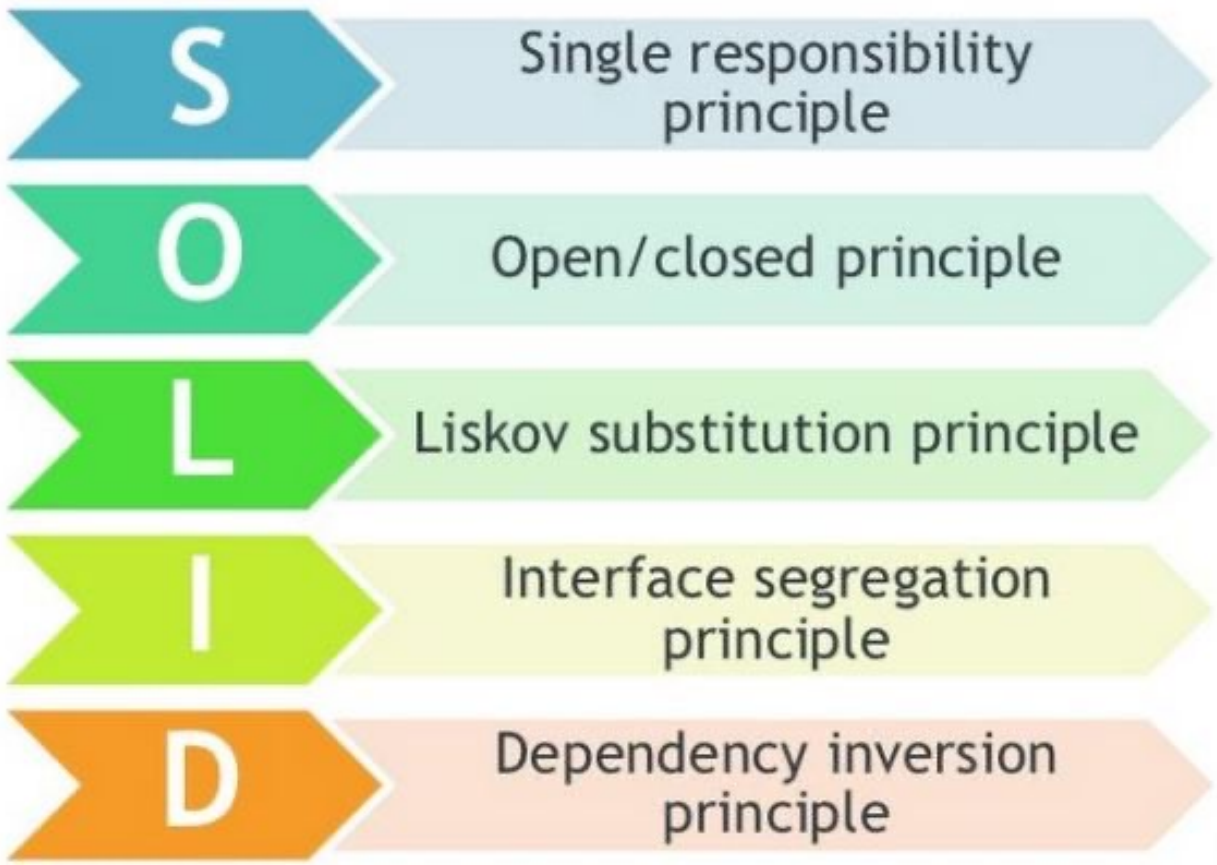
Design Patterns 3

Observer, Adaptor, Proxy, Decorator

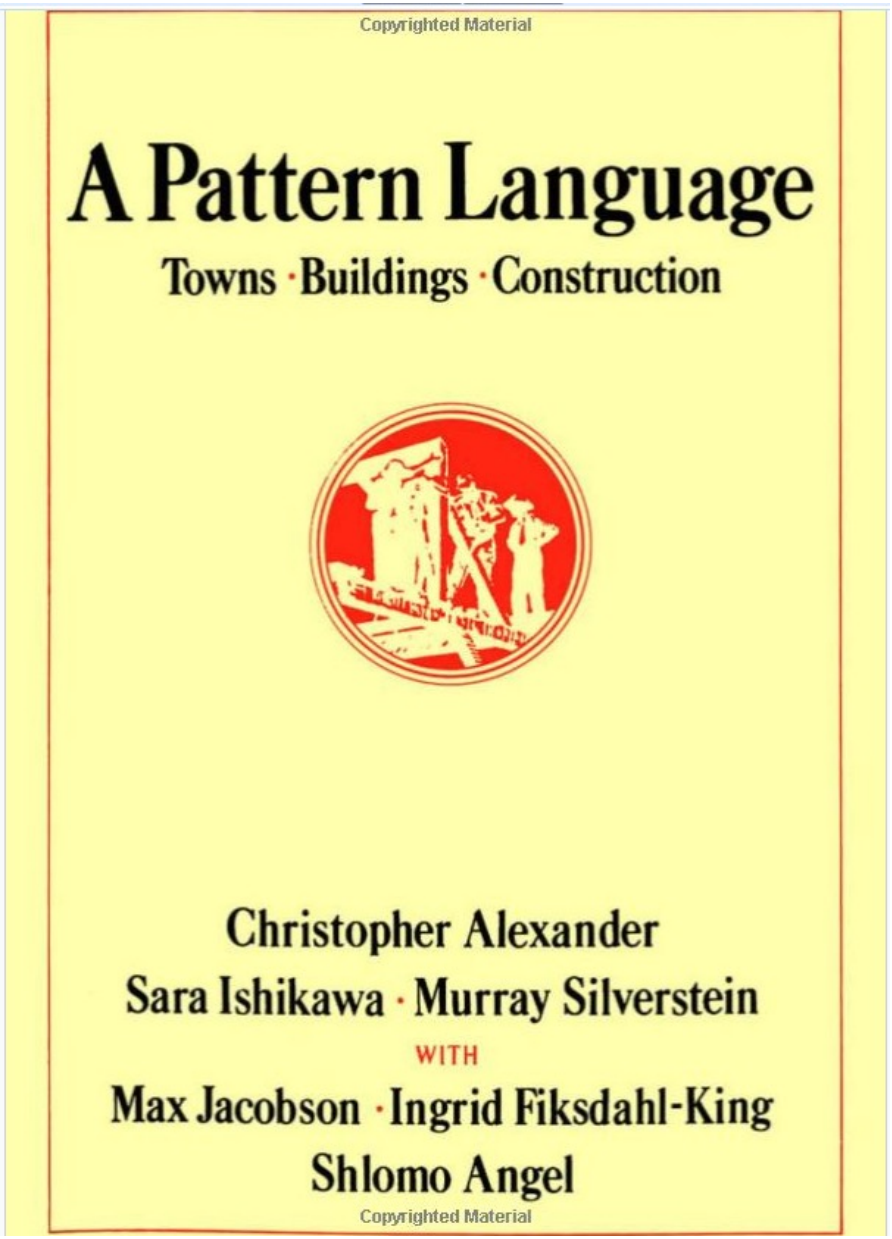
Shurui Zhou

The Edward S. Rogers Sr. Department
of Electrical & Computer Engineering
UNIVERSITY OF TORONTO

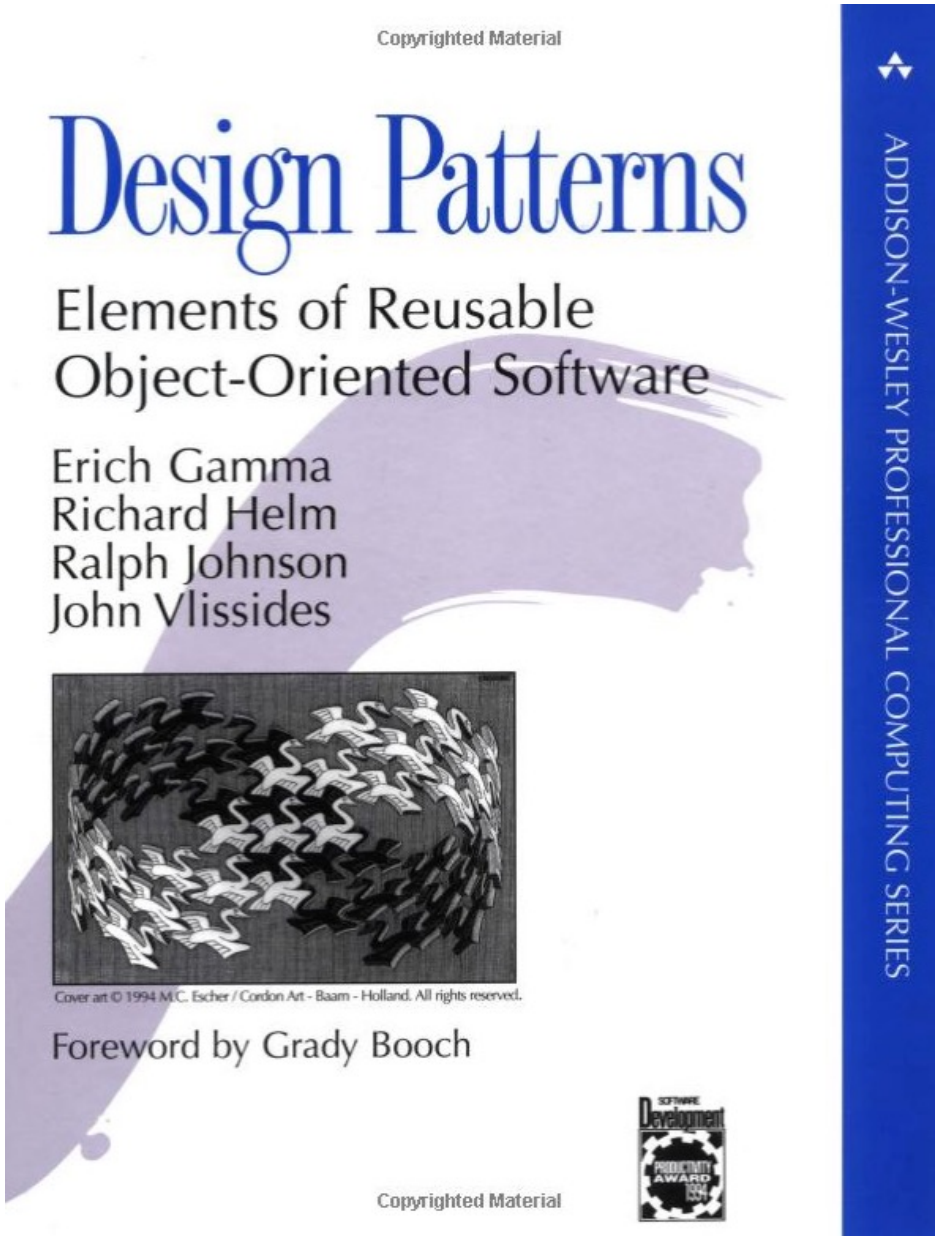
OO Design Principles



Building stable and flexible systems



- A “language” for designing the urban environment.
- The units of this language are patterns.
- window, building, etc..
- 253 design patterns



- 1994
- the GoF book -- the book by the gang of four
- Elements of Reusable Object-Oriented Software
- 23 OO patterns

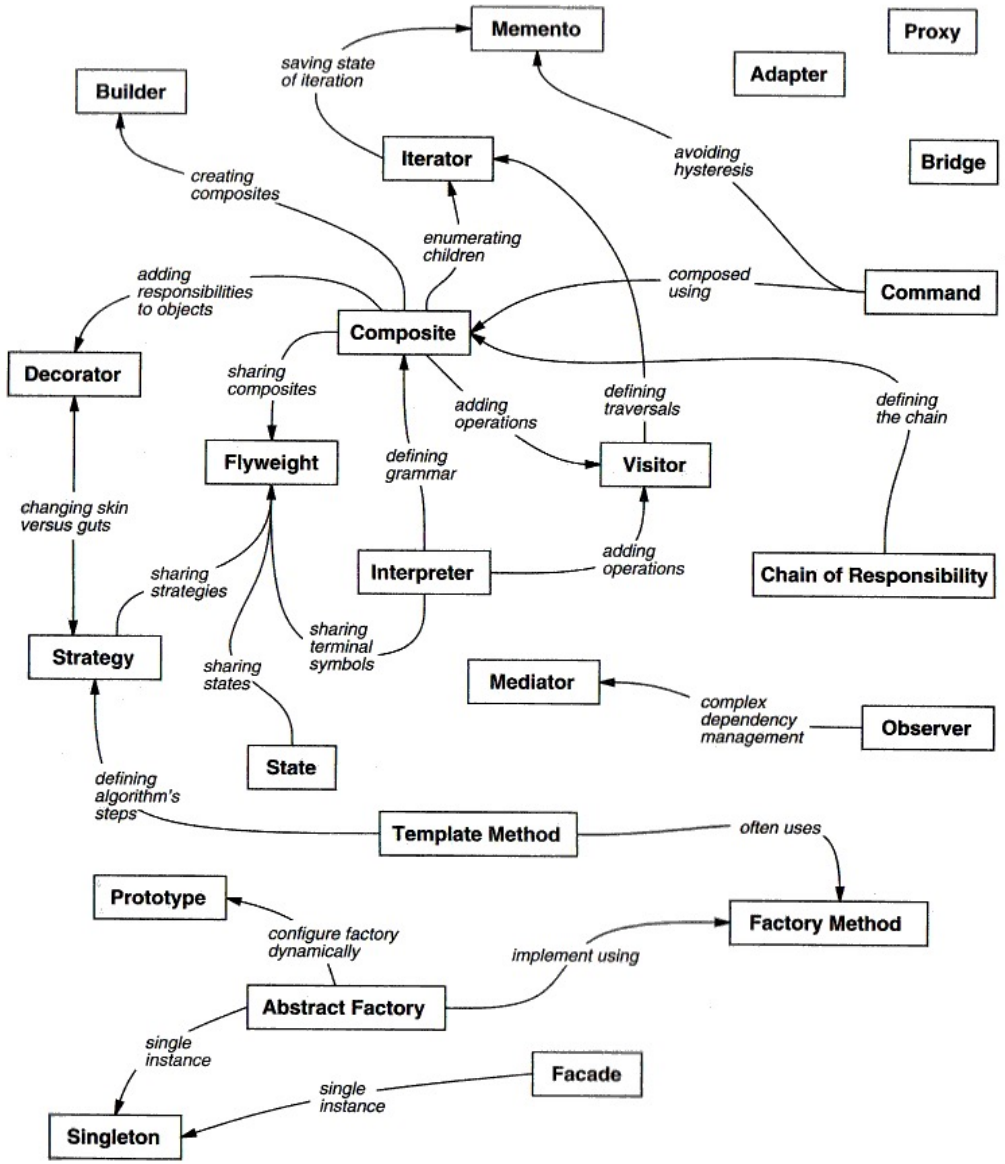


Figure 1.1: Design pattern relationships

Levels of Abstraction

- Requirements
  - high-level “what” needs to be done
- Architecture (High-level design)
  - high-level “how”, mid-level “what”
- OO-Design (Low-level design, e.g. design patterns)
  - mid-level “how”, low-level “what”
- Code
  - low-level “how”

Classification of patterns

- **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
- **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
- **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

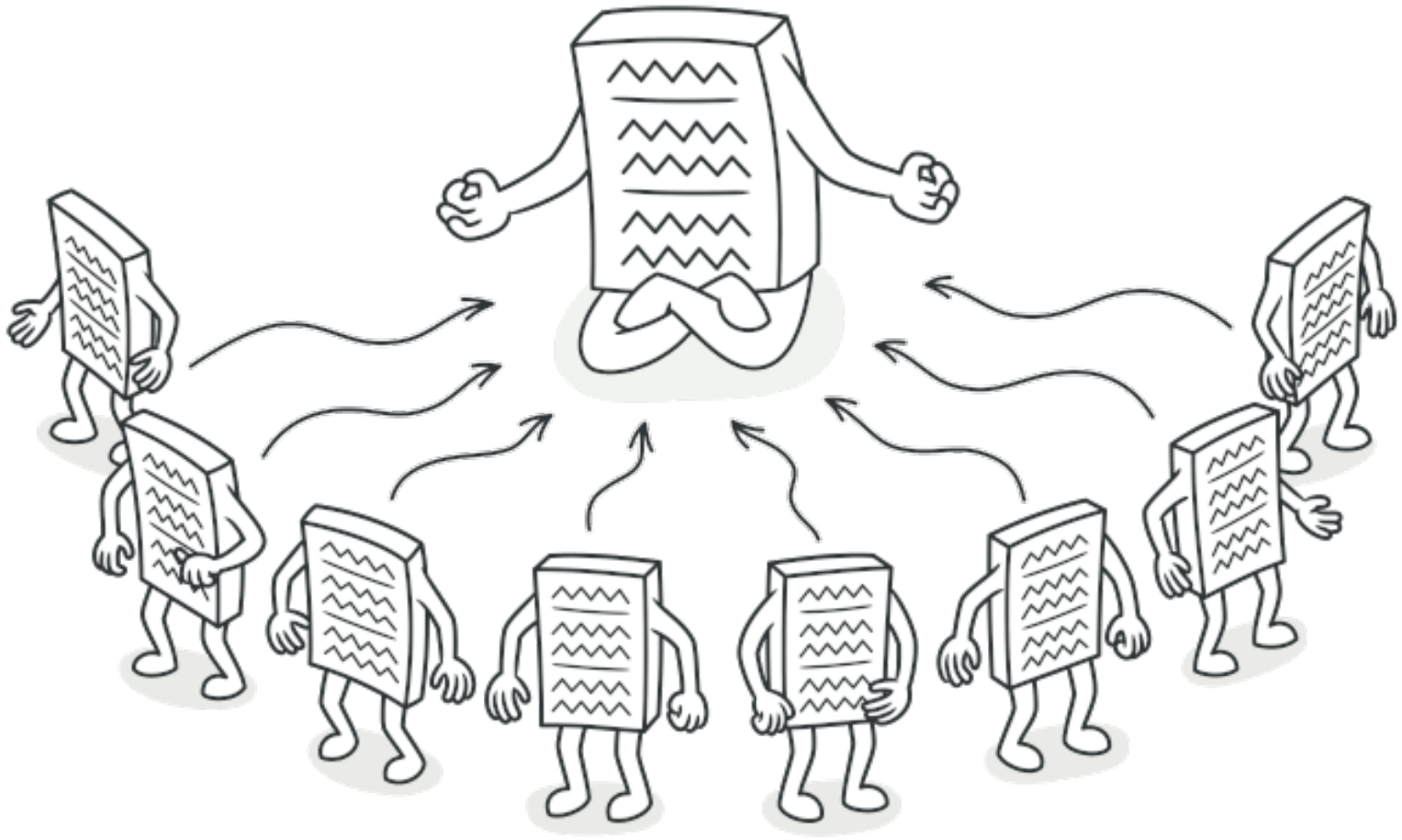
		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

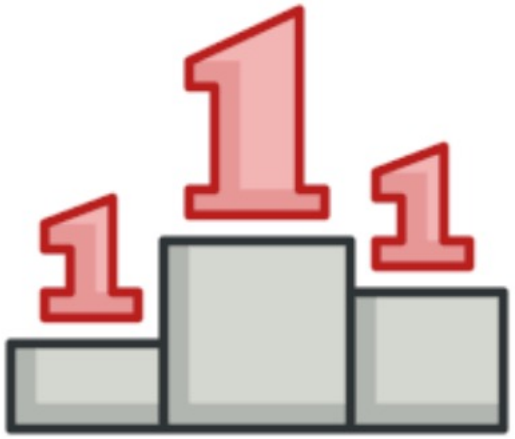
https://circle.visual-paradigm.com/catalog/

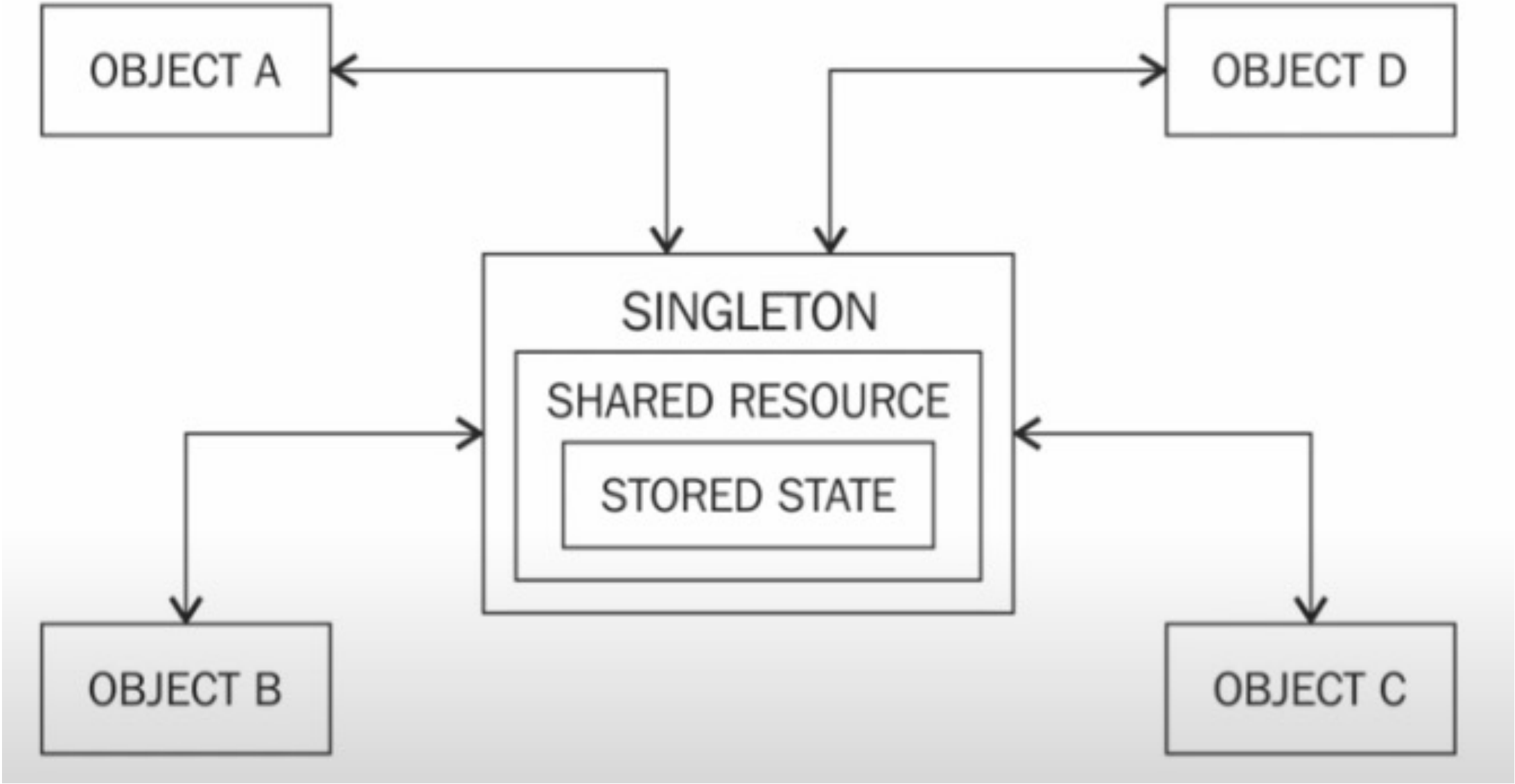
Classification of patterns

- **Creational patterns**
  - **Singleton**
  - **Factory Method**
- **Structural patterns**
  - **Composite**
- **Behavioral patterns**
  - **Strategy**

