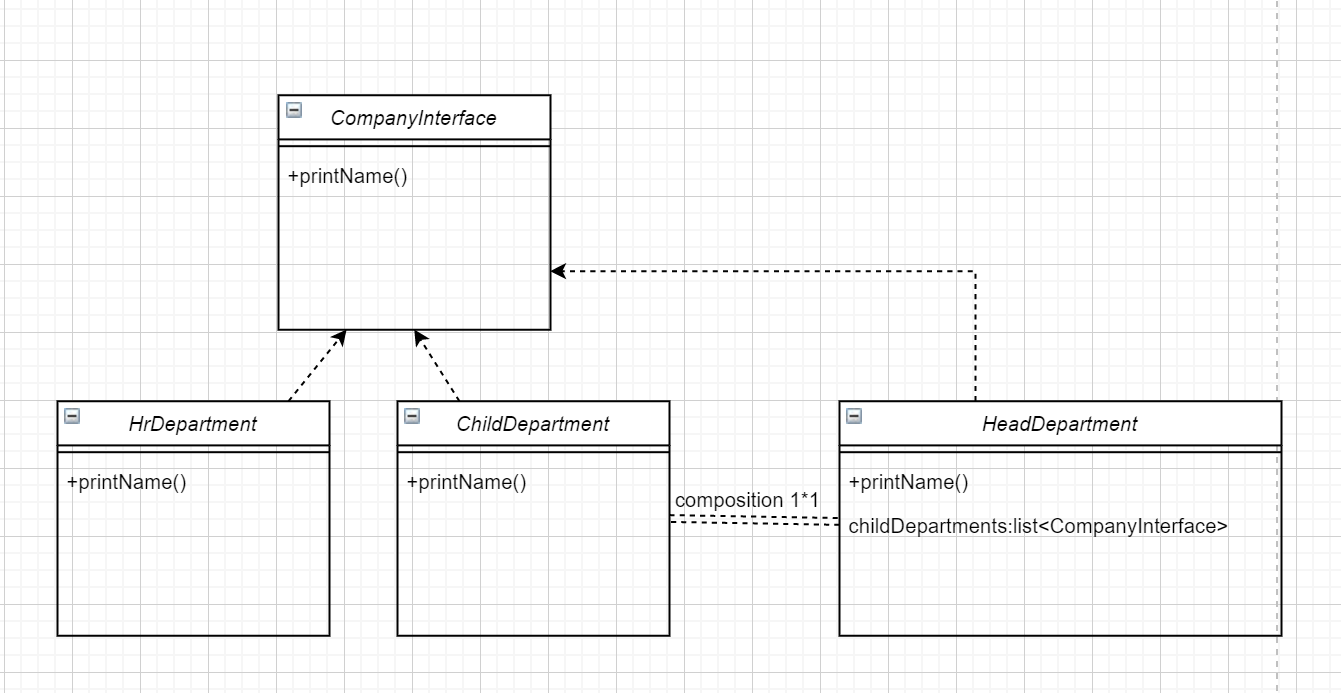
Structural Patterns

* 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: 

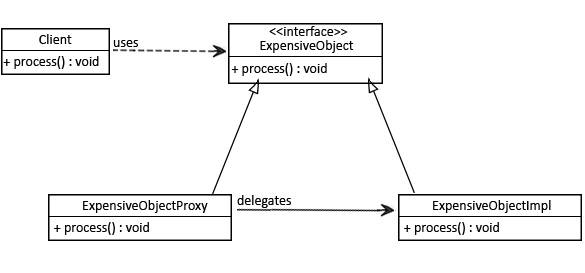
* Bridge
  + Why?
    - You split themonolithic class into several class hierarchies. After this, you abstract a bridge that makes delegation between those hierarchies.
  + How
    - Create a bridge abstract and store as composition other class inside then concrete classes of bridge uses stored composition 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

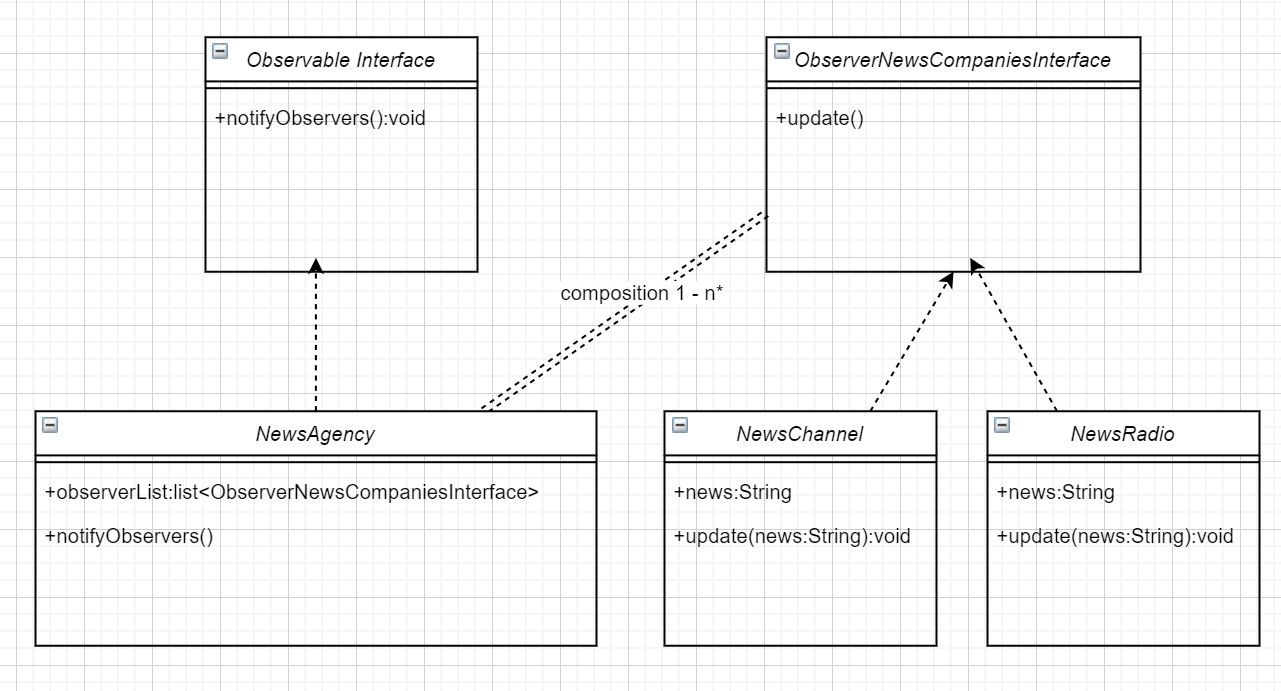


* Proxy
  + Why ?
    - When we want a simplified version of a complex or heavy object.
    - When we want to change a behavior of a class.
  + How ?
    - Have an interface and concrete actual and proxy objects. Replace it with a lightweight version and keep it that heavy weight concrete class inside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