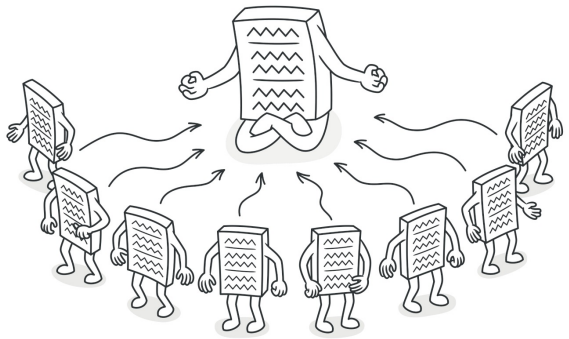
Singleton





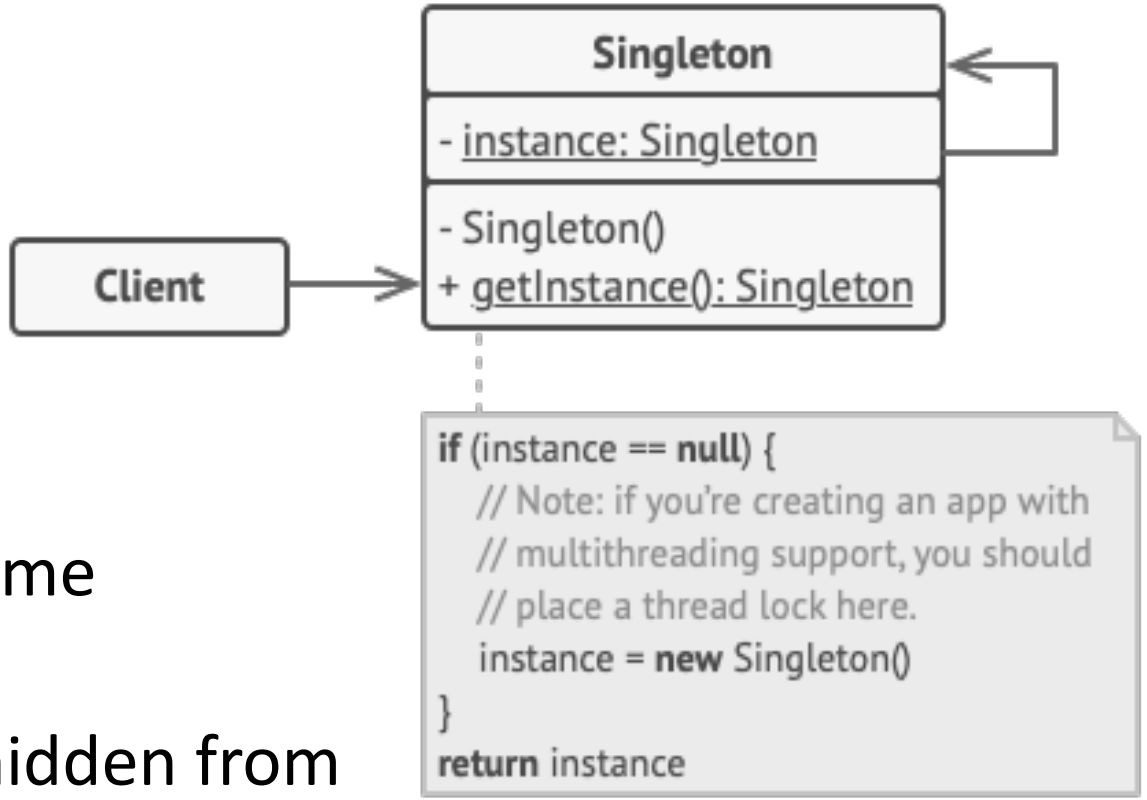

OBJECT A
OBJECT D
SINGLETON
SHARED RESOURCE
STORED STATE
OBJECT B
OBJECT C

Singleton



- a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.
- Example:
  - cache
  - thread pools
  - registries

Singleton



- The **Singleton** class declares the static method *getInstance* that returns the same instance of its own class.
- The Singleton's constructor should be hidden from the client code.
- Calling the *getInstance* method should be the only way of getting the Singleton object.

Singleton - Example

- java.lang.Runtime

Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the *getRuntime* method.

- **java.awt.Desktop#getDesktop()**
- **java.lang.System#getSecurityManager()**

Singleton: Pros and Cons

- ✓ You can be sure that a class has only a single instance.
- ✓ You gain a global access point to that instance.
- ✓ The singleton object is initialized only when it's requested for the first time.

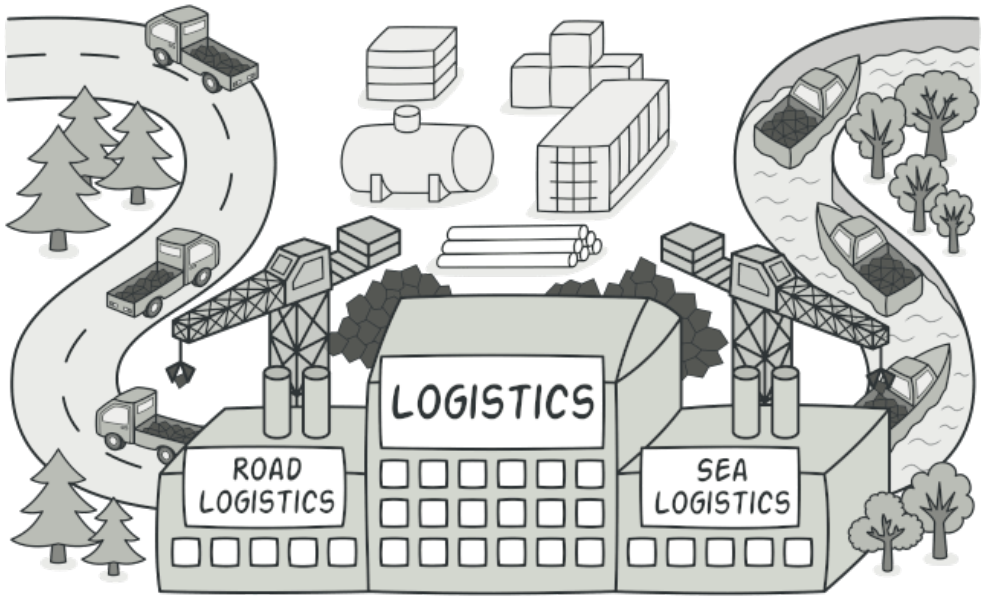
- ✗ Violates the *Single Responsibility Principle*. The pattern solves two problems at the time.
- ✗ The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other.
- ✗ The pattern requires special treatment in a multithreaded environment so that multiple threads won't create a singleton object several times.
- ✗ It may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages, you will need to think of a creative way to mock the singleton. Or just don't write the tests. Or don't use the Singleton pattern.

Classification of patterns

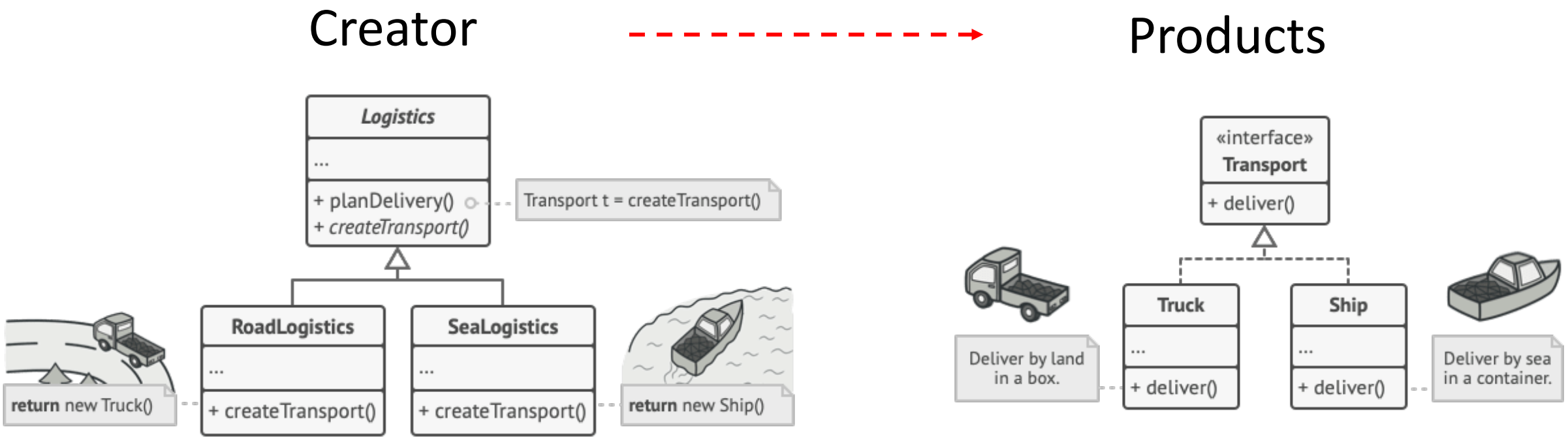
- **Creational patterns**
  - **Singleton**
  - **Factory Method**
- **Structural patterns**
  - **Composite**
- **Behavioral patterns**
  - **Strategy**

Factory Method (example)

a logistics management application

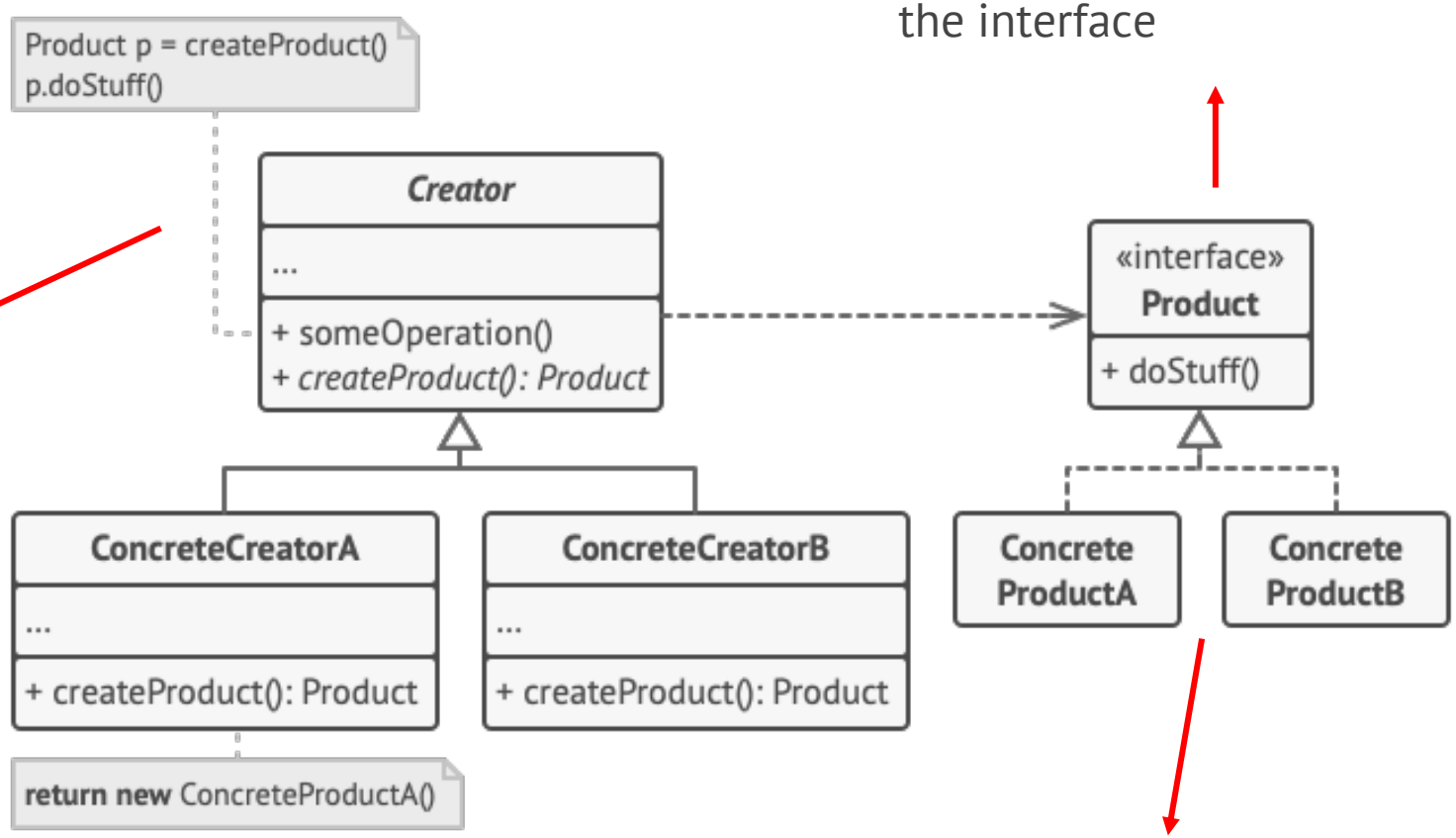


Factory Method



Factory Method

The **Product** declares the interface



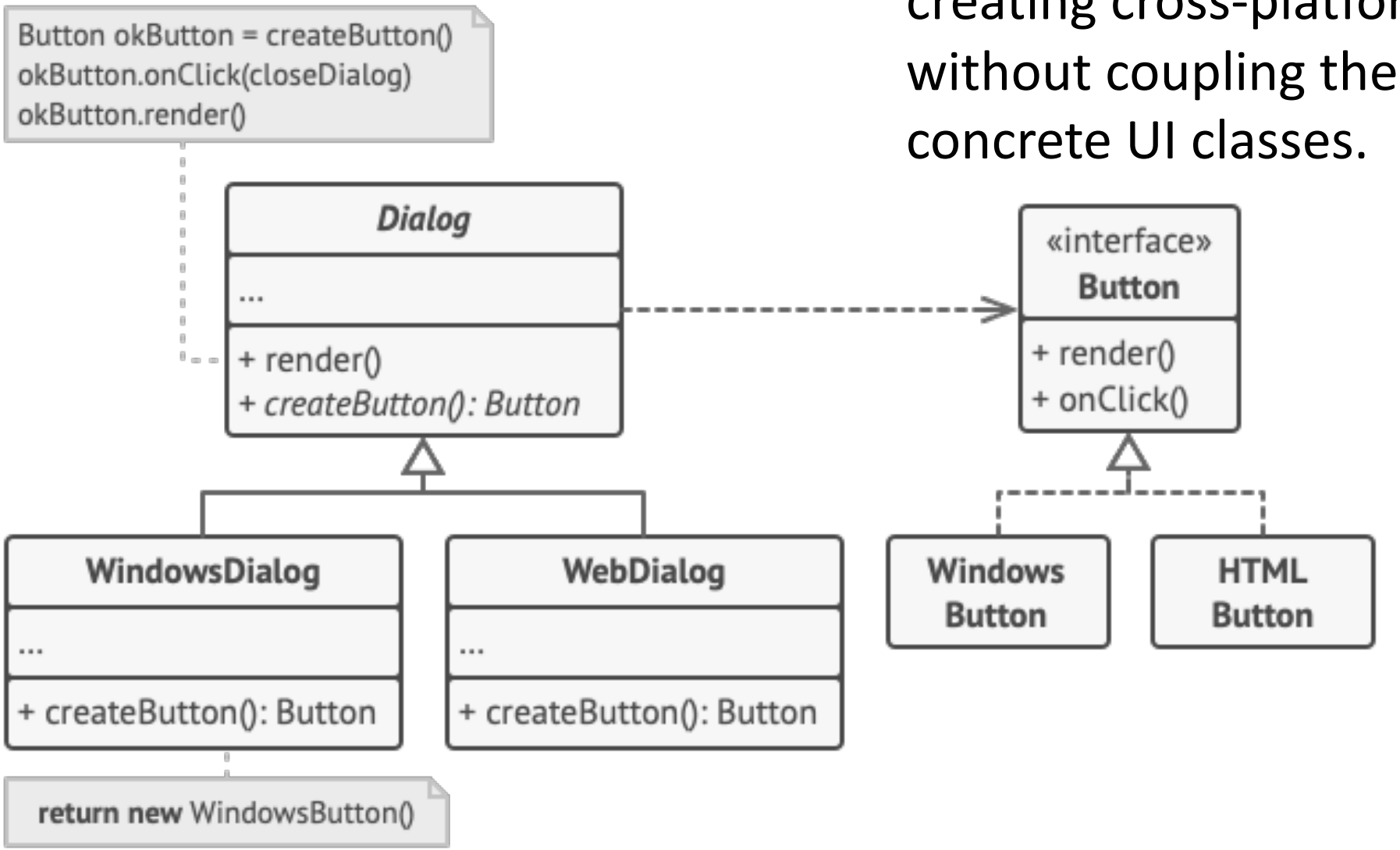
The **Creator** class declares the factory method that returns new product objects. It's important that the return type of this method matches the product interface.

Concrete Creators override the base factory method so it returns a different type of product. Note that the factory method doesn't have to **create** new instances all the time. It can also return existing objects from a cache, an object pool, or another source.

Concrete Products are different implementations of the product interface.

Factory Method - Example

creating cross-platform UI elements without coupling the client code to concrete UI classes.

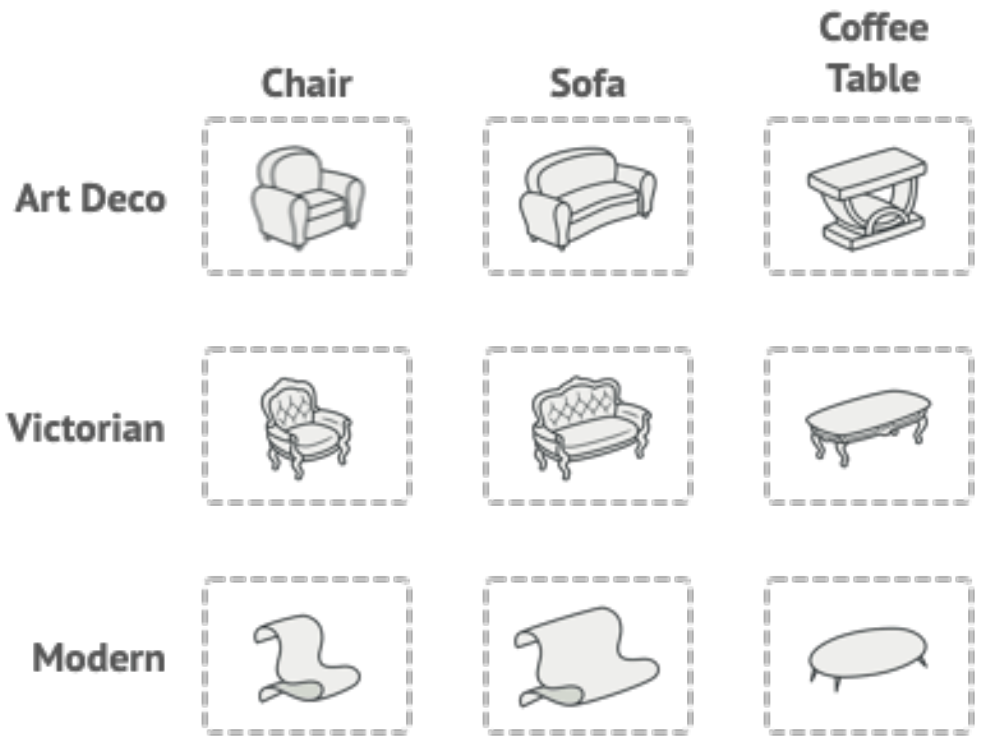


Factory Method – Pros and Cons

- ✓ You avoid tight coupling between the creator and the concrete products.
- ✓ *Single Responsibility Principle*. You can move the product creation code into one place in the program, making the code easier to support.
- ✓ *Open/Closed Principle*. You can introduce new types of products into the program without breaking existing client code.

- ✗ The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern. The best case scenario is when you're introducing the pattern into an existing hierarchy of creator classes.