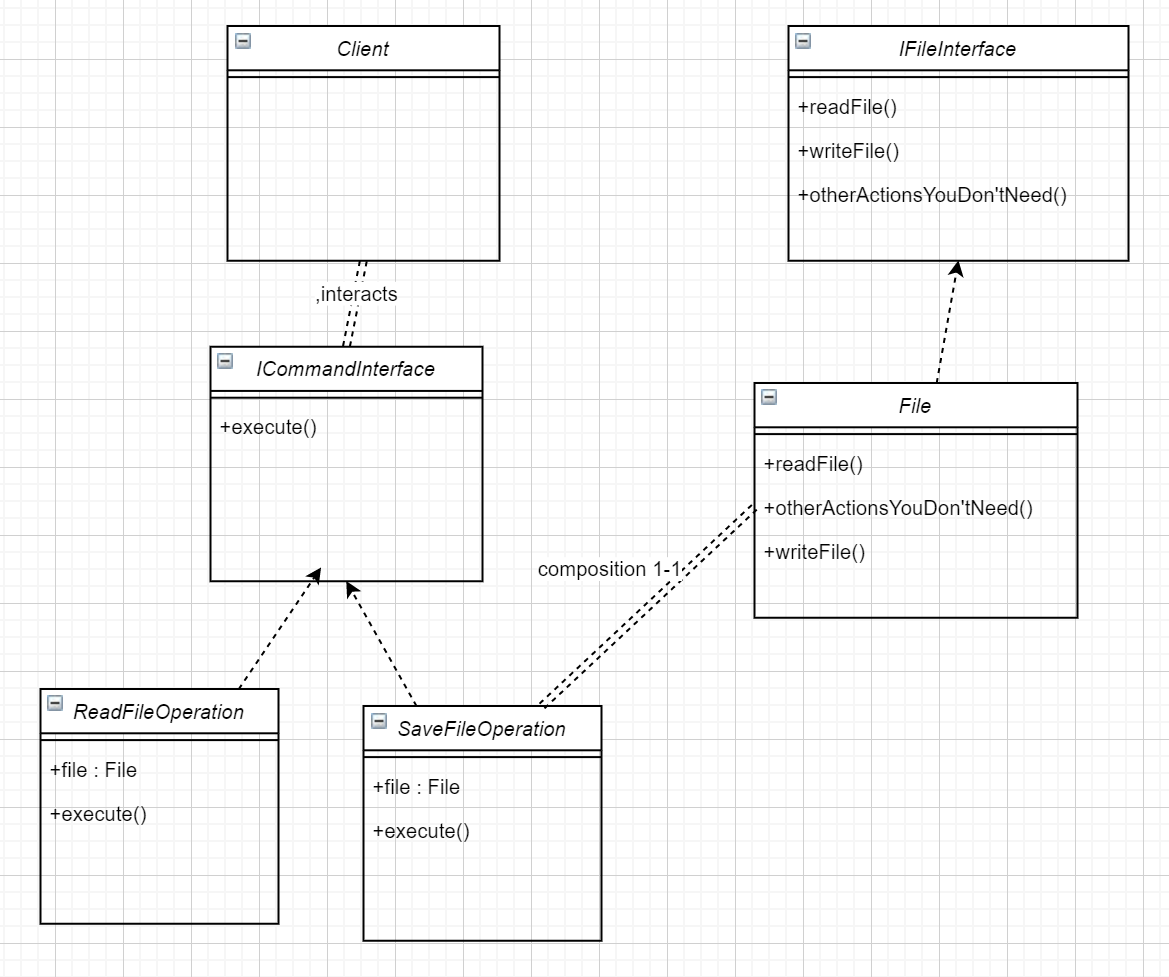


Behavioral Pattern

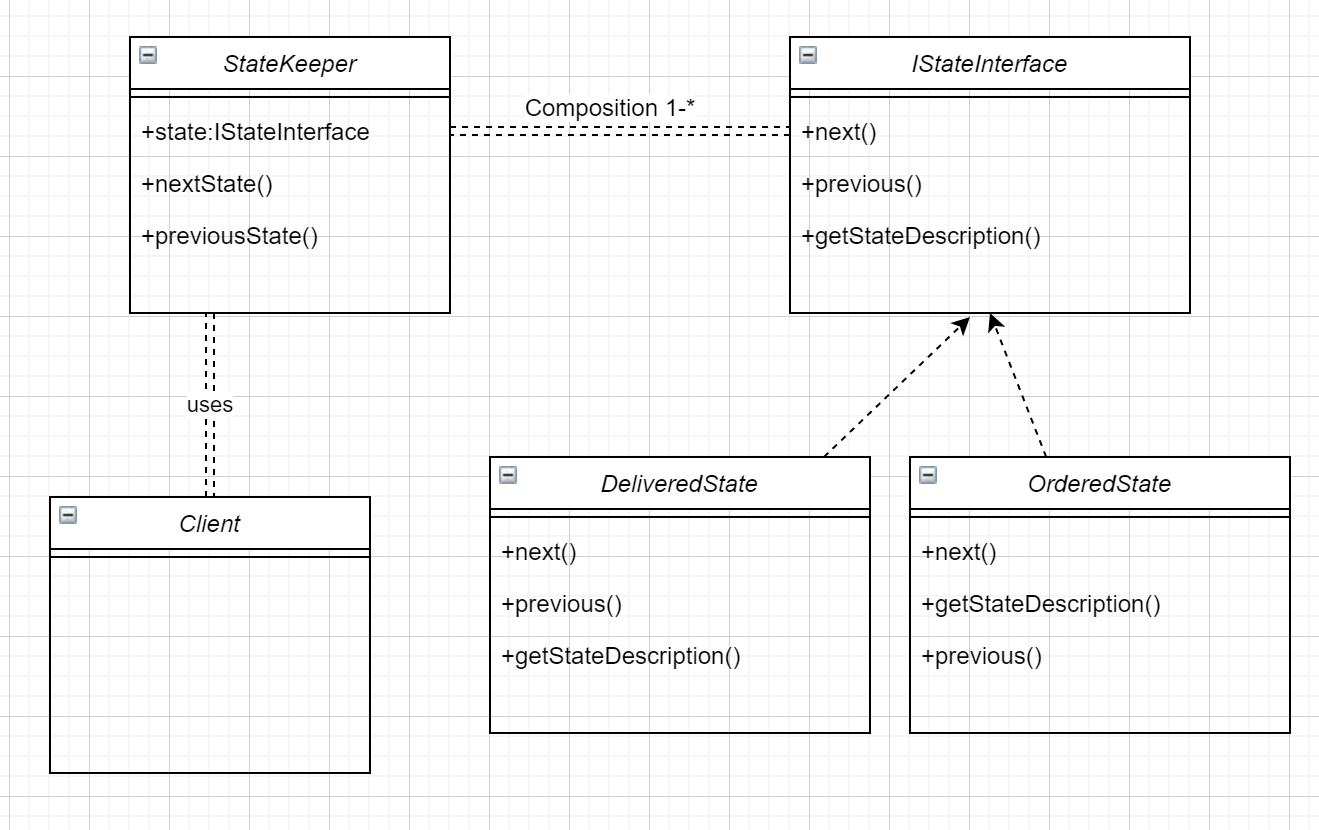
* Observer
  + Why ?
    - use Observer pattern when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically.
  + How ?
    - Create Observable interface put “notifyObservers” method, create a concrete class(the 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
    - News agency when it retrieves a news it passes the observers that it keeps inside.



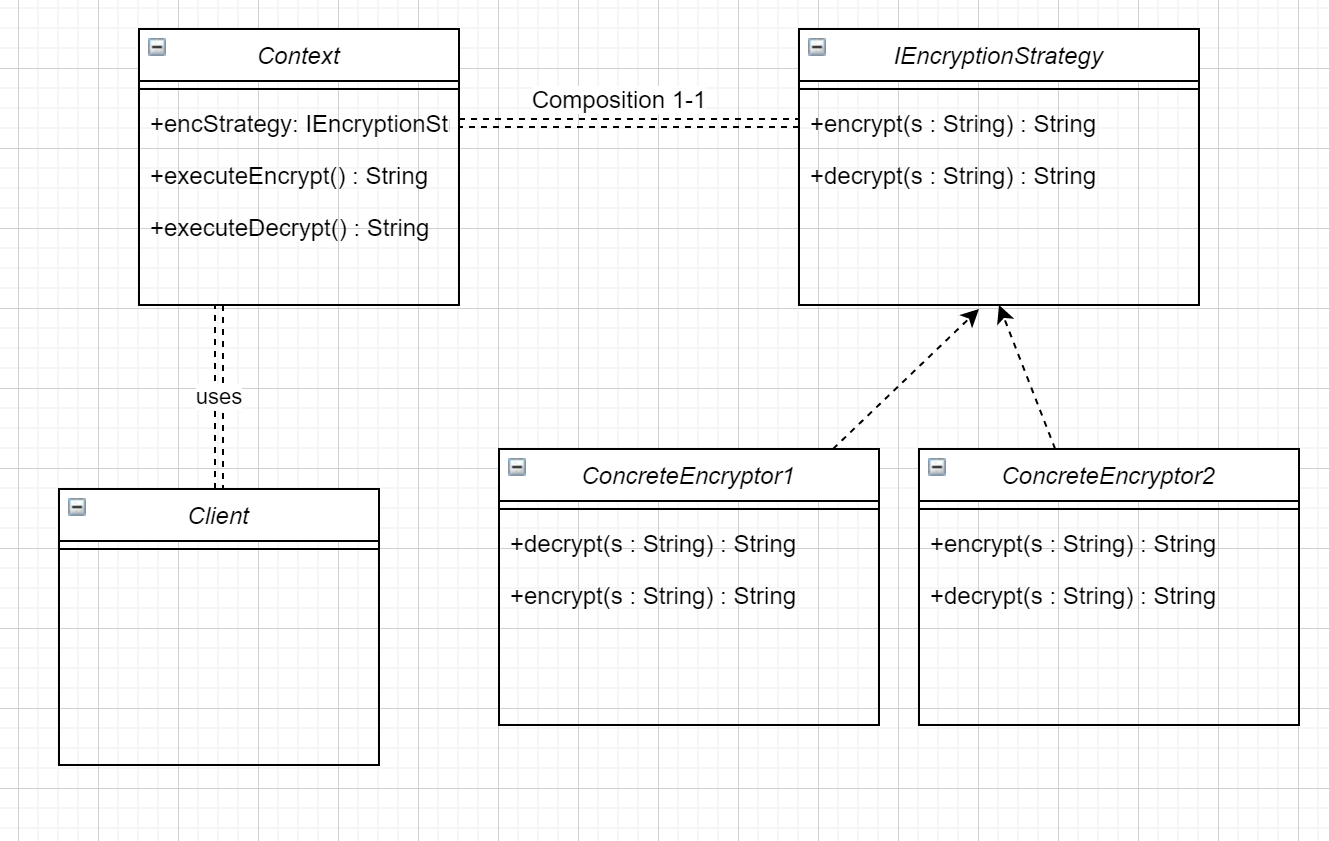
* Command
  + Why?
    - It decouples client and actual implementor. Delegates required actions the corresponding responsible. So it keeps an abstraction and encapsulation to command class.
  + How?
    - Create an interface which contains execute() method. Concrete classes of this interface are the 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.