Abstract Factory



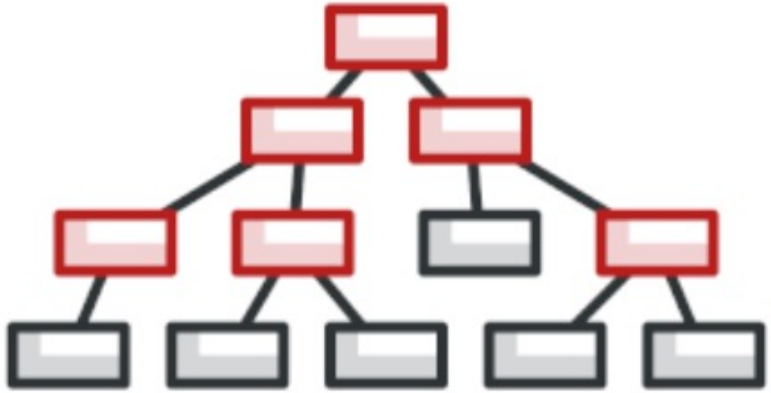
Creational patterns

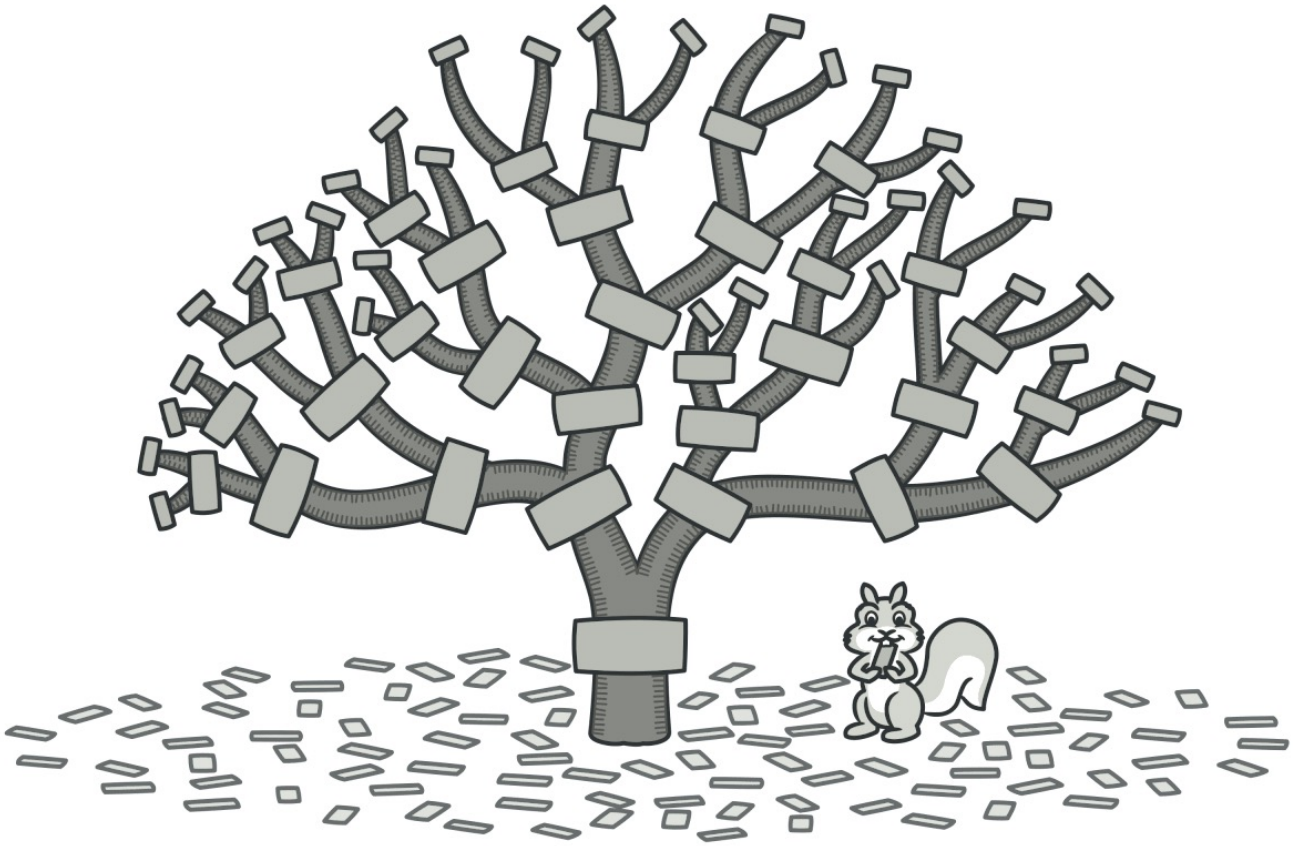
- **Abstract Factory**
  Creates an instance of several families of classes
- **Builder**
  Separates object construction from its representation
- **Factory Method**
  Creates an instance of several derived classes
- **Object Pool**
  Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
- **Prototype**
  A fully initialized instance to be copied or cloned
- **Singleton**
  A class of which only a single instance can exist

Classification of patterns

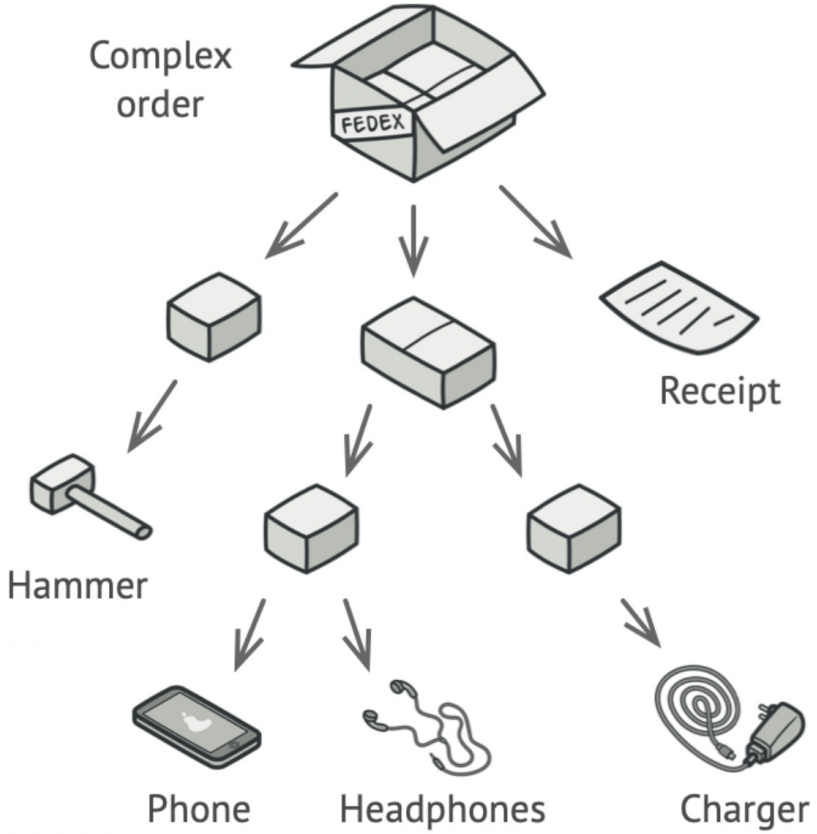
- **Creational patterns**
  - **Singleton**
  - **Factory Method**
- **Structural patterns**
  - **Composite**
- **Behavioral patterns**
  - **Strategy**

Composite Pattern





Composite Pattern - Problem

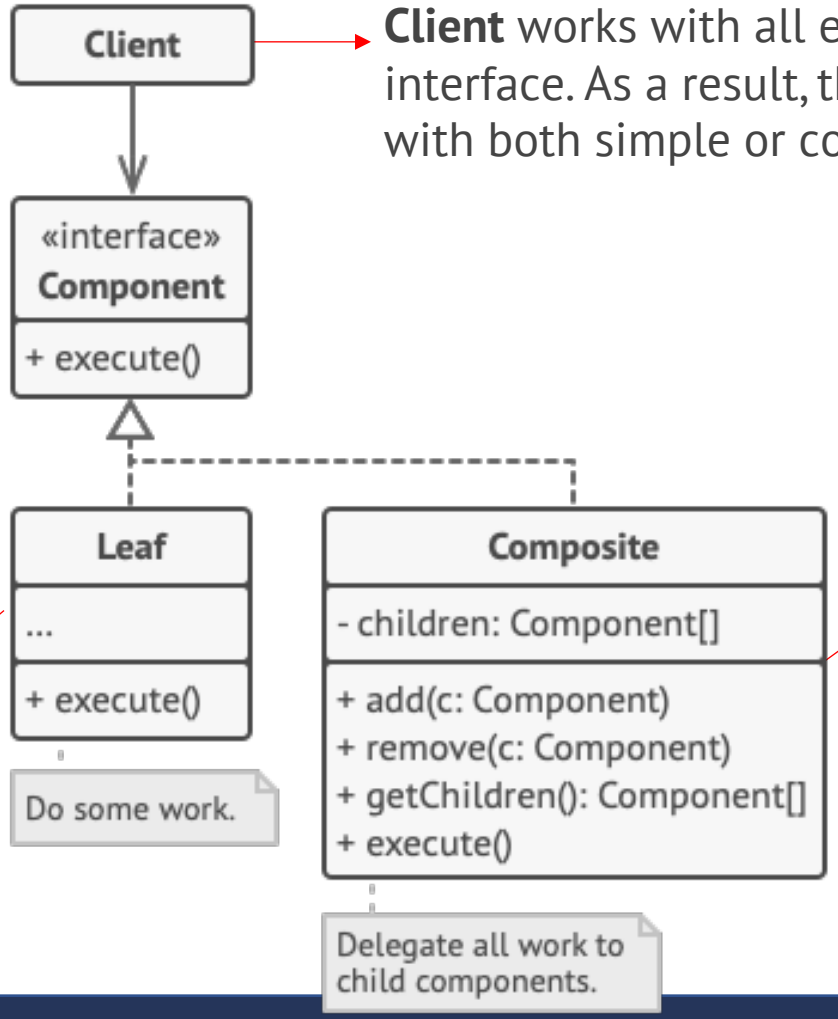


An Ordering System

- 2 types of Objects
  - Products
  - Boxes

Composite Design Pattern - Structure

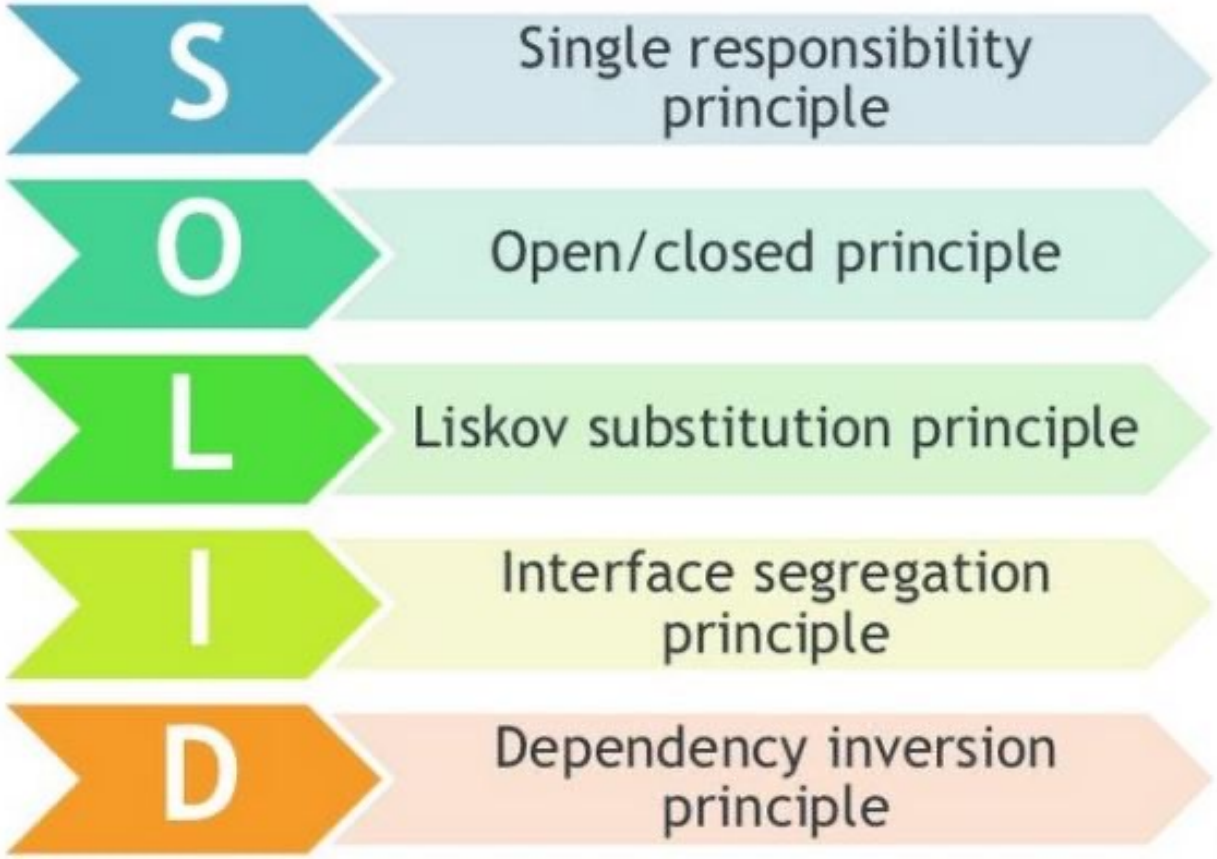
The **Component** interface describes operations that are common to both simple and complex elements of the tree.



Client works with all elements through the component interface. As a result, the client can work in the same way with both simple or complex elements of the tree.

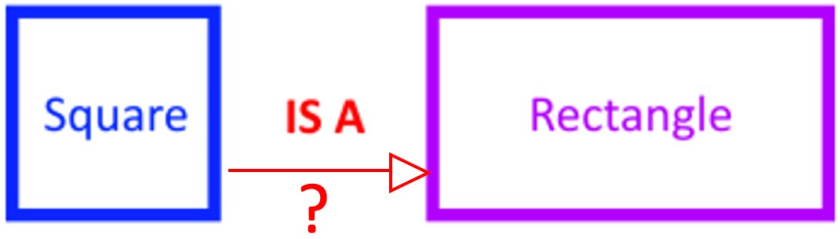
The **Leaf** is a basic element of a tree that doesn't have sub-elements.

The **Composite/container** is an element that has sub-elements: leaves or other containers. A container doesn't know the concrete classes of its children. It works with all sub-elements only via the component interface


S
Single responsibility principle
O
Open/closed principle
L
Liskov substitution principle
I
Interface segregation principle
D
Dependency inversion principle

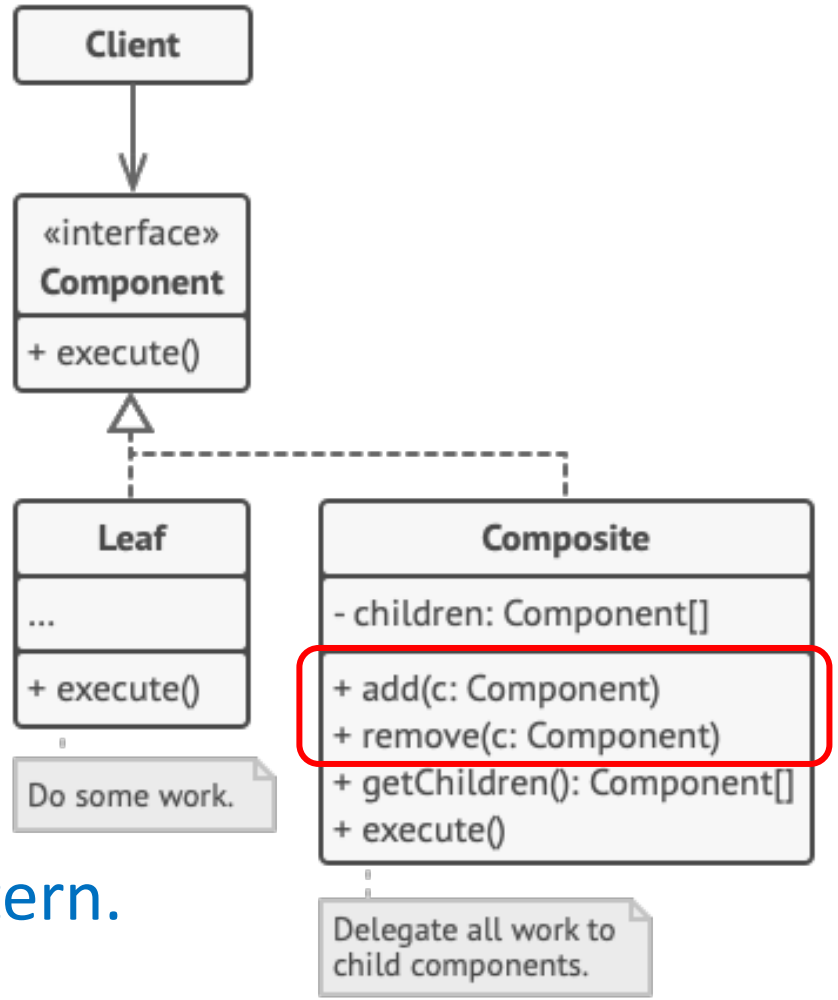
violates the Liskov substitution principle (LSP)

- Leaf inherits from Component so it will have an Add() method like any other Component.
- But Leafs don't have children, so the following method call cannot return a meaningful result:



Which classes *declare* add and remove children operation?

- Trade-off between safety and transparency
  - **Component**: transparency, because you can treat all components uniformly.
  - **Composite**: safety, because any attempt to add or remove objects from leaves will be caught at compile-time in a statically typed language



We emphasized transparency over safety in this pattern.

Composite – Pros & Cons

- ✓ You can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage.
- ✓ *Open/Closed Principle*. You can introduce new element types into the app without breaking the existing code, which now works with the object tree.
- ✗ It might be difficult to provide a common interface for classes whose functionality differs too much. In certain scenarios, you'd need to overgeneralize the component interface, making it harder to comprehend.

Classification of patterns

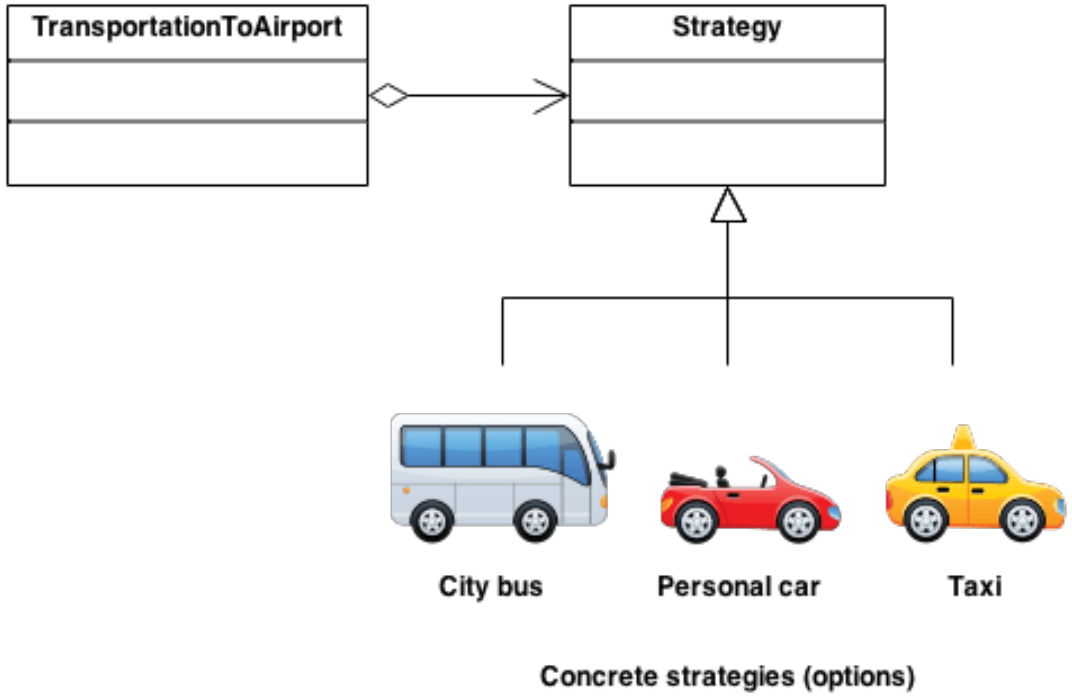
- **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
- **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
- **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

Classification of patterns

- **Creational patterns**
  - **Singleton**
  - **Factory Method**
- **Structural patterns**
  - **Composite**
- **Behavioral patterns**
  - **Strategy**

Strategy

- **Strategy** is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.



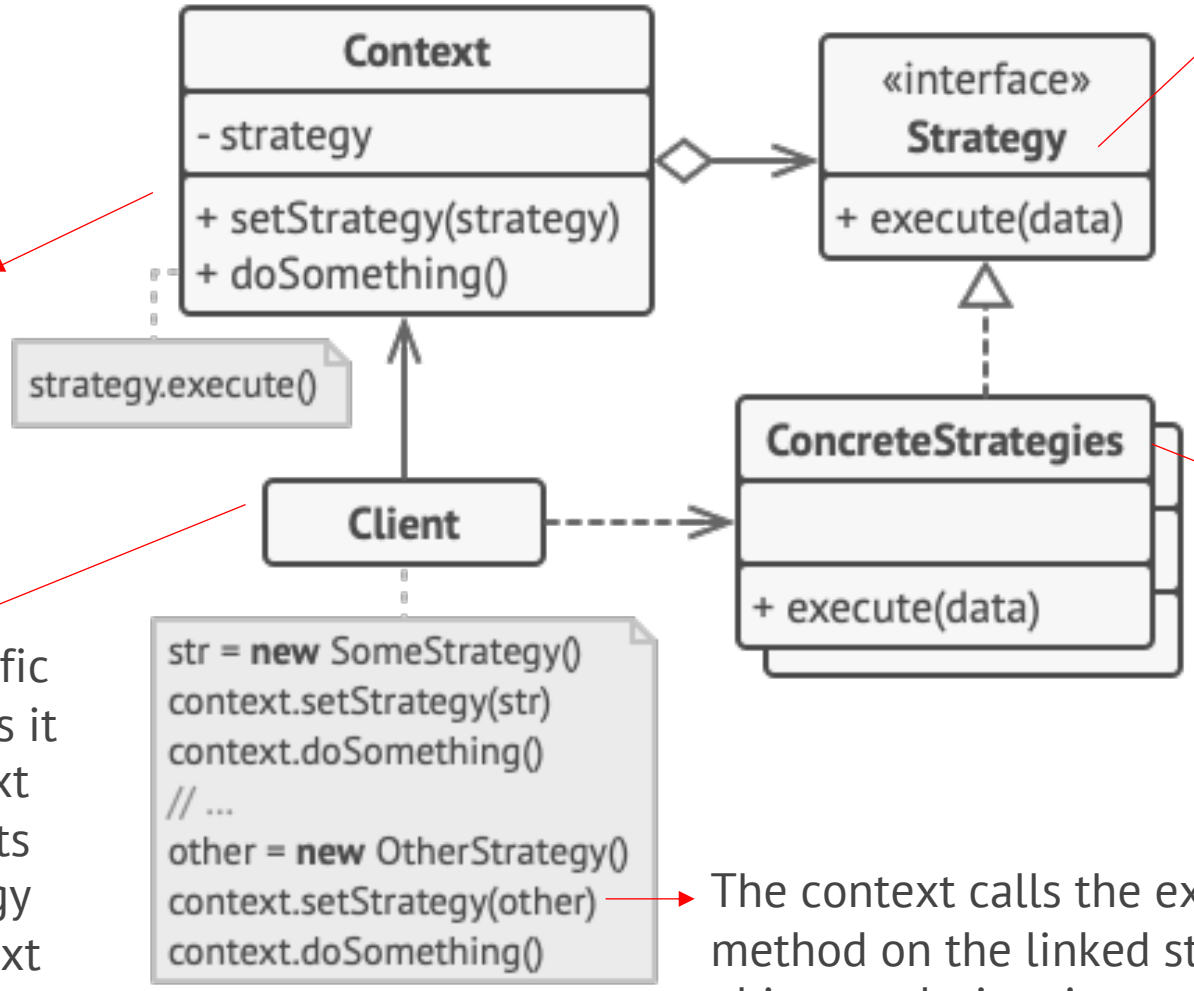
Navigation app
- automatic route planning

Strategy

- Grouping related algorithms under an abstraction, which allows switching out one algorithm or policy for another without modifying the client.
- Instead of directly implementing a single algorithm, the code receives runtime instructions specifying which of the group of algorithms to run.

Strategy

The **Context** maintains a reference to one of the concrete strategies and communicates with this object only via the strategy interface.



The **Strategy** interface is common to all concrete strategies. It declares a method the context uses to execute a strategy.

Concrete Strategies implement different variations of an algorithm the context uses.

The **Client** creates a specific strategy object and passes it to the context. The context exposes a setter which lets clients replace the strategy associated with the context at runtime.

The context calls the execution method on the linked strategy object each time it needs to run the algorithm. The context doesn't know what type of strategy it works with or how the algorithm is executed.

Strategy Pattern

- Sorting
- Layout manager in UI Toolkits
- Data compression
- in a game where we can have different characters and each character can have multiple weapons to attack but at a time can use only one weapon.
  - character as the context, for example King, Commander, Knight ,Soldier and weapon as a strategy where attack() could be the method/algorithm which depends on the weapons being used
  - concrete weapon classes were Sword, Axe, Crossbow, BowAndArrow etc .. they would all implement the attack() method

https://stackoverflow.com/questions/370258/real-world-example-of-the-strategy-pattern

Strategy – Pros & Cons

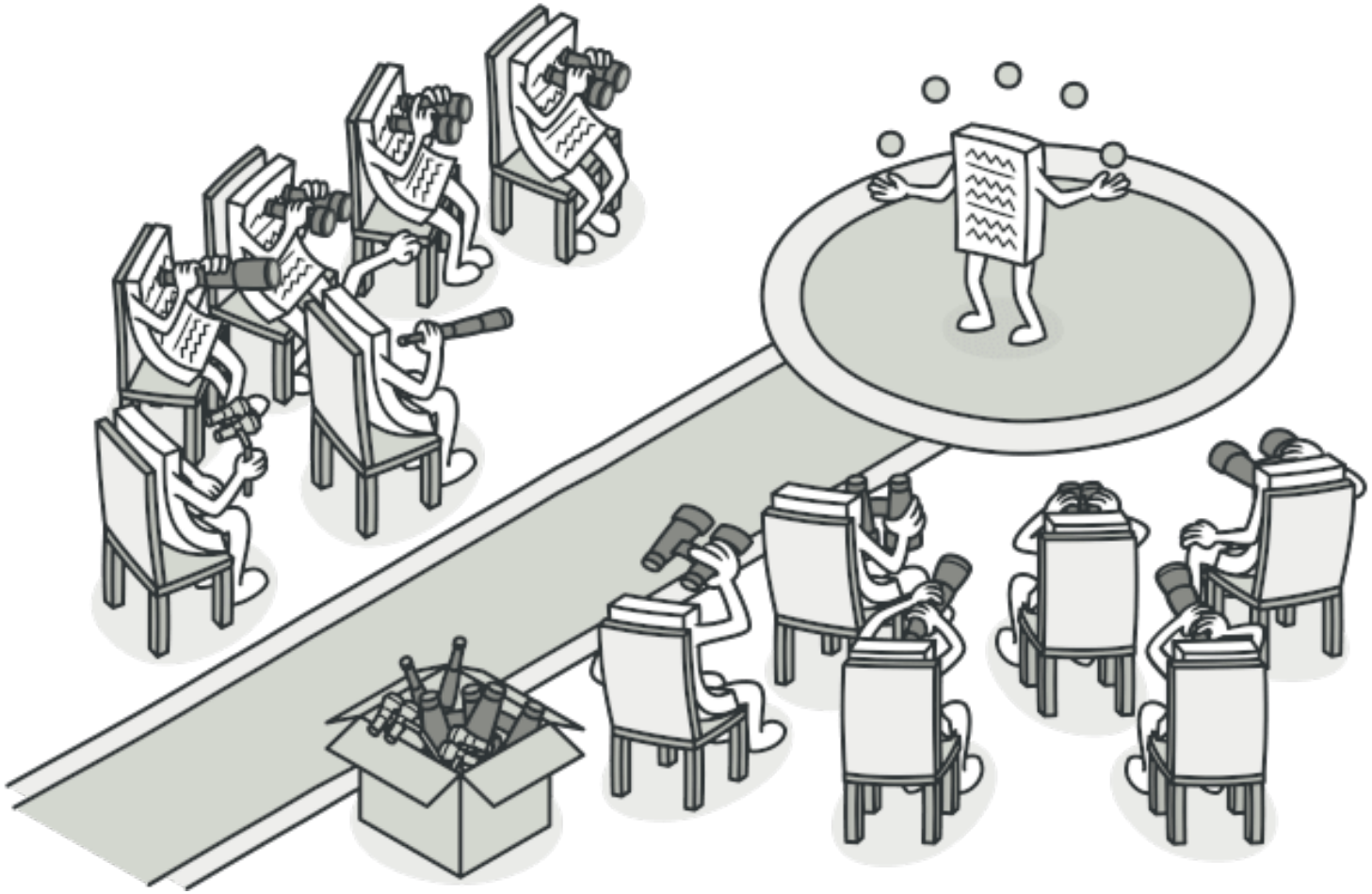
- ✓ You can swap algorithms used inside an object at runtime.
- ✓ You can isolate the implementation details of an algorithm from the code that uses it.
- ✓ You can replace inheritance with composition.
- ✓ *Open/Closed Principle*. You can introduce new strategies without having to change the context.

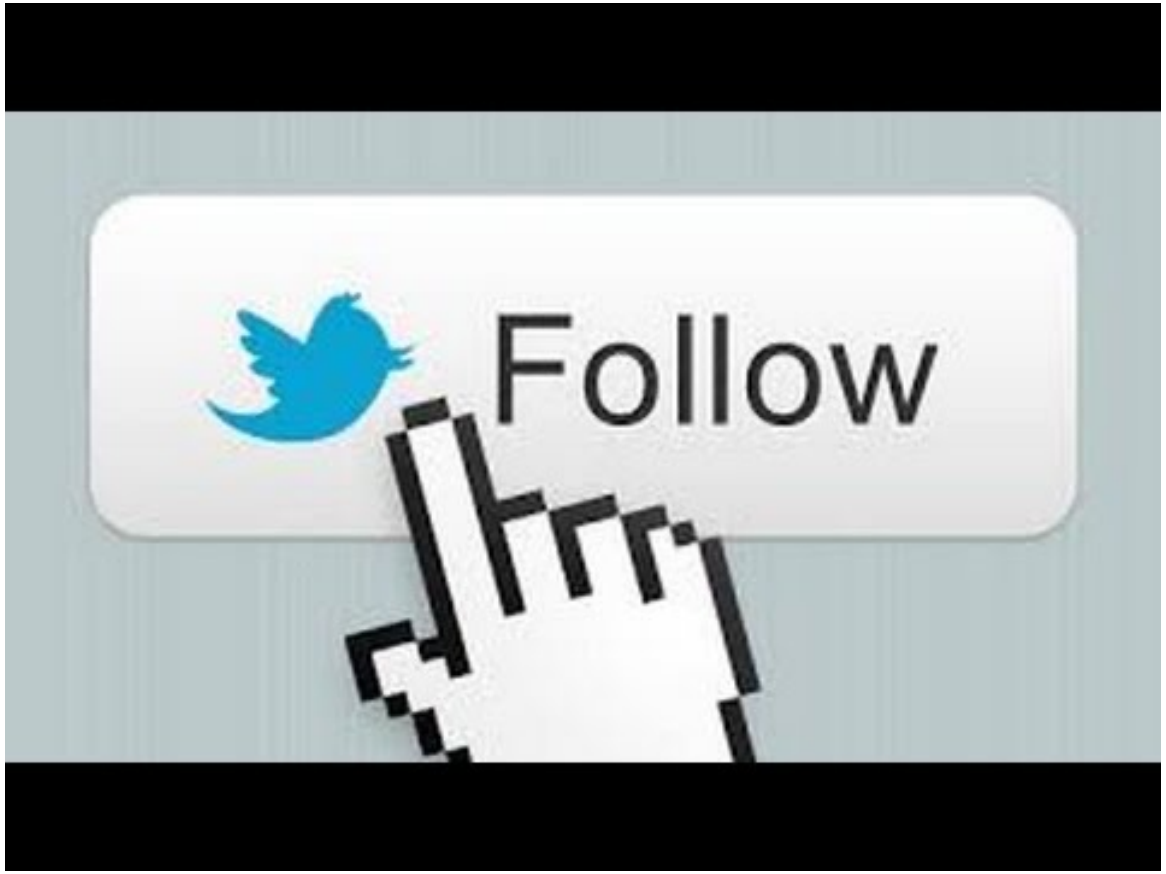
- ✗ If you only have a couple of algorithms and they rarely change, there's no real reason to overcomplicate the program with new classes and interfaces that come along with the pattern.
- ✗ Clients must be aware of the differences between strategies to be able to select a proper one.
- ✗ A lot of modern programming languages have functional type support that lets you implement different versions of an algorithm inside a set of anonymous functions. Then you could use these functions exactly as you'd have used the strategy objects, but without bloating your code with extra classes and interfaces.

Classification of patterns

- **Creational patterns**
  - Singleton
  - Factory Method
- **Structural patterns**
  - Composite
- **Behavioral patterns**
  - Strategy
  - Observer

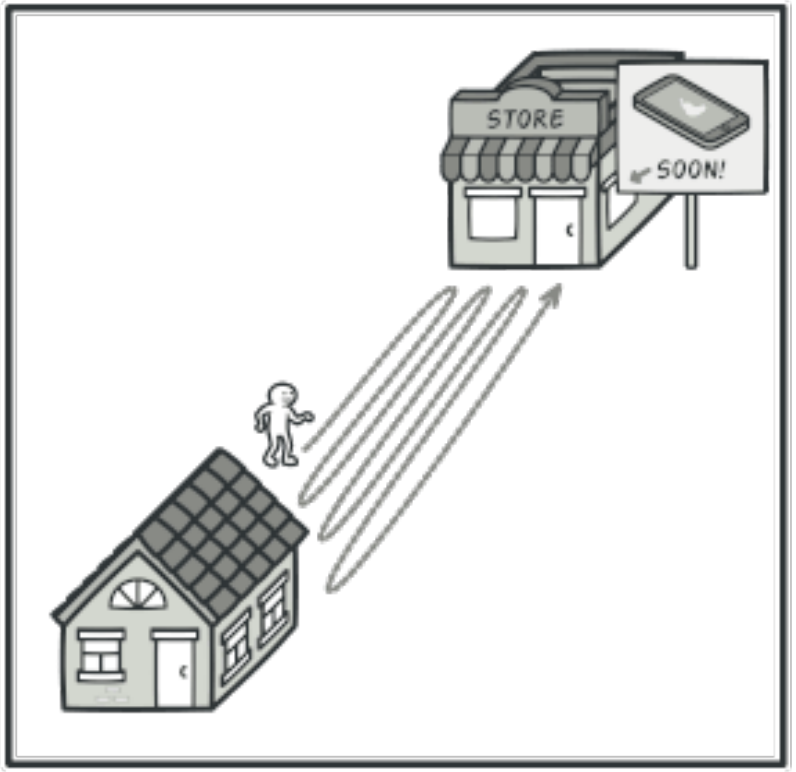
Observer Pattern

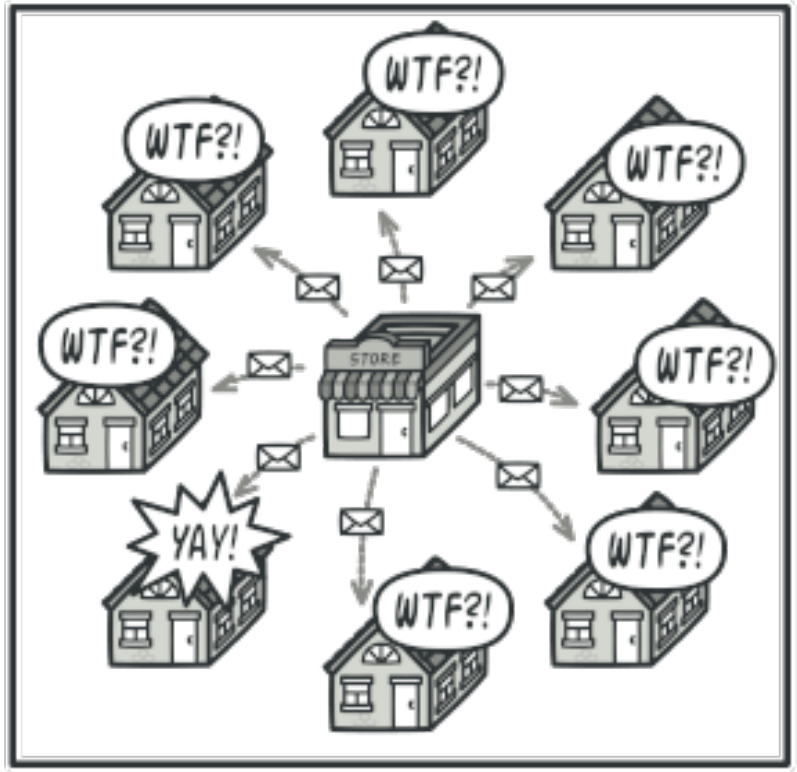




The observer + The subject

Observer Pattern





Visiting the store vs. sending spam

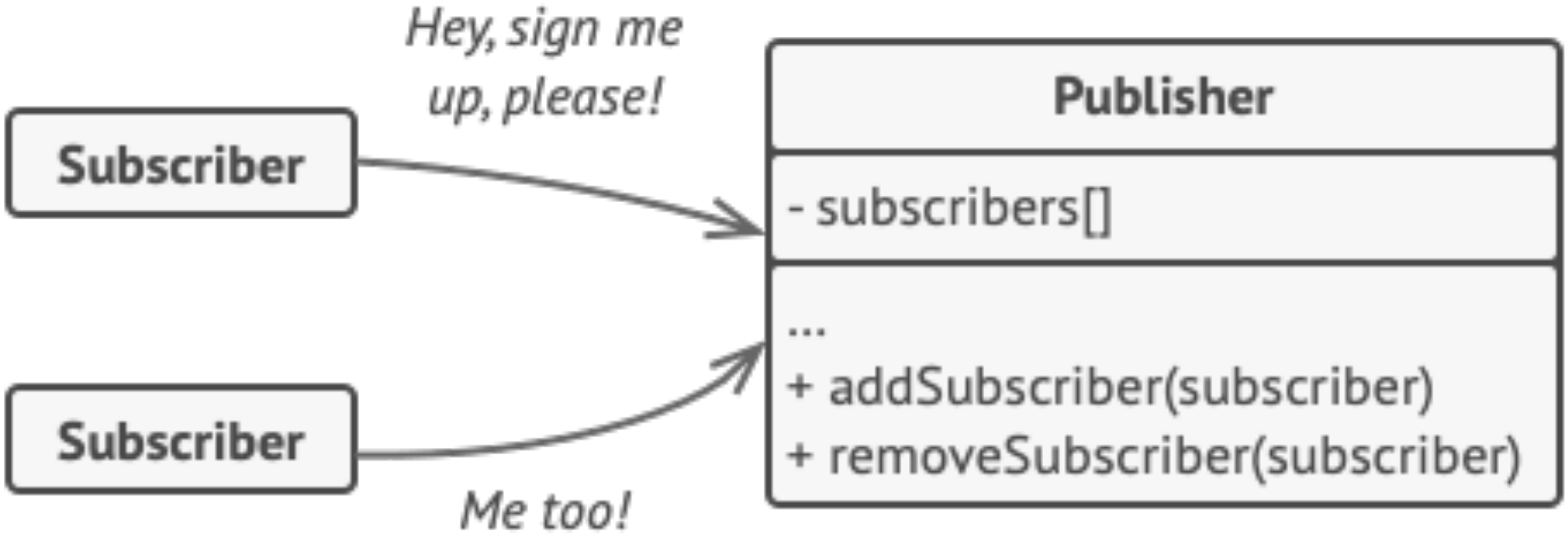
Observer Pattern

- **Observer** is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.
- Publishers + Subscribers = Observer Pattern

How newspaper or magazine subscriptions work?

1. A newspaper publisher goes into business and begins publishing newspapers.
2. You subscribe to a particular publisher, and every time there's a new edition it gets delivered to you. As long as you remain a subscriber, you get new newspapers.
3. You unsubscribe when you don't want papers anymore, and they stop being delivered.
4. While the publisher remains in business, people, hotels, airlines, and other businesses constantly subscribe and unsubscribe to the newspaper.

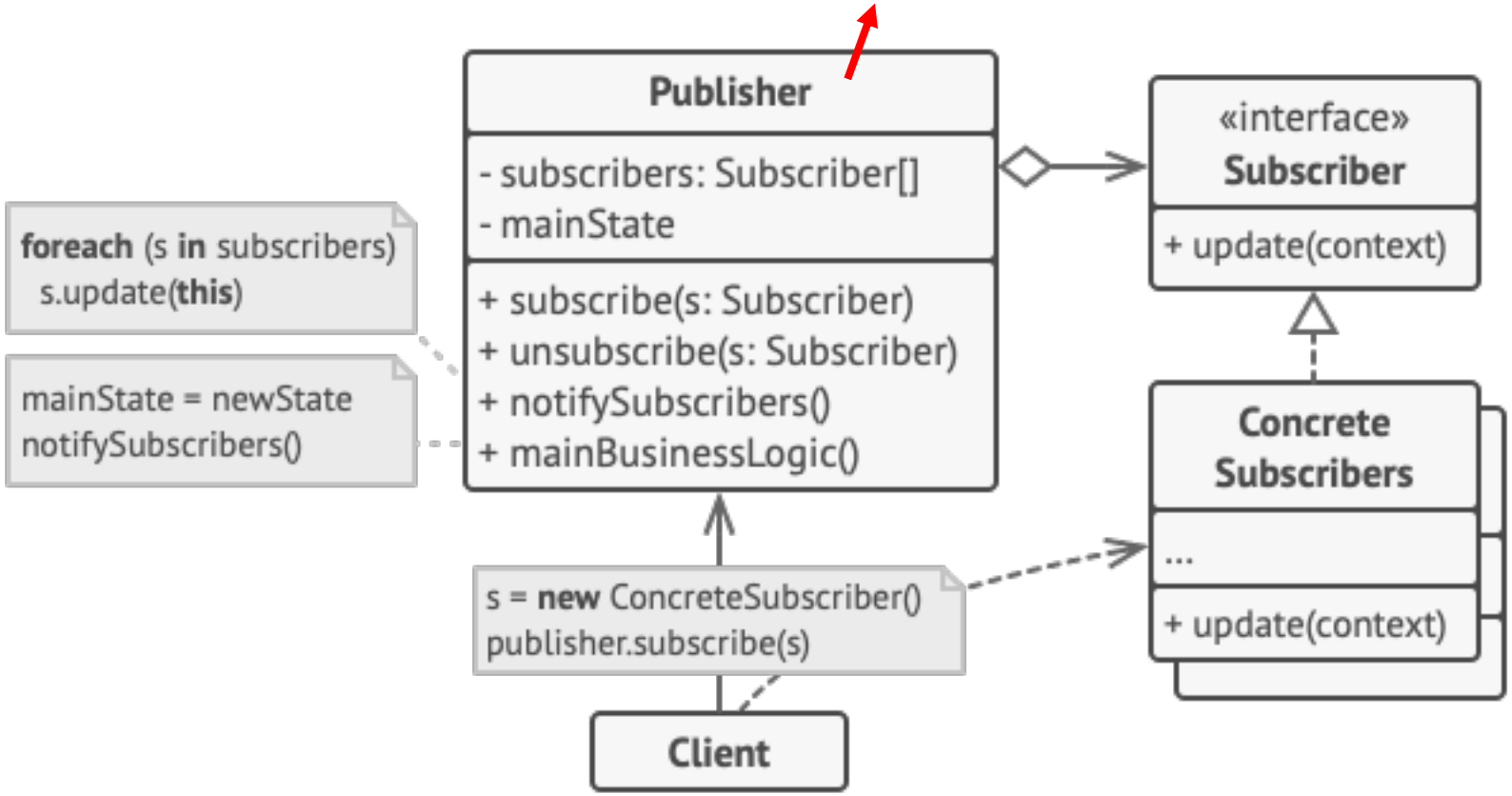
Observer Pattern



All subscribers implement the same interface and that the publisher communicates with them only via that interface.

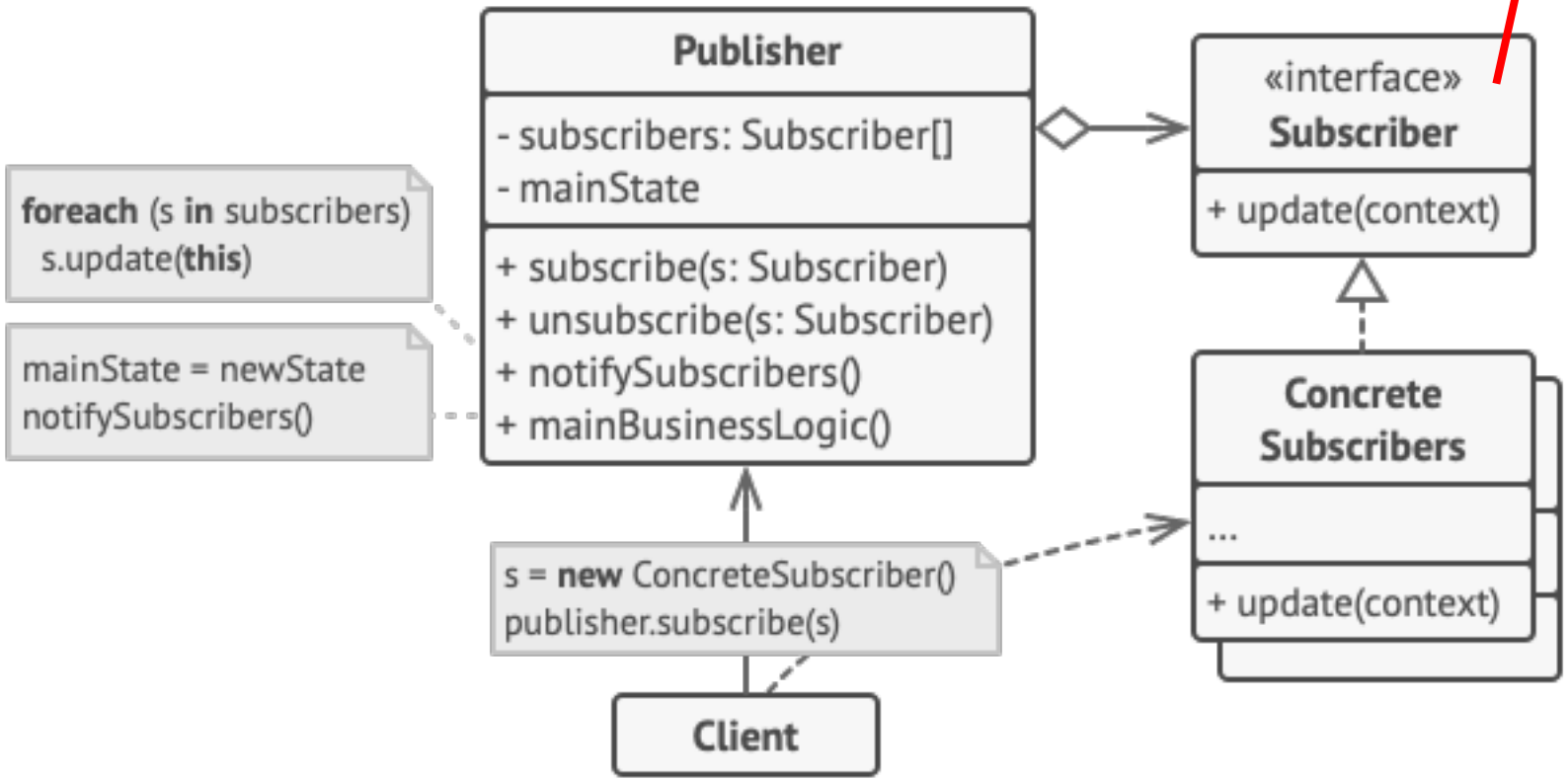
Observer Pattern

The **Publisher** issues events of interest to other objects. These events occur when the publisher changes its state or executes some behaviors.