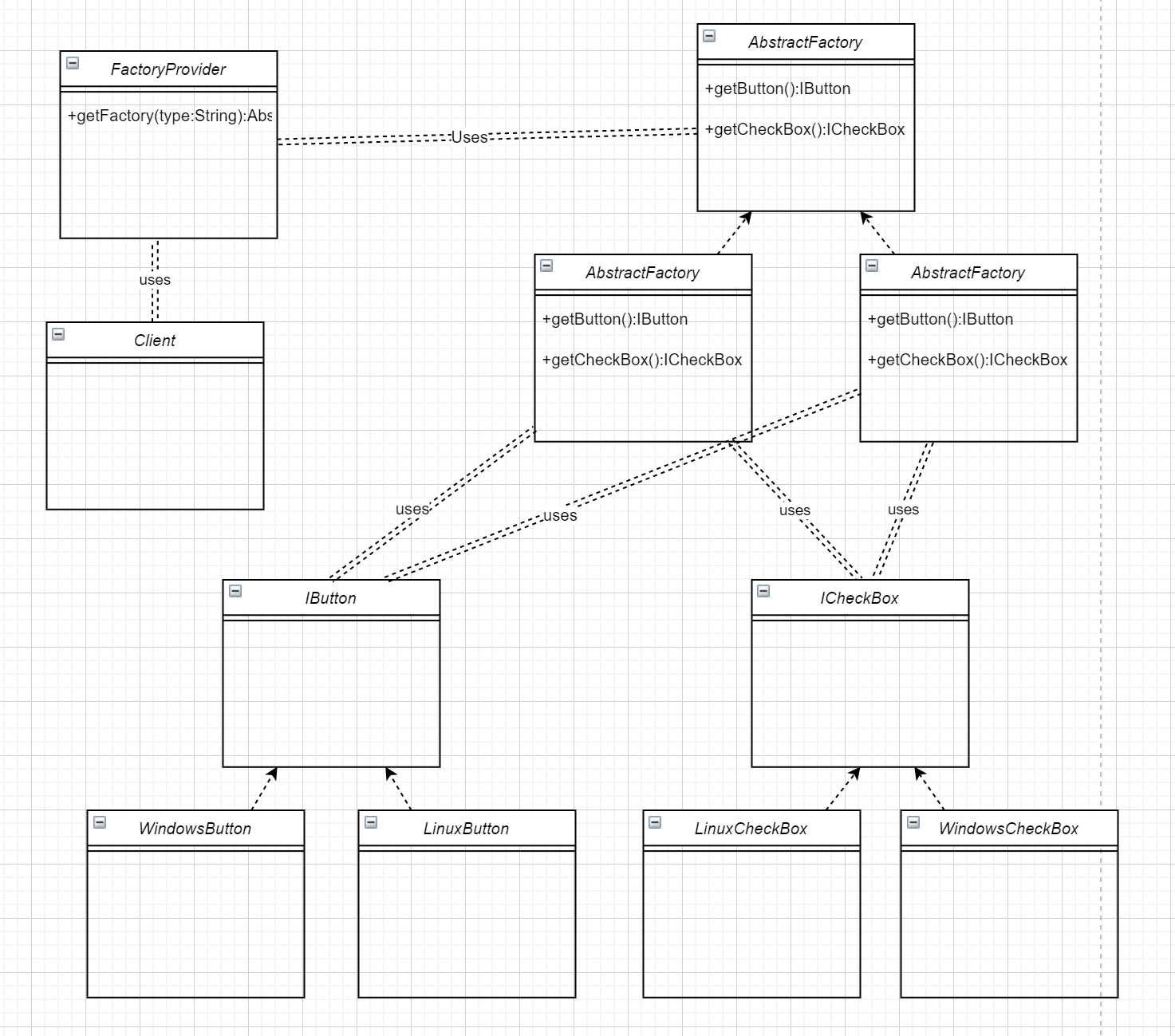


* Strategy
  + What
    - Essentially, the strategy pattern allows us to change the behavior of an algorithm at runtime.
  + When
    - Use the Strategy pattern when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.
  + 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.