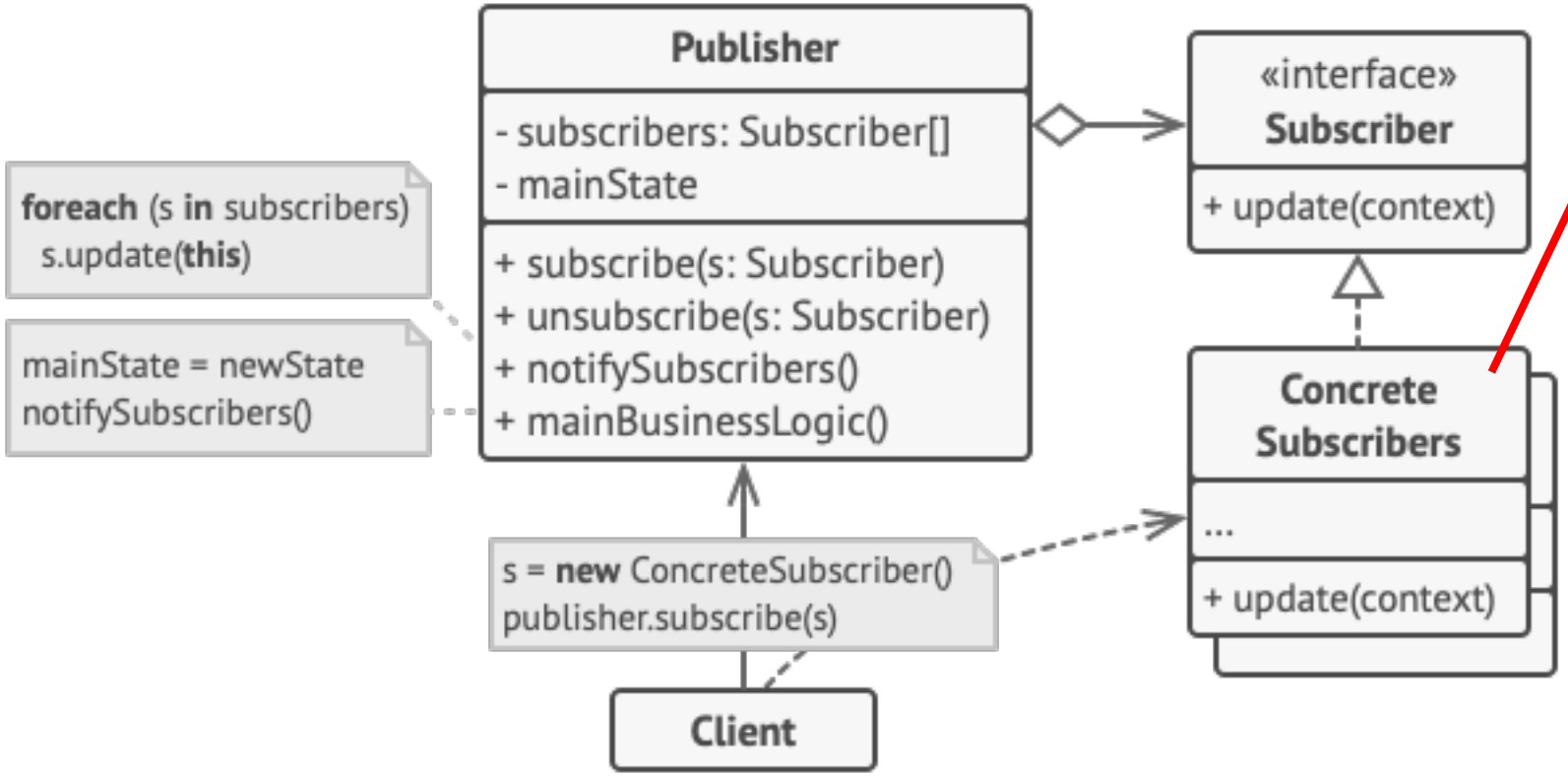
Observer Pattern

The **Subscriber** interface declares the notification interface. In most cases, it consists of a single update method.

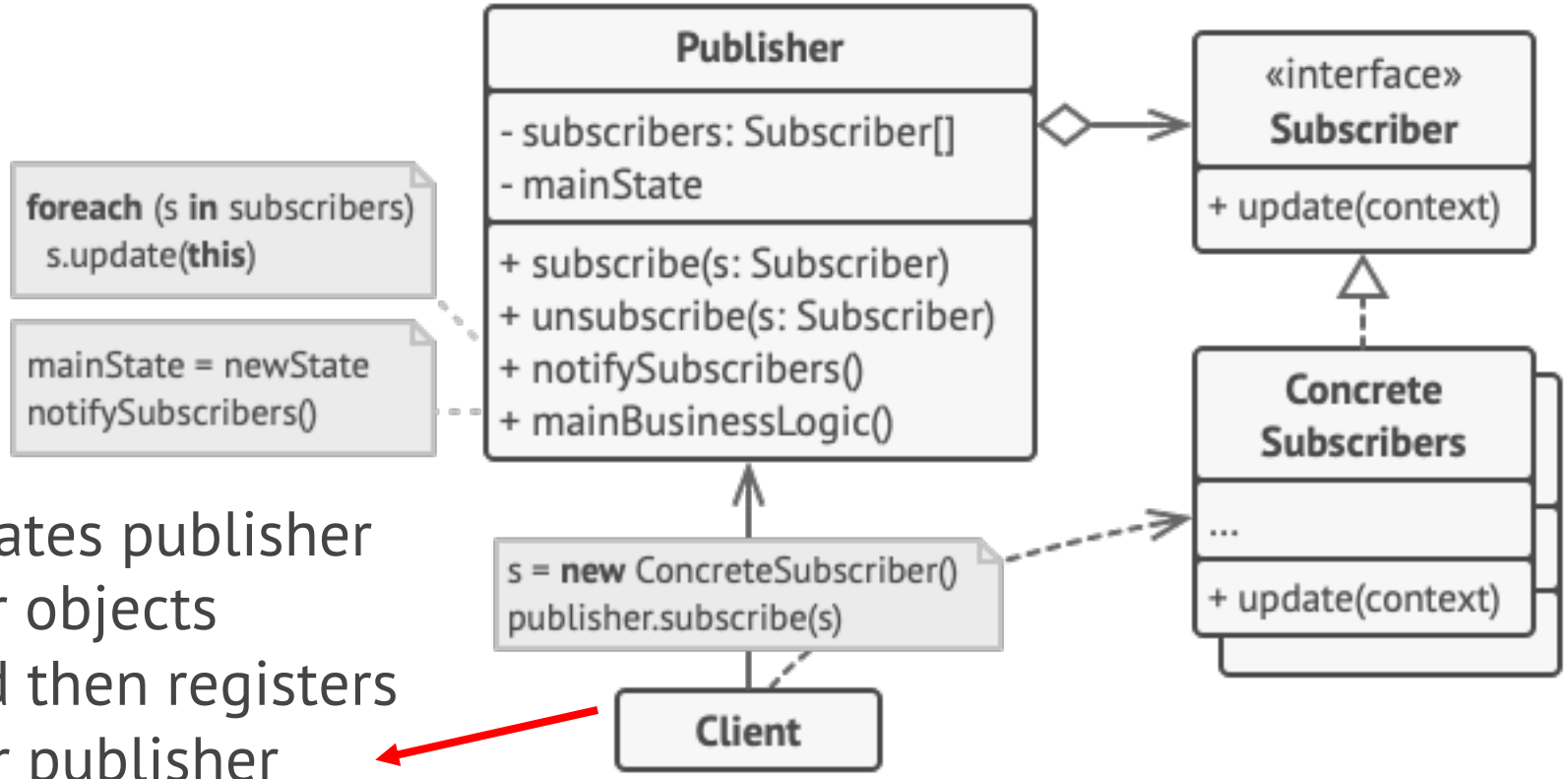


Observer Pattern

Concrete Subscribers implement the same interface so the publisher isn't coupled to concrete classes.

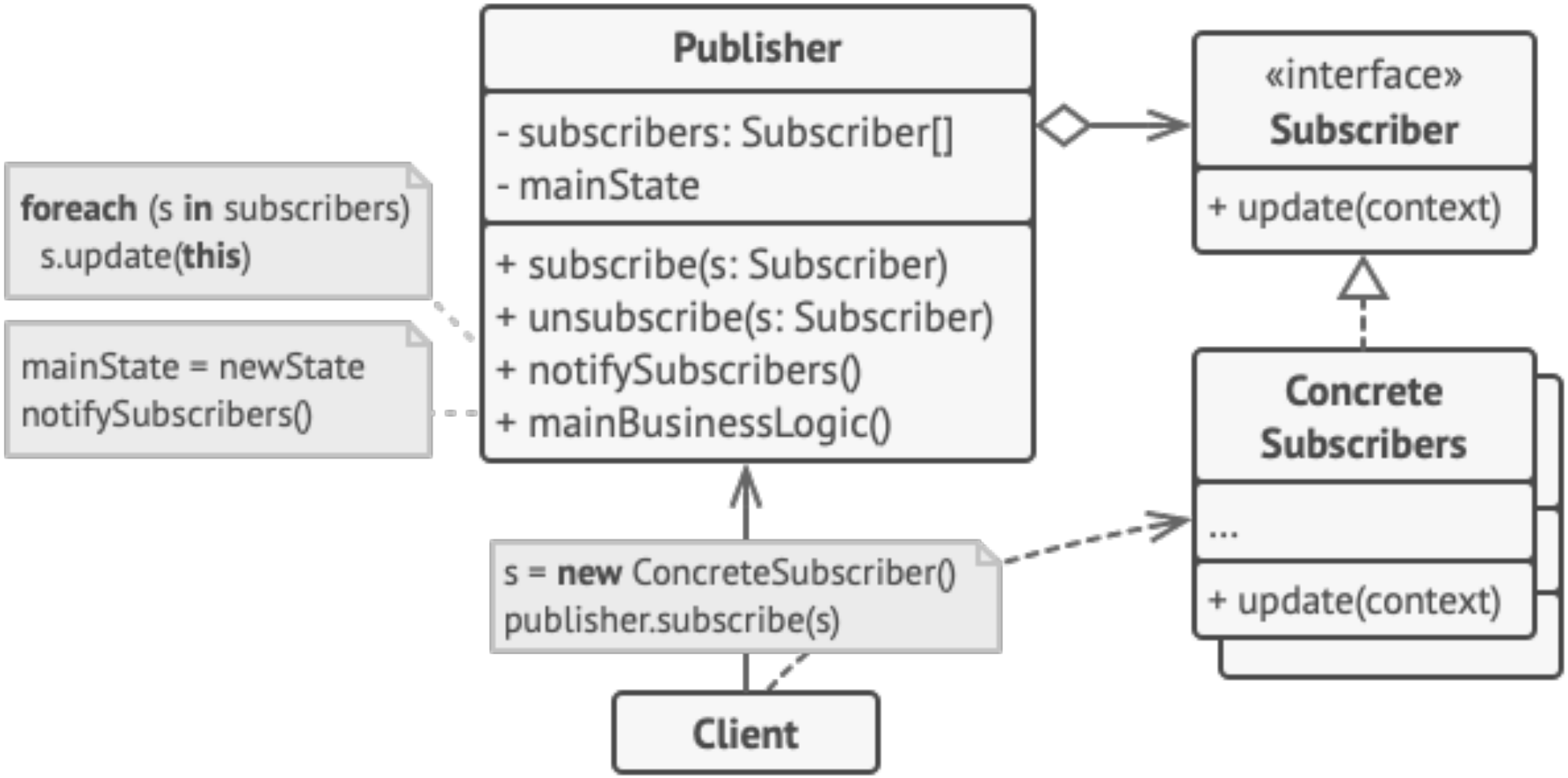


Observer Pattern



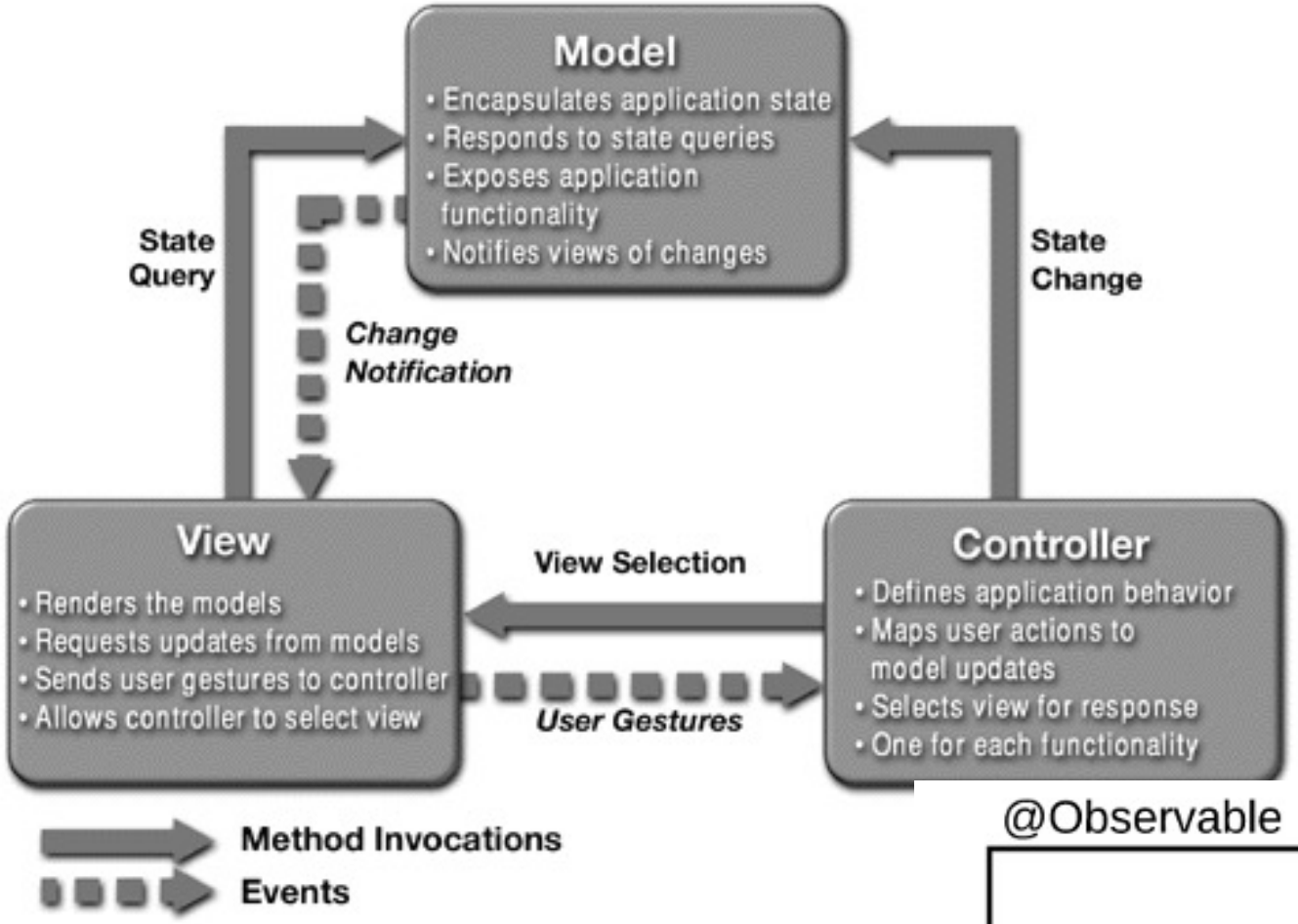
The **Client** creates publisher and subscriber objects separately and then registers subscribers for publisher updates.

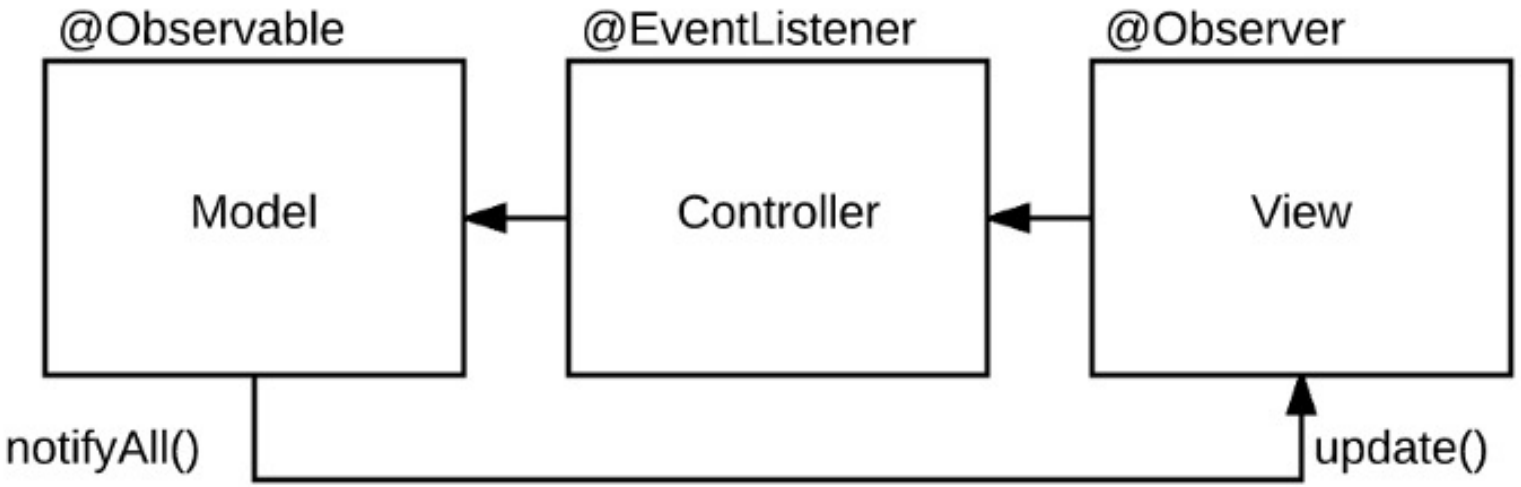
Observer Pattern

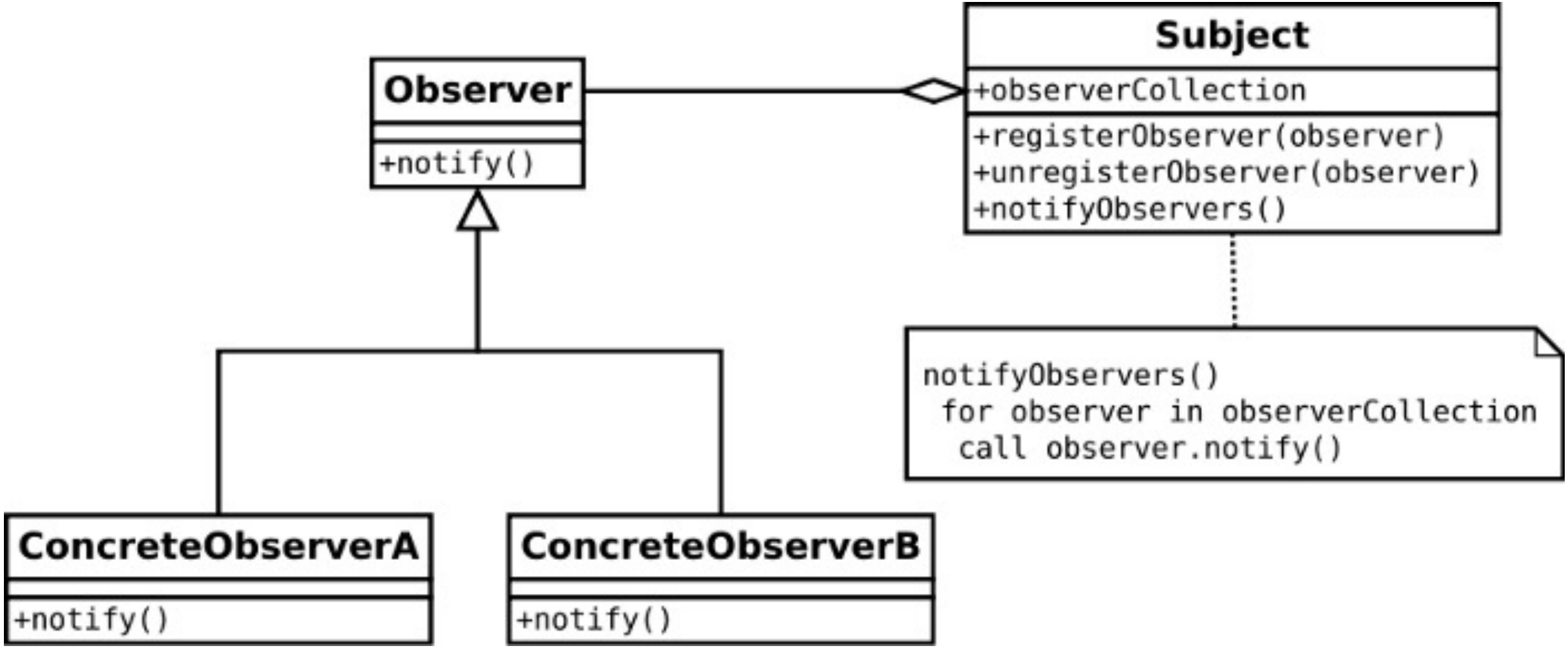


Observer - Applicability

- 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.
- Common related/special case use: MVC (Model-View-Controller)


Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
State Change
Change Notification
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events


@Observable
@EventListener
@Observer
Model
Controller
View
notifyAll()
update()



https://medium.com/@patrickackerman/the-observer-pattern-with-vanilla-javascript-8f85ea05eaa8

java.util

Interface Observer

`public interface Observer`

A class can implement the `Observer` interface when it wants to be informed of changes in observable objects.