Creational Pattern

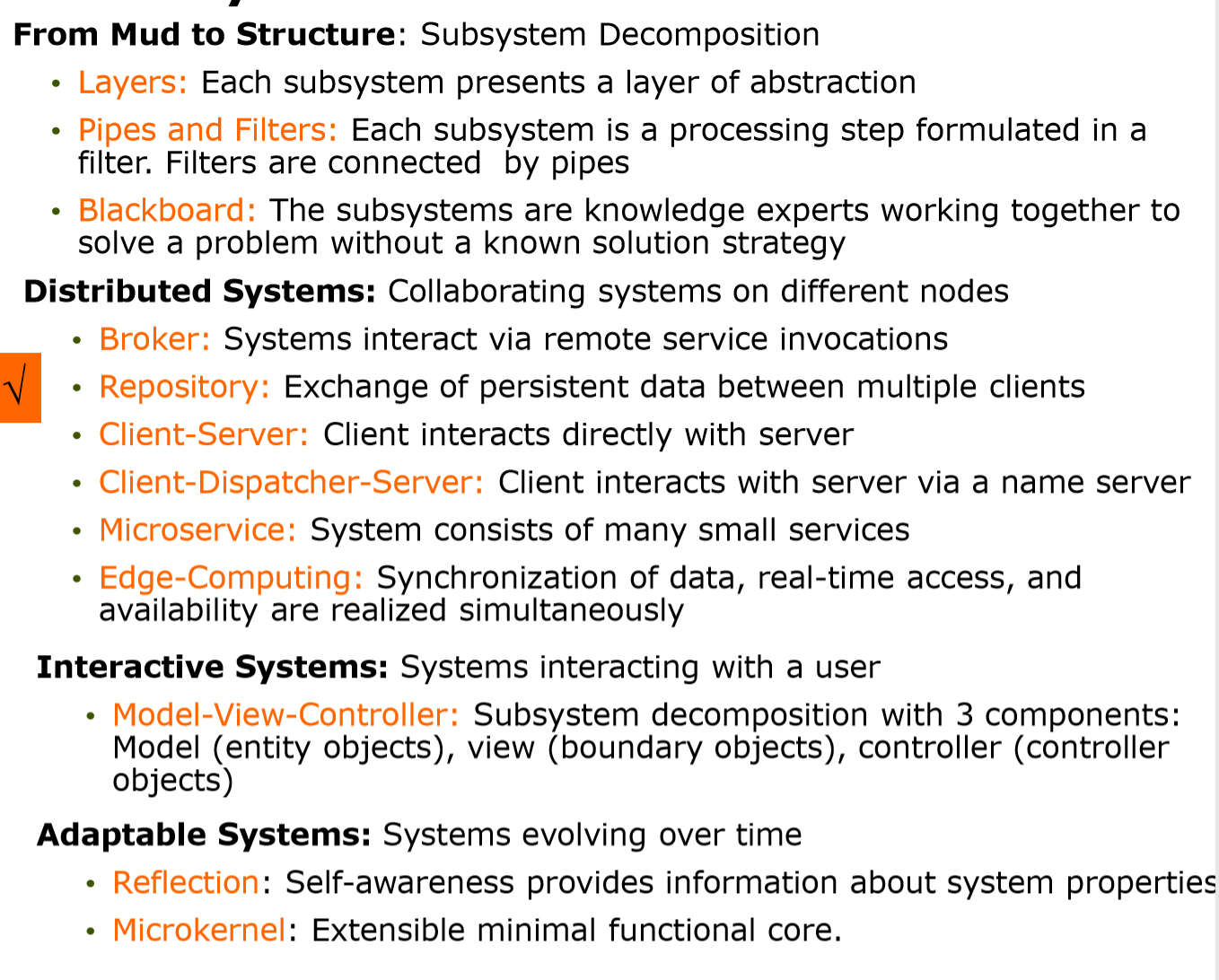
* Abstract factory
  + What?
    - Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.
  + When ?
    - Use the Abstract Factory when your code needs to work with various families of related products, but you don’t 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

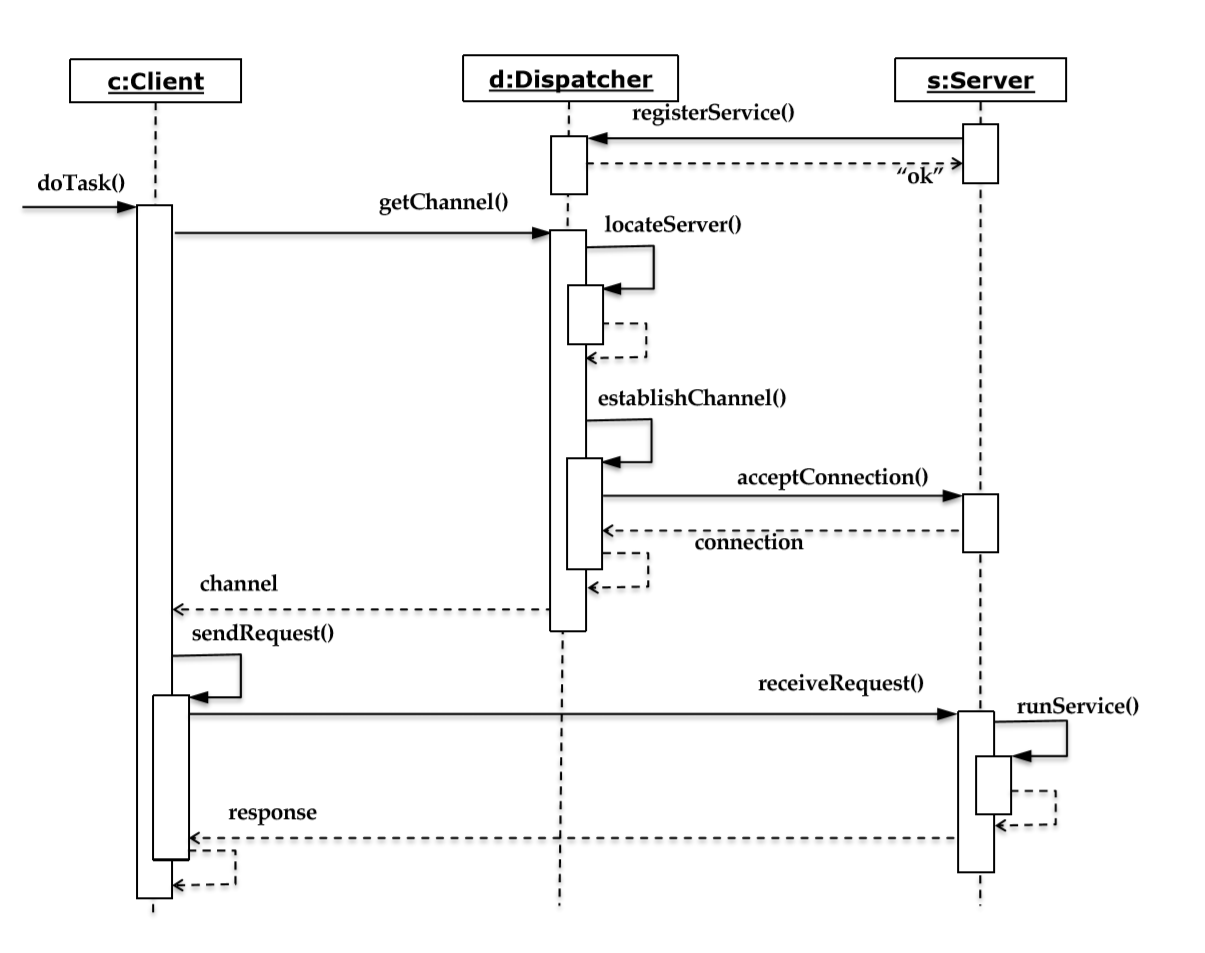


From mud to structure

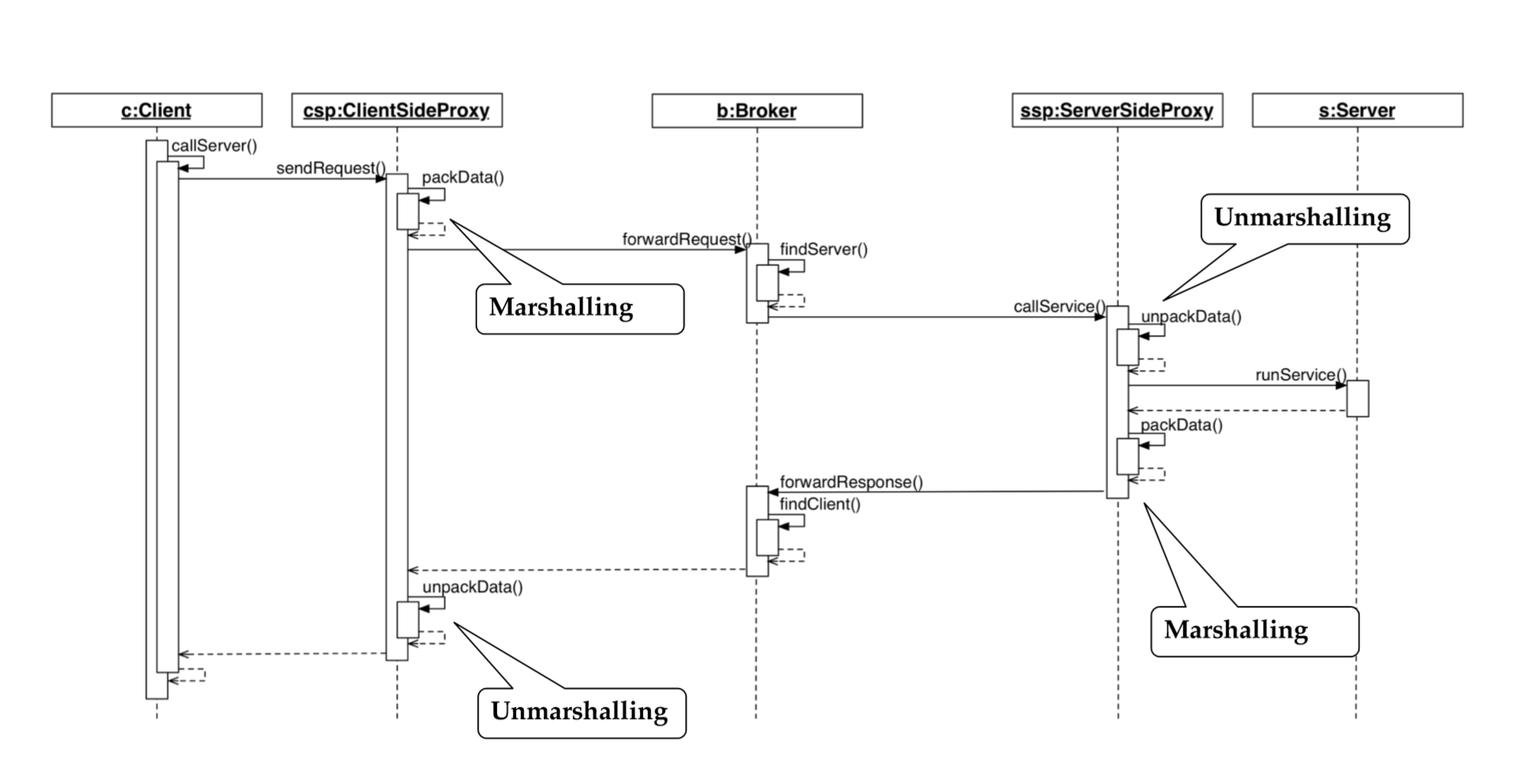
* Layers
  + What ?
    - Layered are the subsystems or the existing architecture. Closed if only one layer communicates the one below, open if it can communicate any layer bypassing the others.
  + Why ?
    - Reusability of Layers, Standardization, Low Coupling, Improved Testability
  + Example
    - 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