Since:

JDK1.0

See Also:

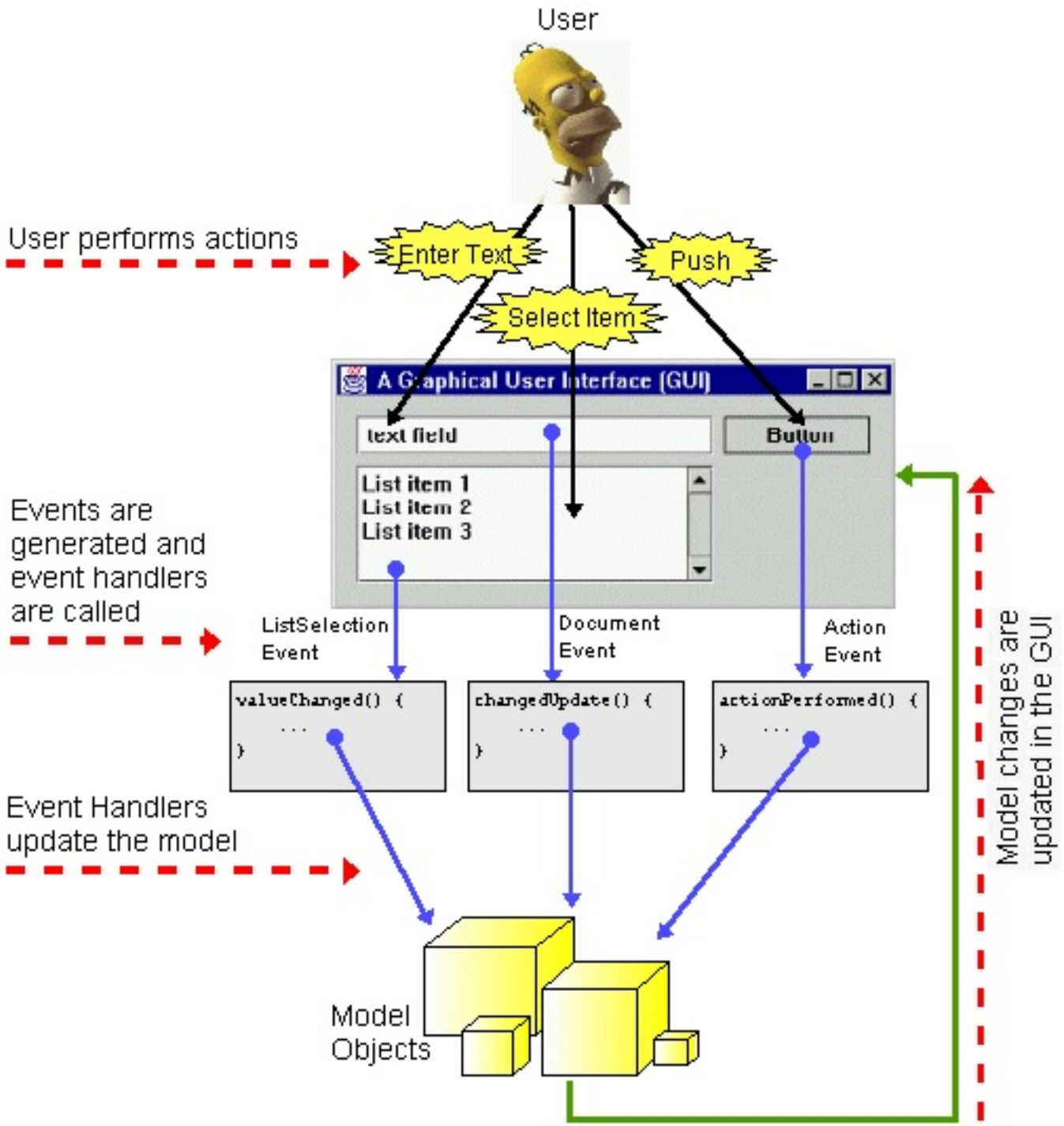
`Observable`

Method Summary

Methods

Modifier and Type	Method and Description
`void`	`update(Observable o, Object arg)` This method is called whenever the observed object is changed.

Observer Pattern – Example (GUI)



Keyboard and Mouse Events

Mouse Events:

clickAndHold()
contextClick()
doubleClick()
dragAndDrop()
dragAndDropBy()
moveByOffset()
moveToElement()
release()

Keyboard Events:

- keyDown()
- keyUp()
- sendKeys()

Real world Application

- **Splitwise group** : Anyone adds or updates any entry in the group - all members of group get a notification
- **Following a post/event**: If one follows a post , (s)he gets added to the observers & any further comments on the same post , send a notification to all the other observers
- **Software Repository**: Under the push notification model , devices are observable for the central software repository & as soon as there is new software from one of the observers , all the devices registered will be sent a push notification to check for that software
- **Weather update**
- **Stock prices update**
- **Train ticket confirmation**

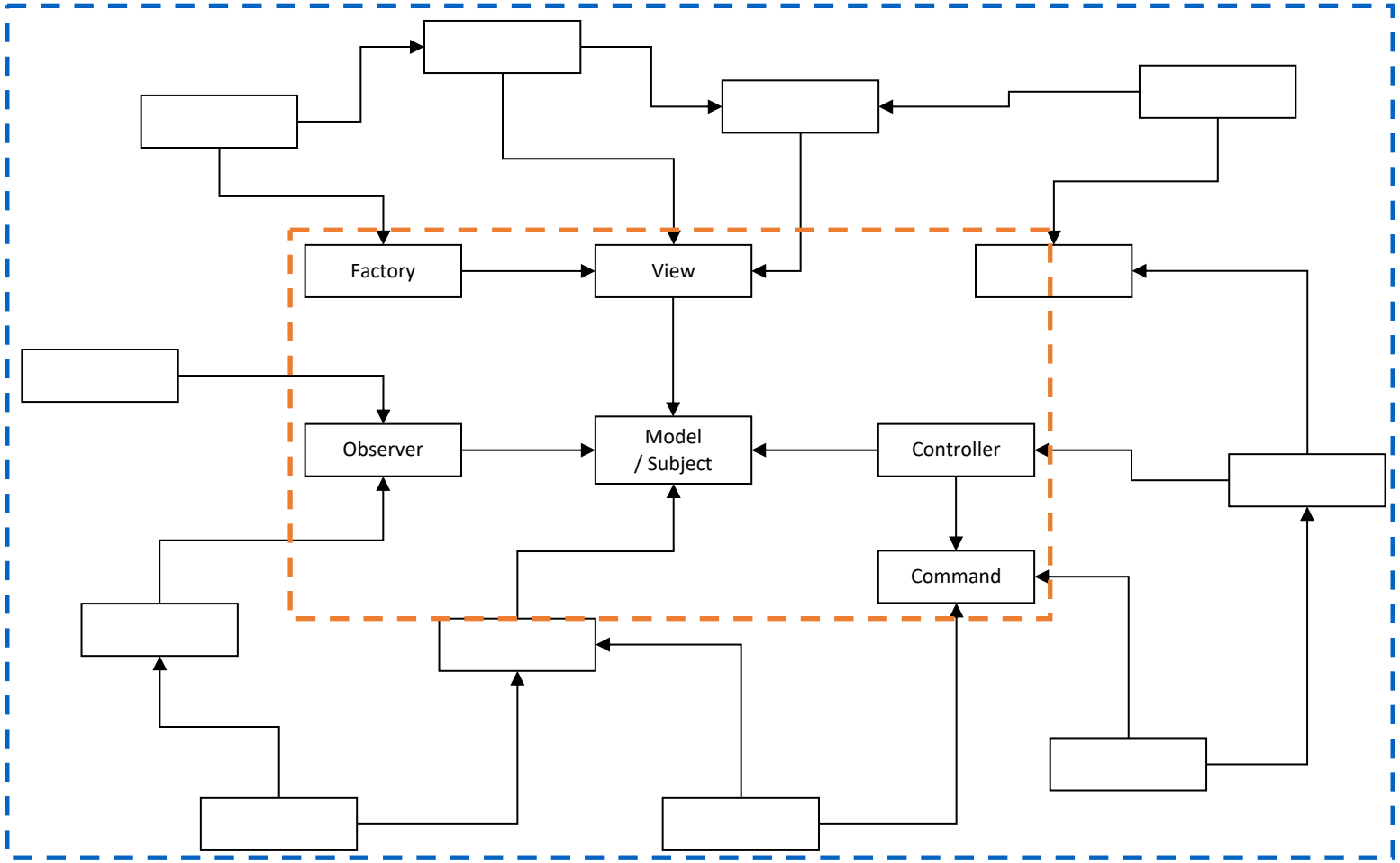
Observer - Pros and Cons

- ✓ *Open/Closed Principle*. You can introduce new subscriber classes without having to change the publisher's code (and vice versa if there's a publisher interface).
- ✓ You can establish relations between objects at runtime.
- ✗ Subscribers are notified in random order.

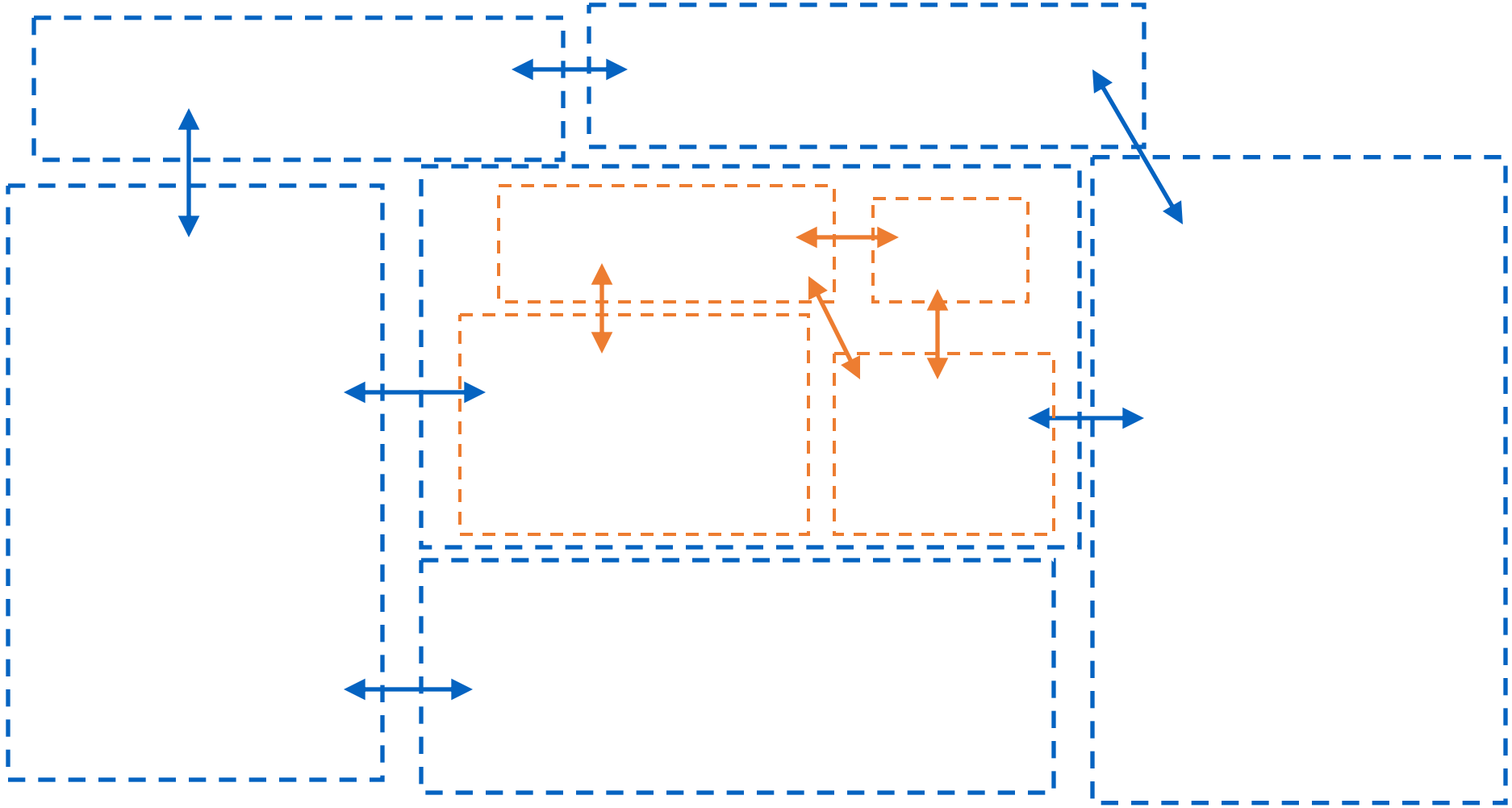
Classification of patterns

- **Creational patterns**
  - **Singleton**
  - **Factory Method**
- **Structural patterns**
  - **Composite**
- **Behavioral patterns**
  - **Strategy**
  - **Observer**

Design Patterns



Architecture



MVC Architecture

CONTROLLER

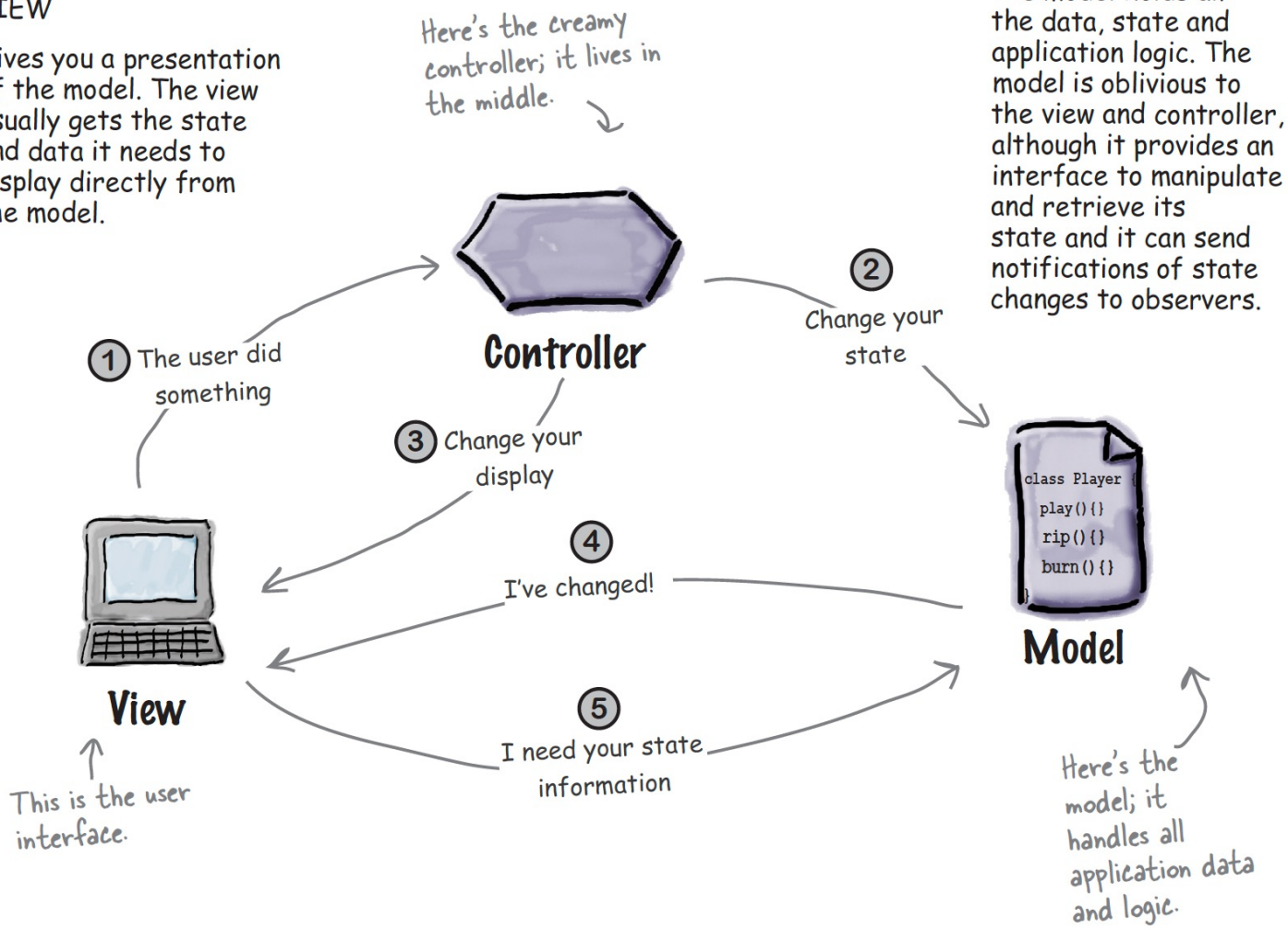
Takes user input and figures out what it means to the model.

MODEL

The model holds all the data, state and application logic. The model is oblivious to the view and controller, although it provides an interface to manipulate and retrieve its state and it can send notifications of state changes to observers.

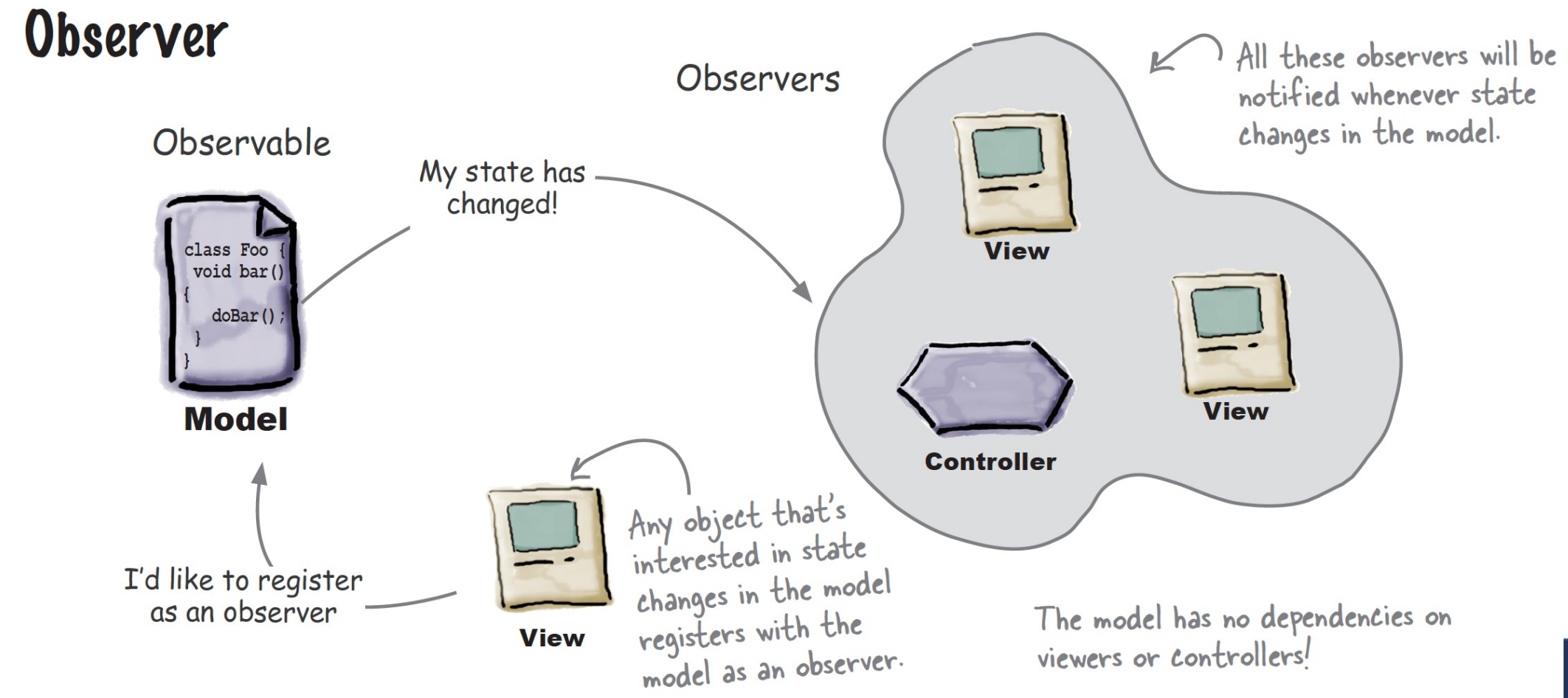
VIEW

Gives you a presentation of the model. The view usually gets the state and data it needs to display directly from the model.