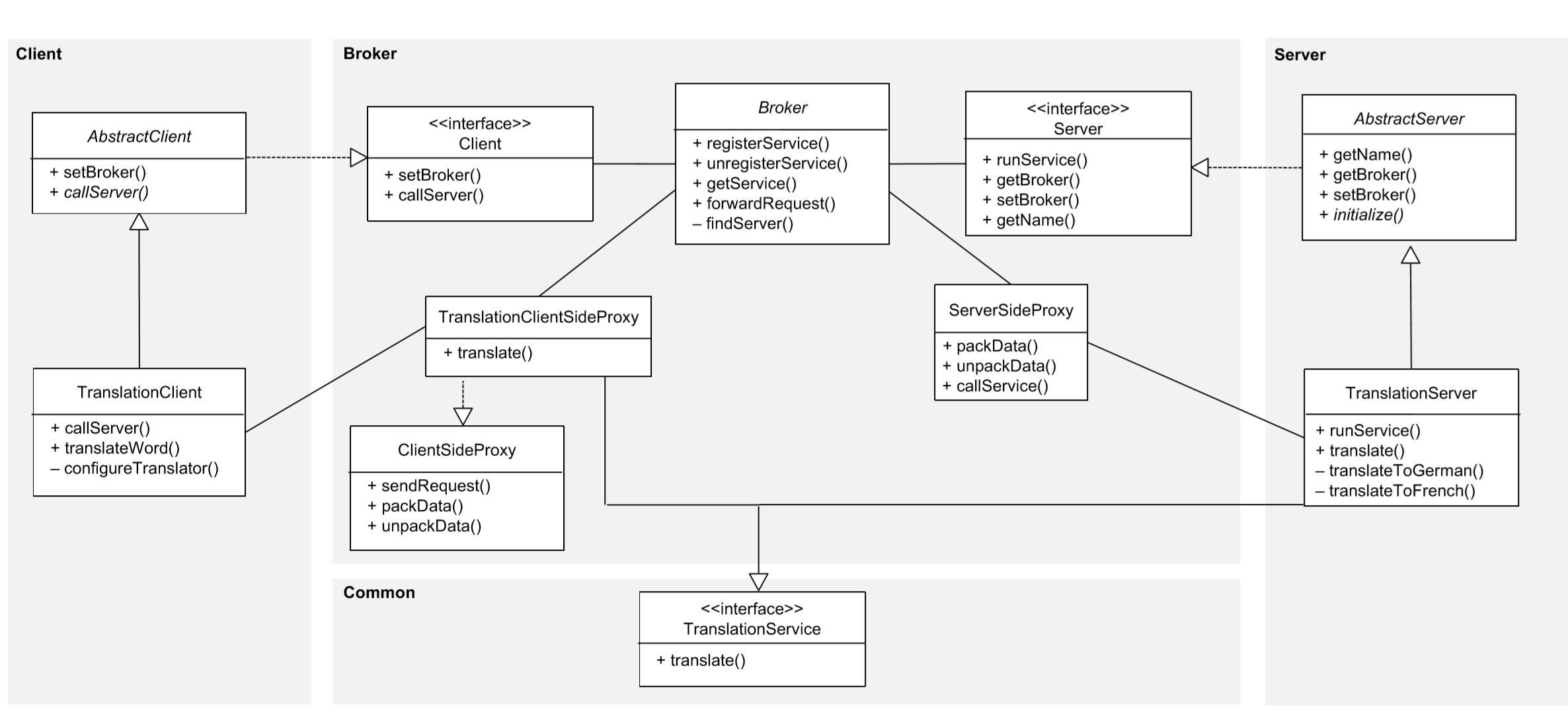
Distributed Systems

* 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
* Edge Computing

Interactive systems

* MVC

Testing Patterns

* Test Model
* Model based testing
  + Generation of a test model from the system model
  + Platform independent and platform dependent tests
* Testing activities
* Unit testing with Junit
  + Java Annotations and Assertions
* Object oriented test modelling
  + Object oriented test patterns (Junit)
  + Dependency Injection (Google juice)

Organizational Patterns

Antipatterns

Developer Antipattern

* The Blob
* Lava flow
* Spaghetti code
* Golden hammer
* Functional decomposition

Architectural antipattern

Vendor locking

Management Antipatterns

Analysis paralysis

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.