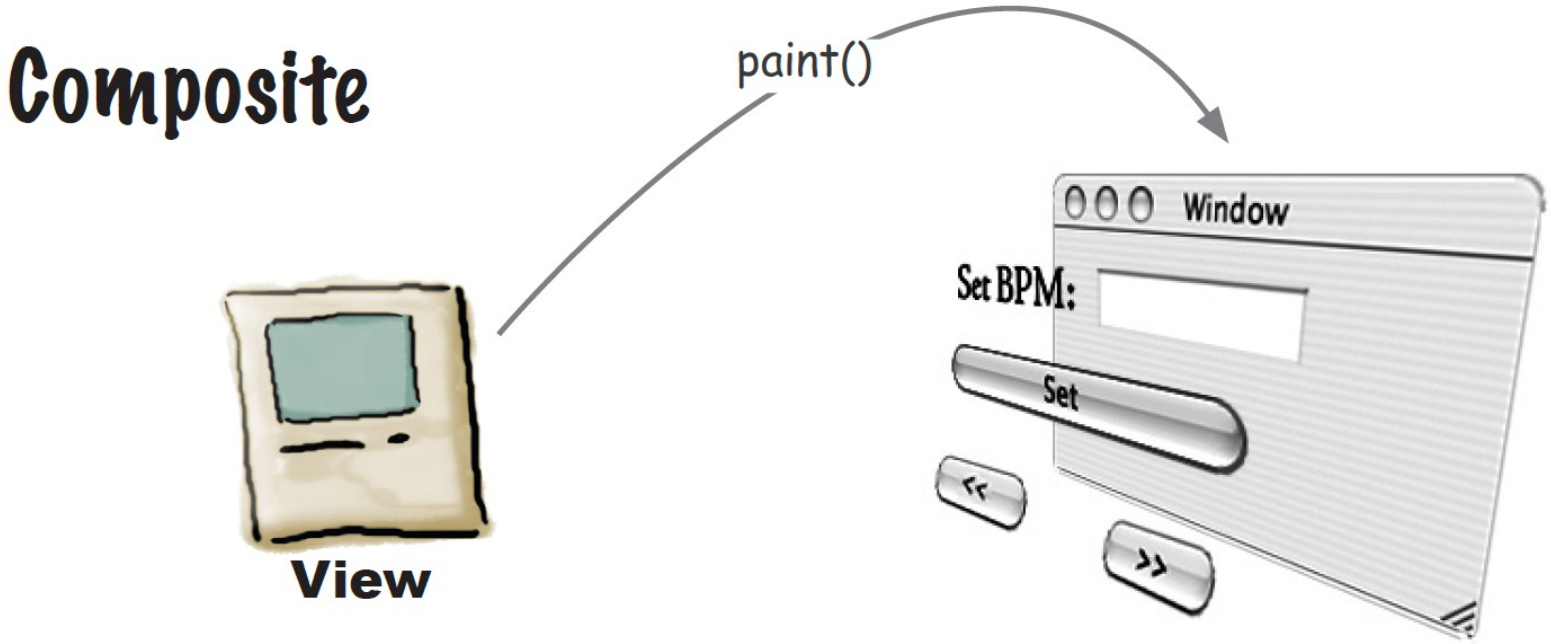
MVC Architecture

- Model – Observer Pattern



MVC Architecture

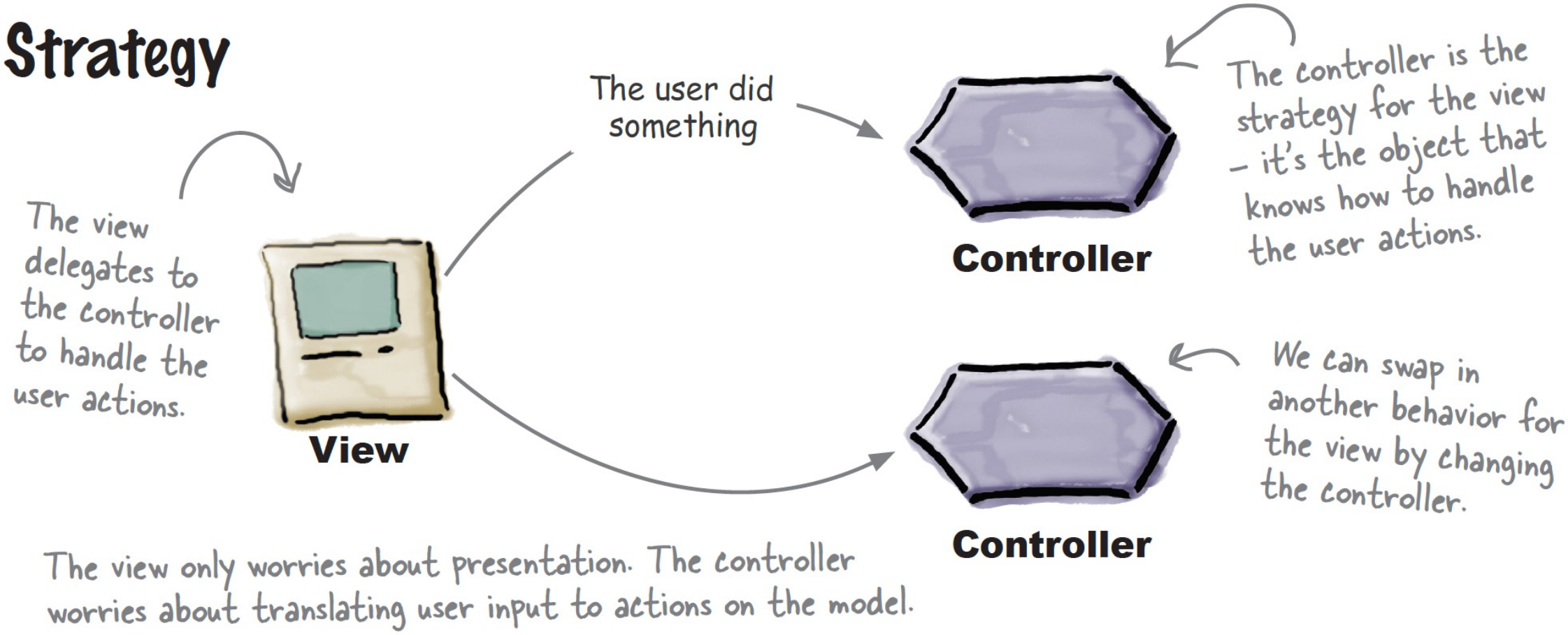
- View – Composite



The view is a composite of GUI components (labels, buttons, text entry, etc.). The top-level component contains other components, which contain other components, and so on until you get to the leaf nodes.

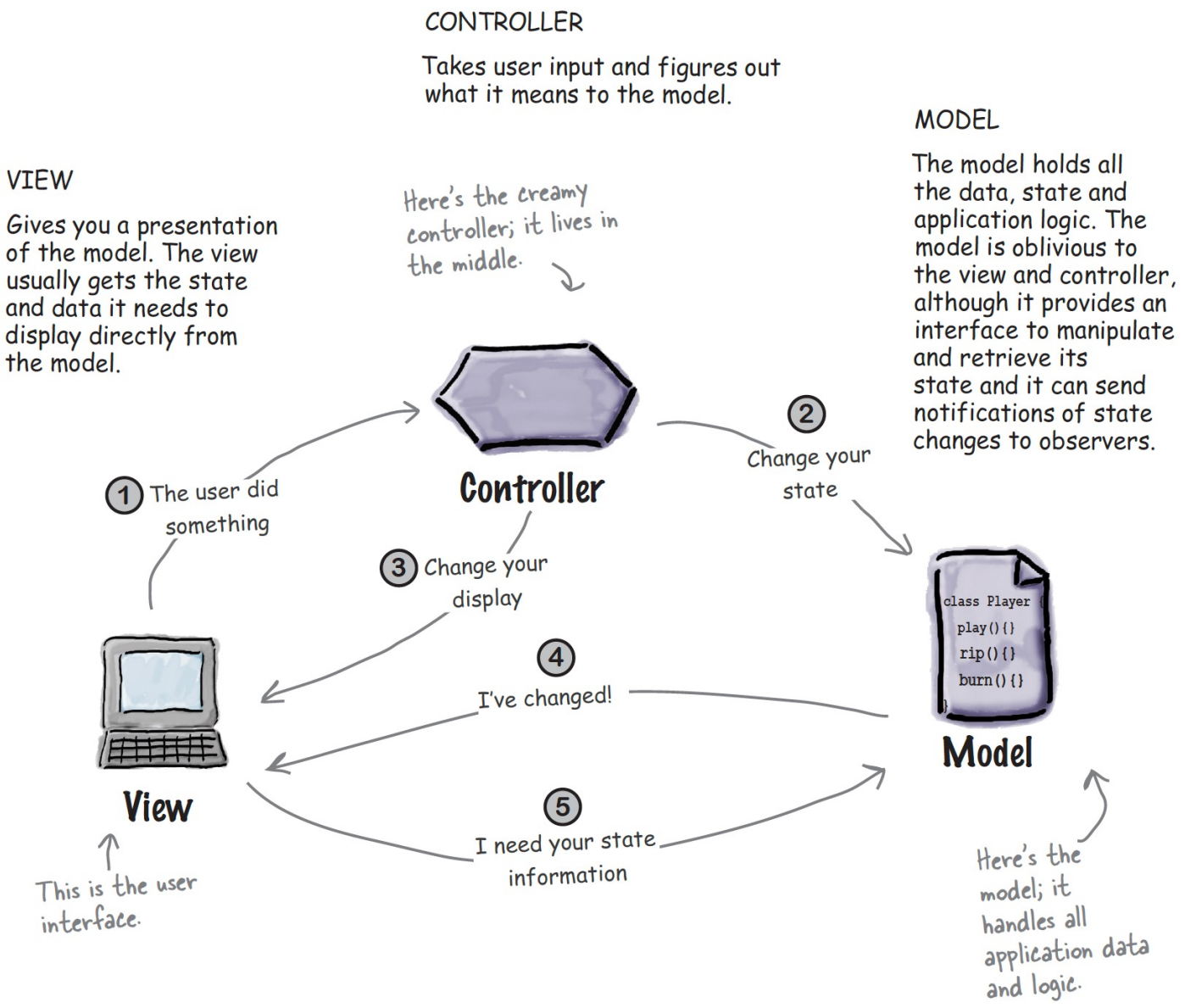
MVC Architecture

- View + Controller – Strategy Pattern



MVC Architecture

- Model – Observer Pattern
- View – Composite + Strategy
- Controller -- Strategy Pattern



Classification of patterns

- **Creational patterns**
  - **Singleton**
  - **Factory Method**
- **Structural patterns**
  - **Composite**
  - **Adapter**
- **Behavioral patterns**
  - **Strategy**
  - **Observer**

Adapter

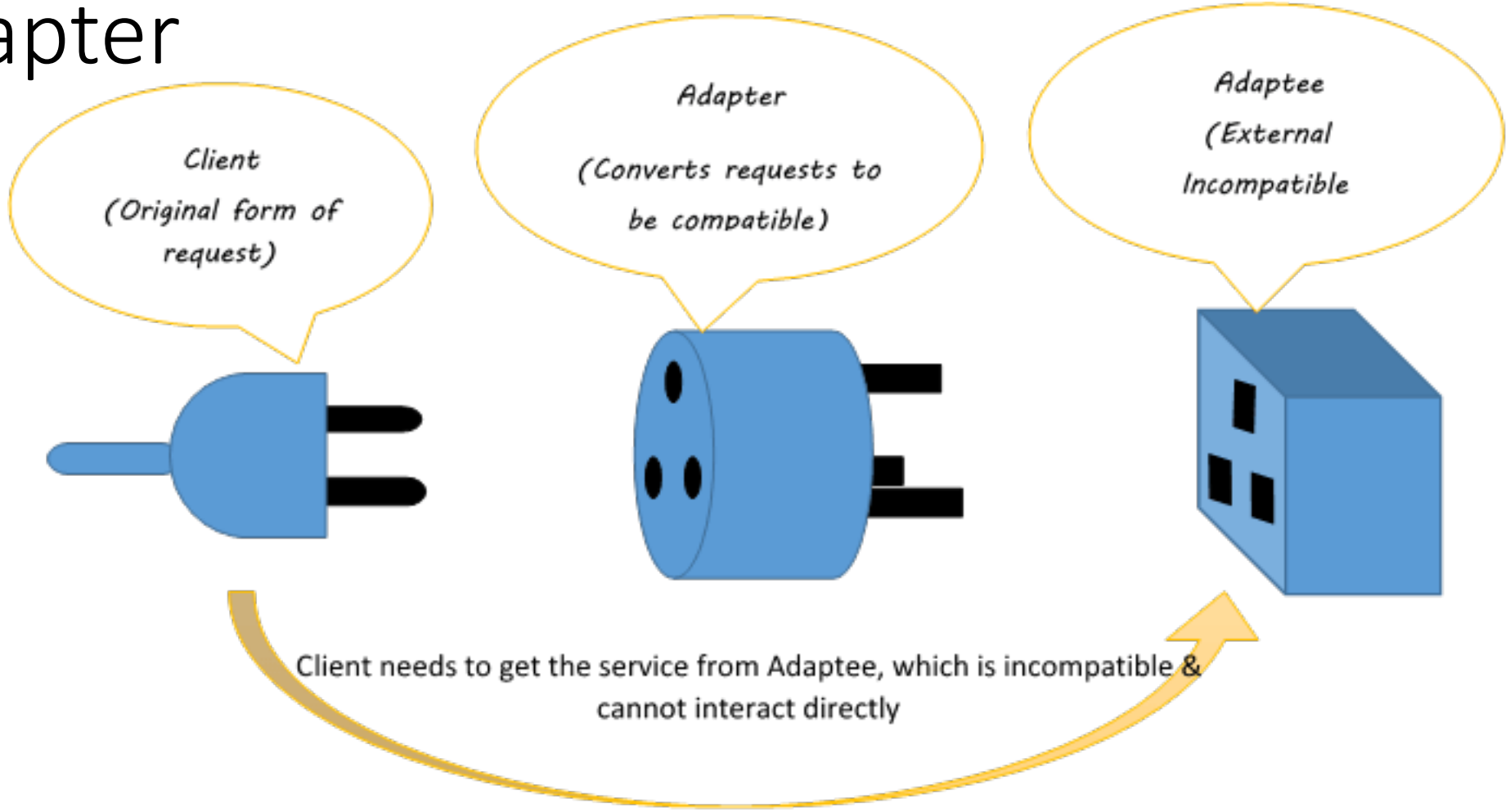
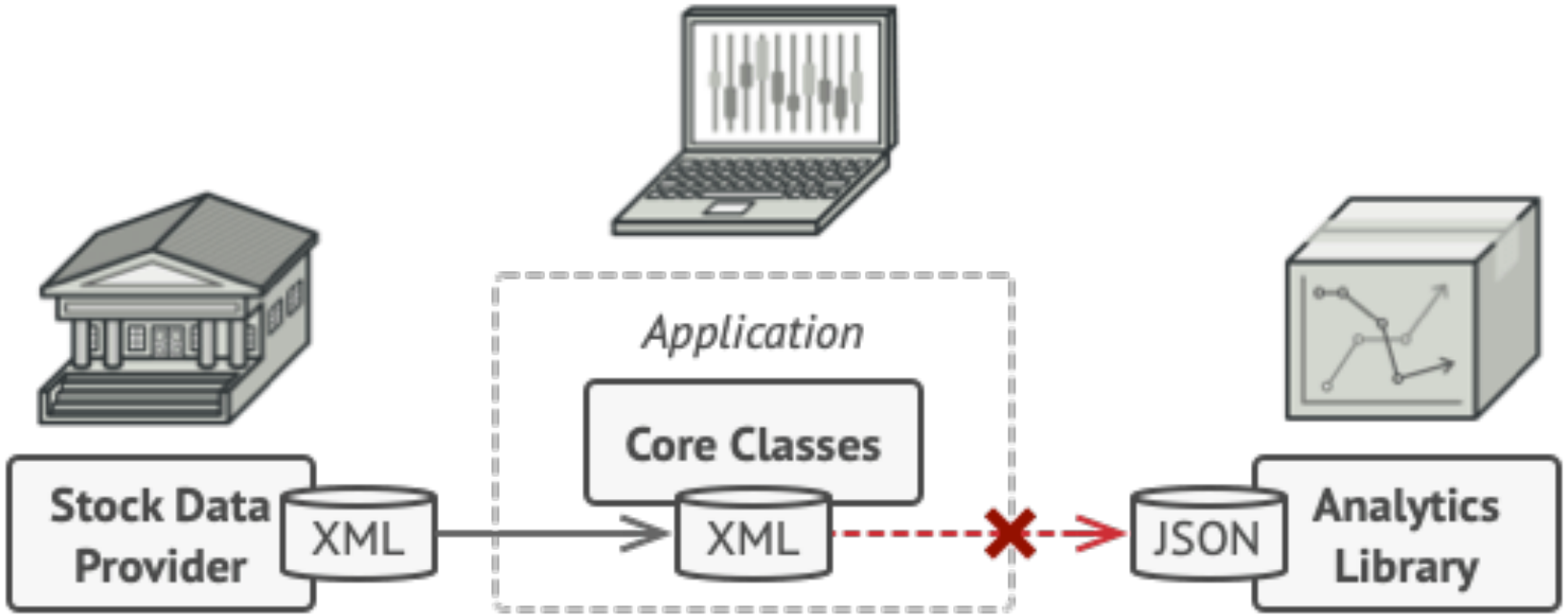


Figure 1-Adapter Pattern Concept

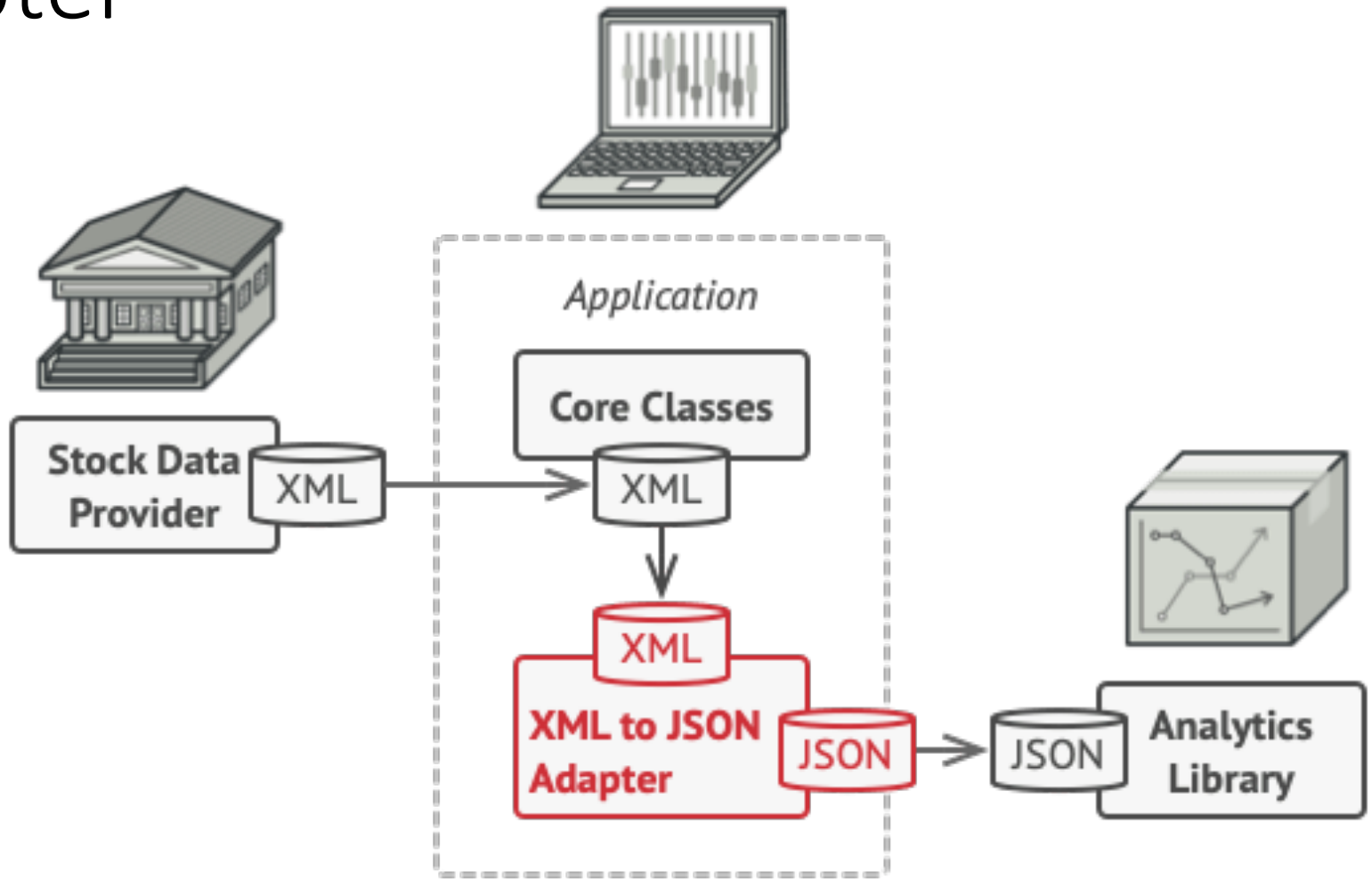
https://medium.com/@fazalcs13/adapter-design-pattern-acd51418572f

Adapter

- **Adapter** is a structural design pattern that allows objects with incompatible interfaces to collaborate.



Adapter


Application
Core Classes
Stock Data Provider
XML
XML
XML
XML to JSON Adapter
JSON
JSON
Analytics Library

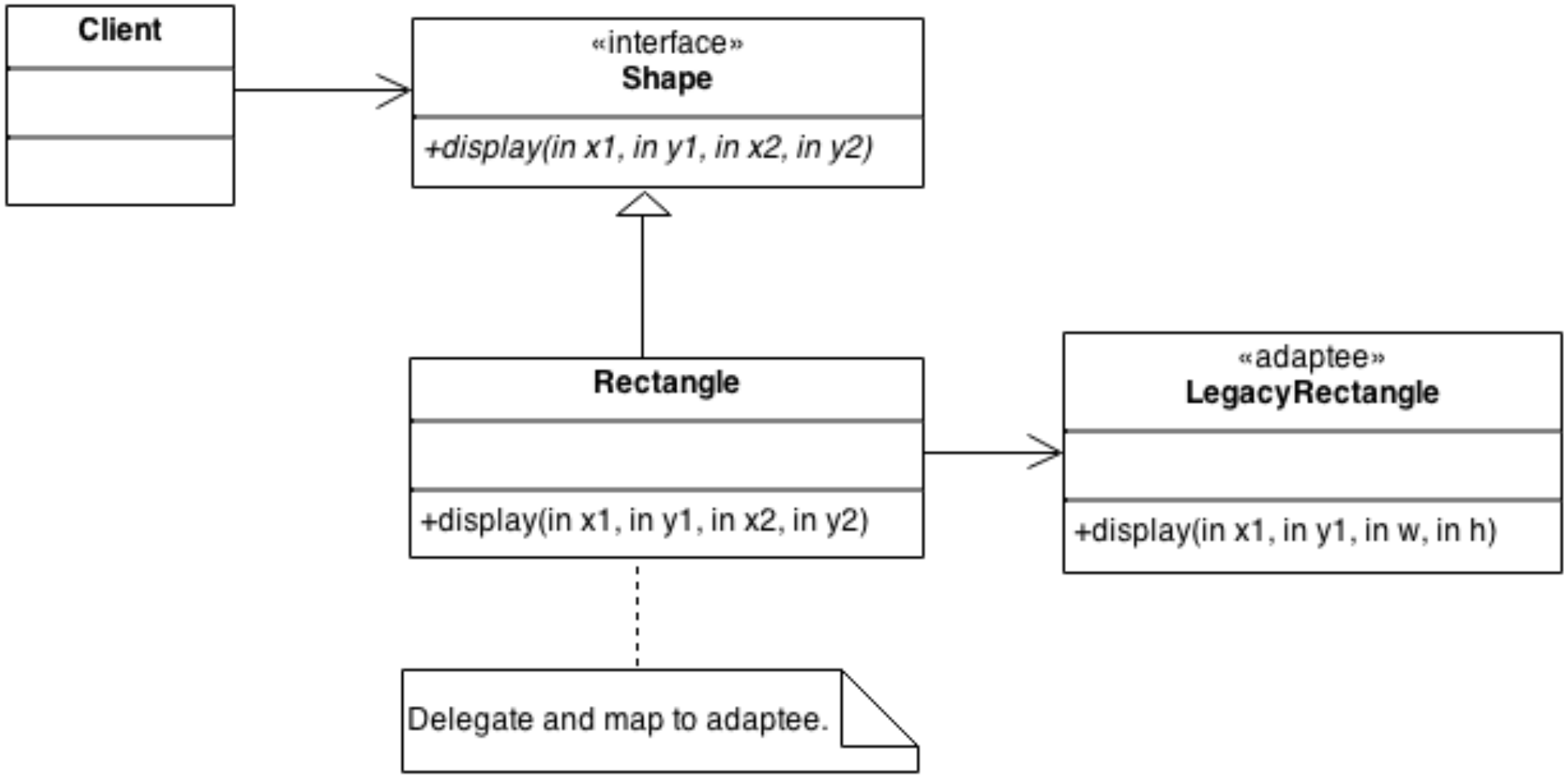
Adapter - Intent

- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Wrap an existing class with a new interface.
- Impedance match an old component to a new system

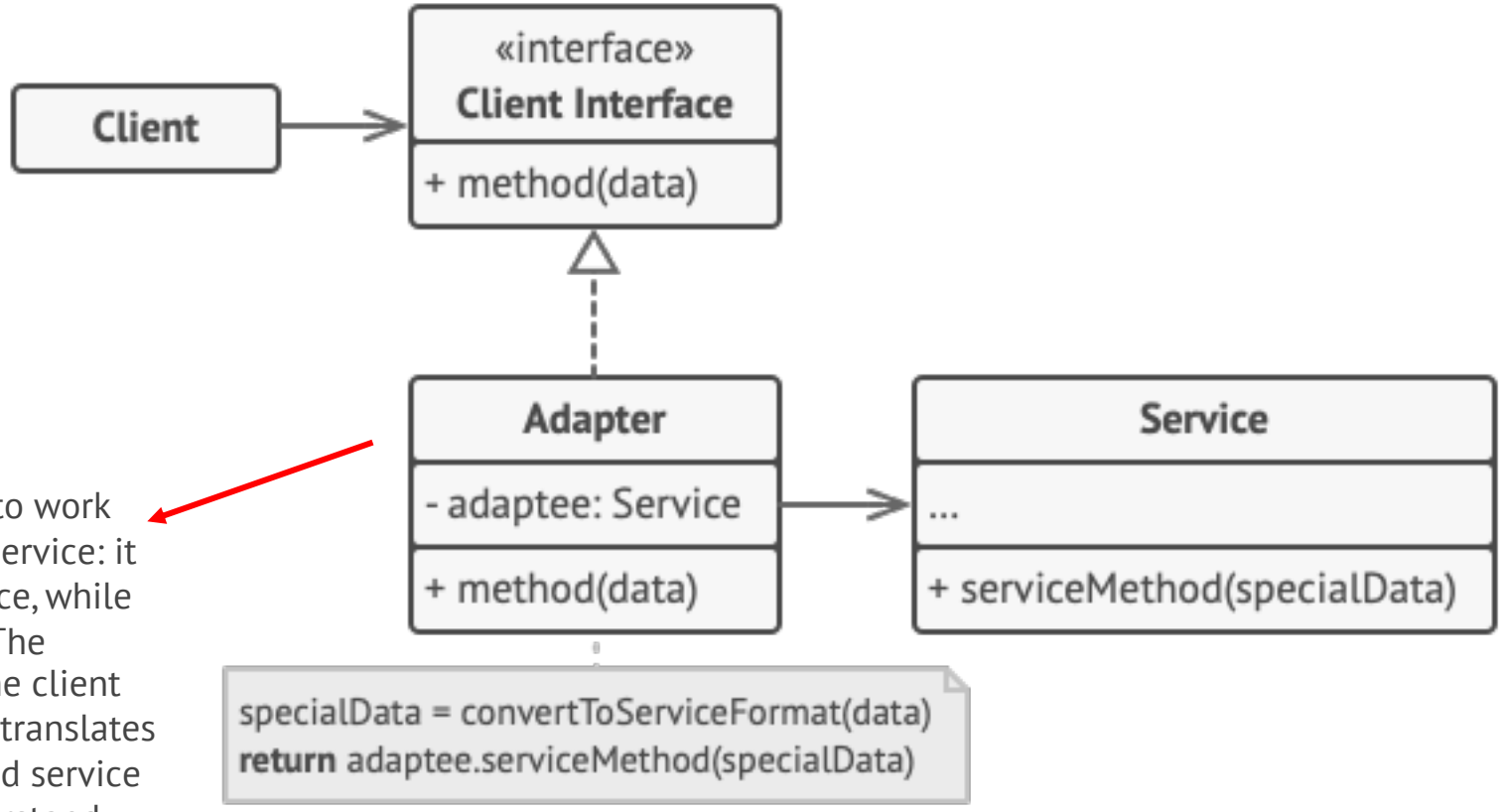
Adapter - Problem

An "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.

a legacy Rectangle component's display() method expects to receive "x, y, w, h" parameters. But the client wants to pass "upper left x and y" and "lower right x and y".



Adapter



Adapter is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the adapter interface and translates them into calls to the wrapped service object in a format it can understand.

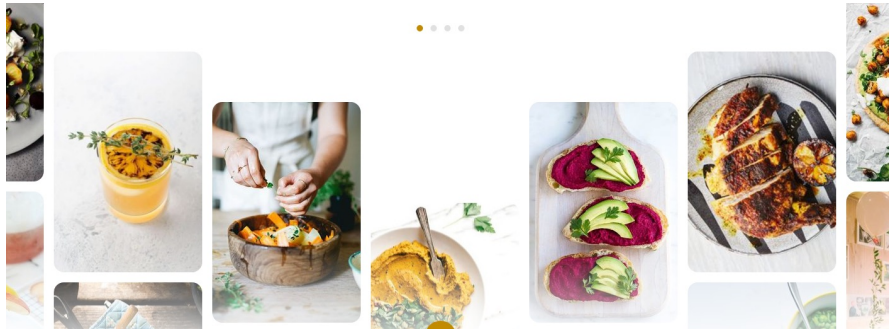
Adapter – Pros and Cons

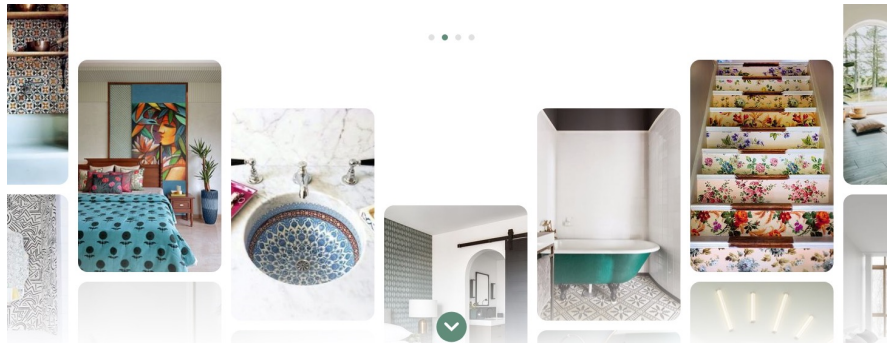
- ✓ *Single Responsibility Principle*. You can separate the interface or data conversion code from the primary business logic of the program.
- ✓ *Open/Closed Principle*. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface.
- ✗ The overall complexity of the code increases because you need to introduce a set of new interfaces and classes. Sometimes it's simpler just to change the service class so that it matches the rest of your code.

Classification of patterns

- **Creational patterns**
  - **Singleton**
  - **Factory Method**
- **Structural patterns**
  - **Composite**
  - **Adapter**
  - **Proxy**
- **Behavioral patterns**
  - **Strategy**
  - **Observer**

Proxy Pattern



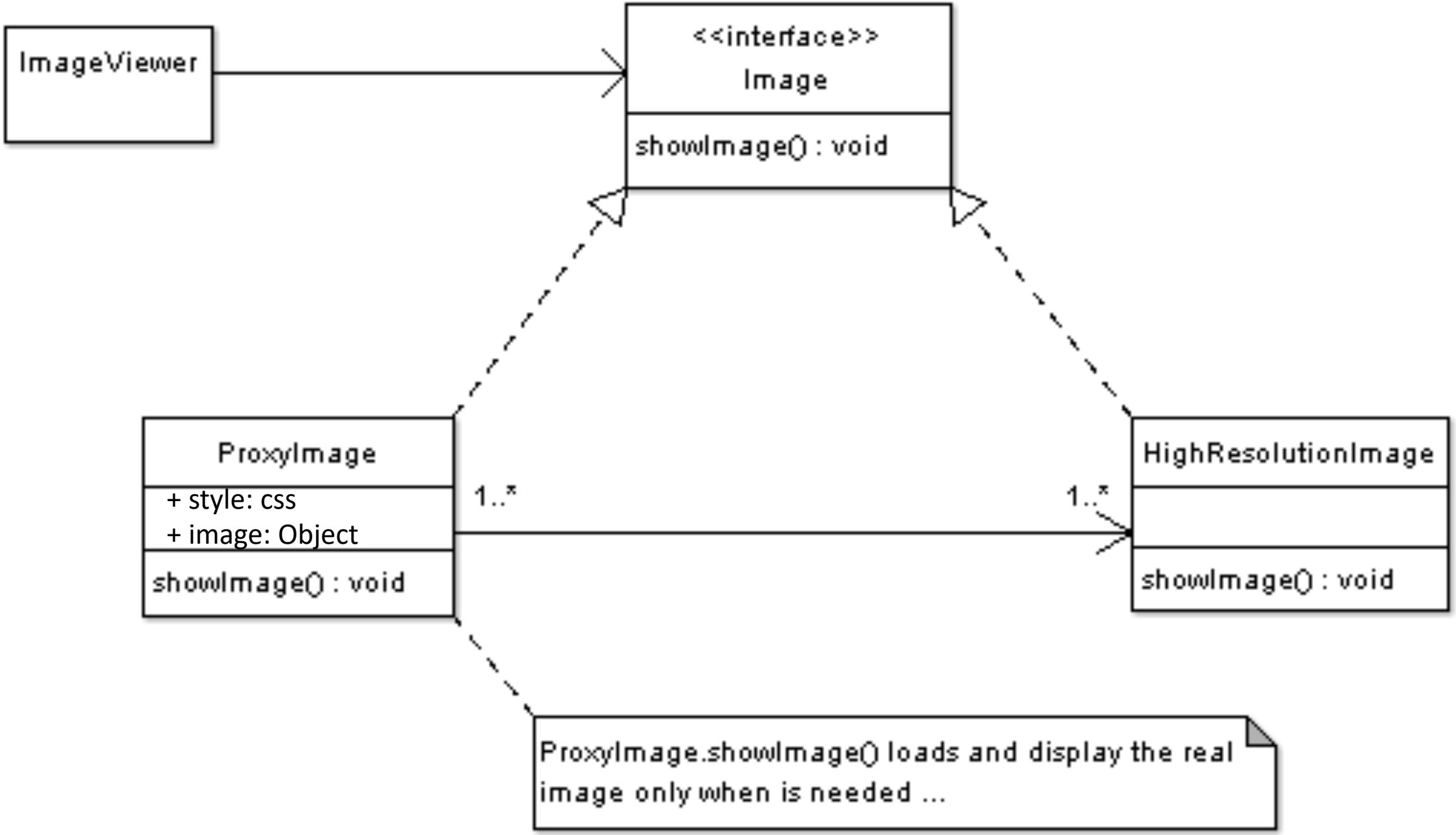


Problem:

- High-resolution images on website
- Long loading time
- Style images

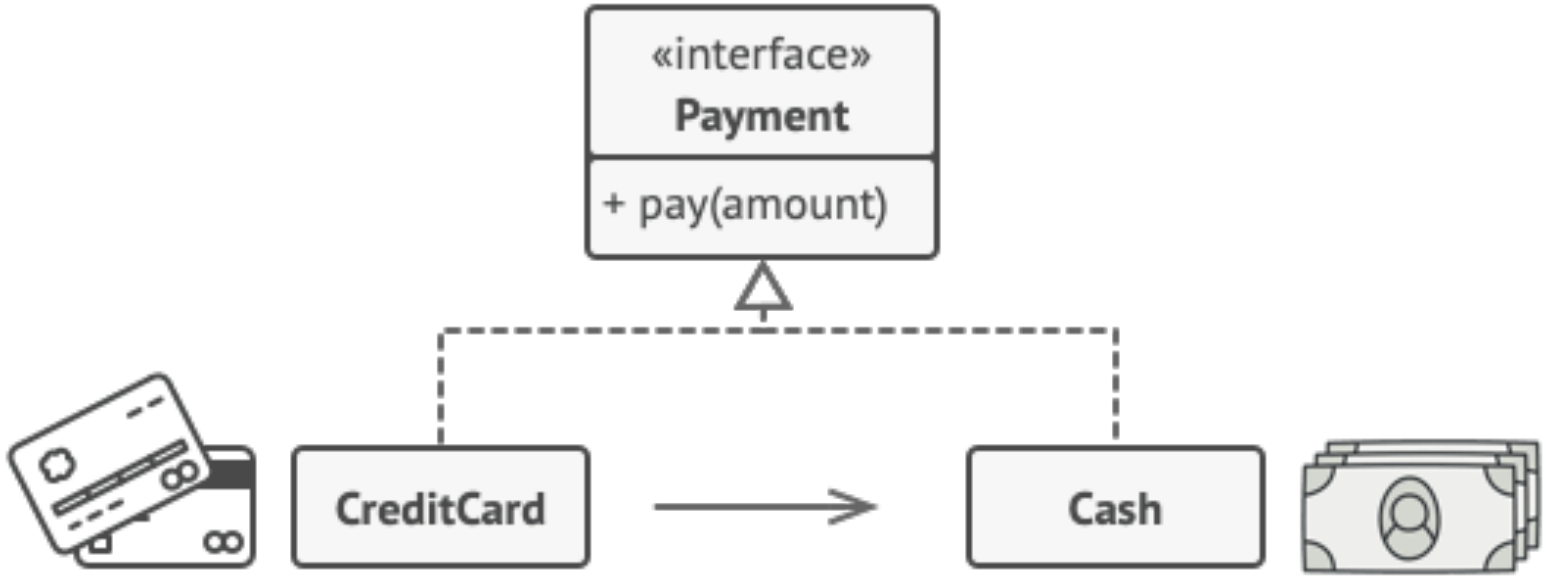
Solution:

- Replace with placeholders (proxies)
- Style placeholders


ImageViewer
<<interface>>
Image
showImage() : void
ProxyImage
+ style: css
+ image: Object
showImage() : void
1..*
1..*
HighResolutionImage
showImage() : void
ProxyImage.showImage() loads and display the real image only when is needed ...

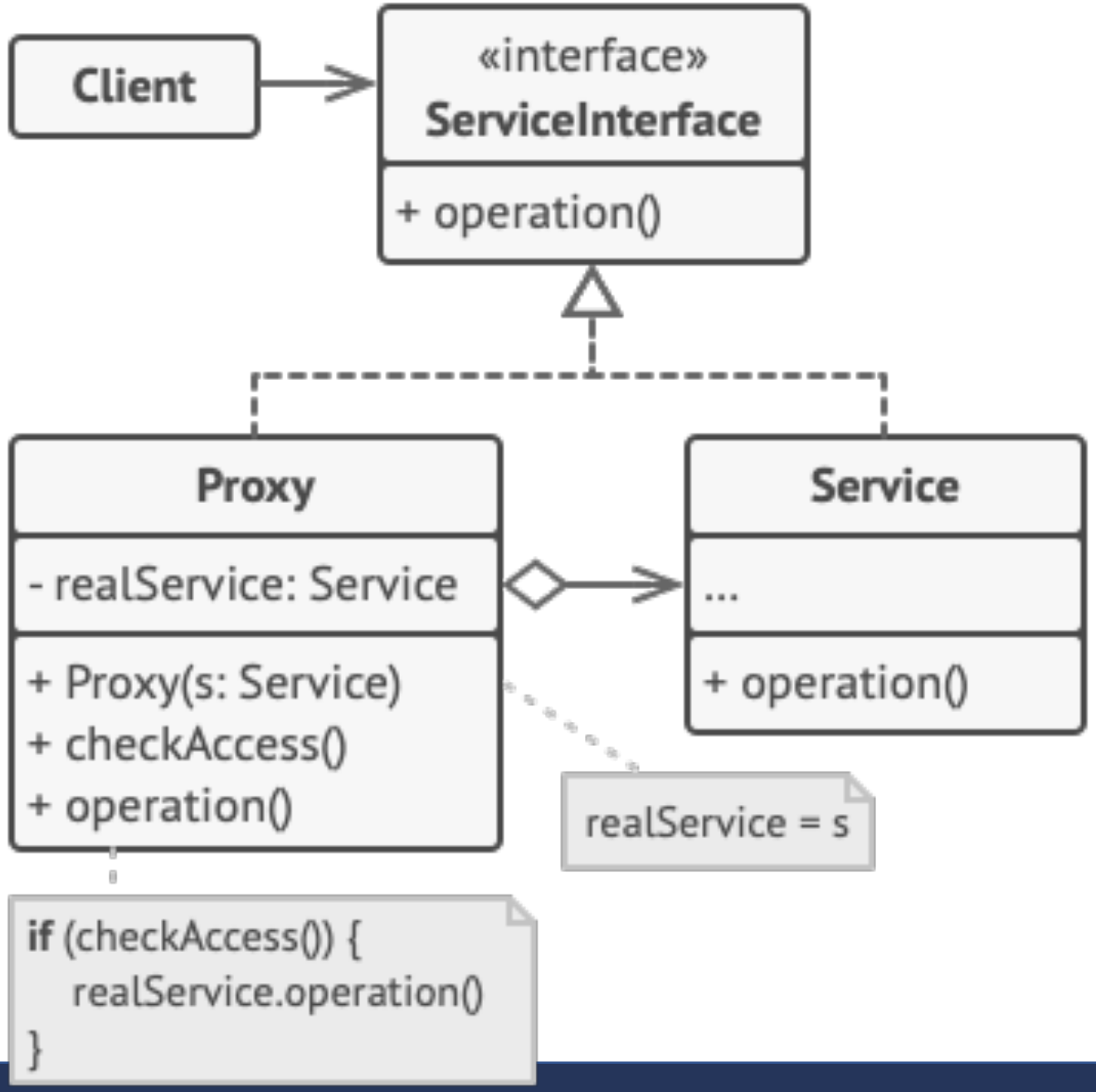
Proxy Pattern

Implement lazy initialization: create this object only when it's actually needed.



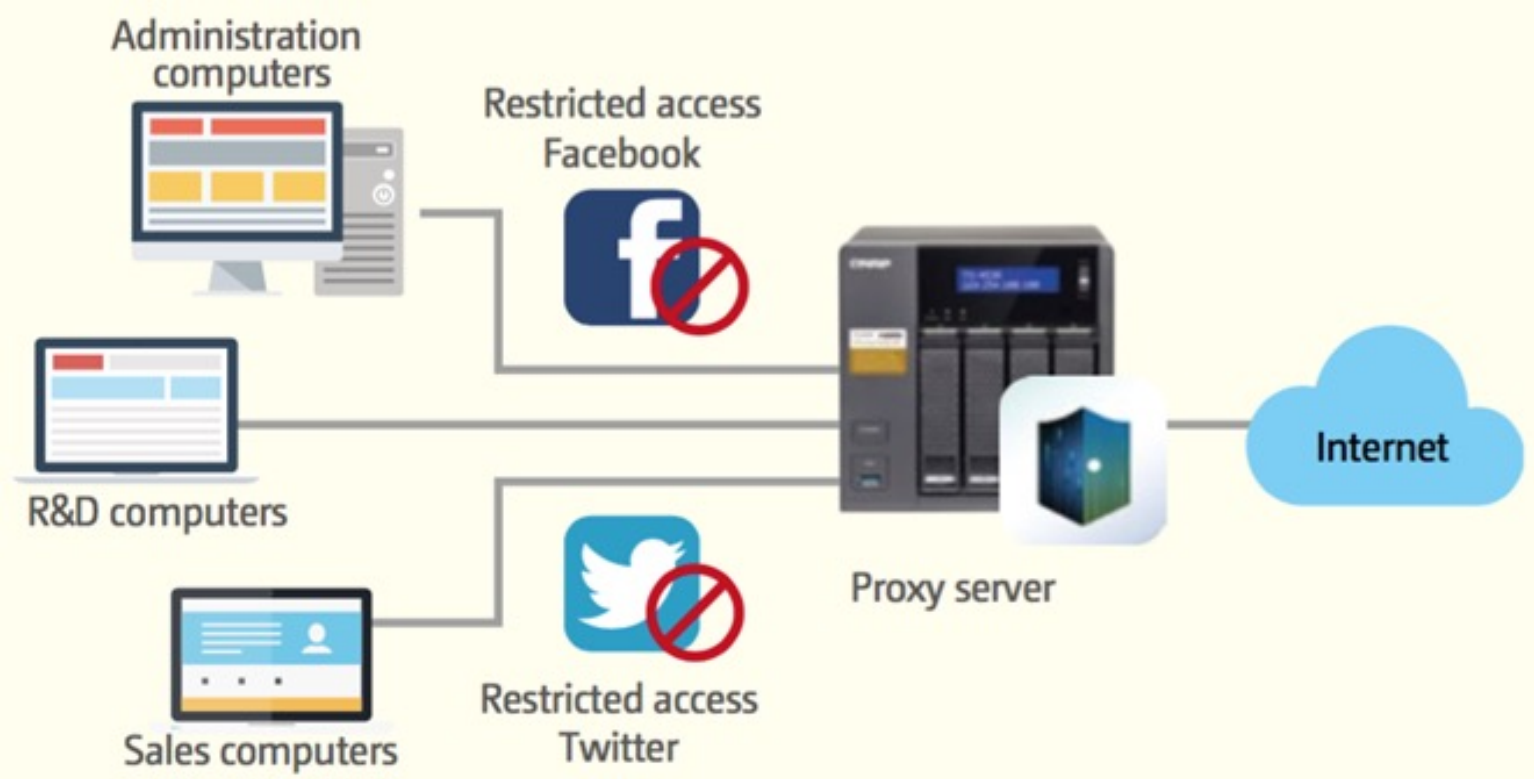
Proxy Pattern

- After the proxy finishes its processing (e.g., lazy initialization, logging, access control, caching, etc.), it passes the request to the service object.



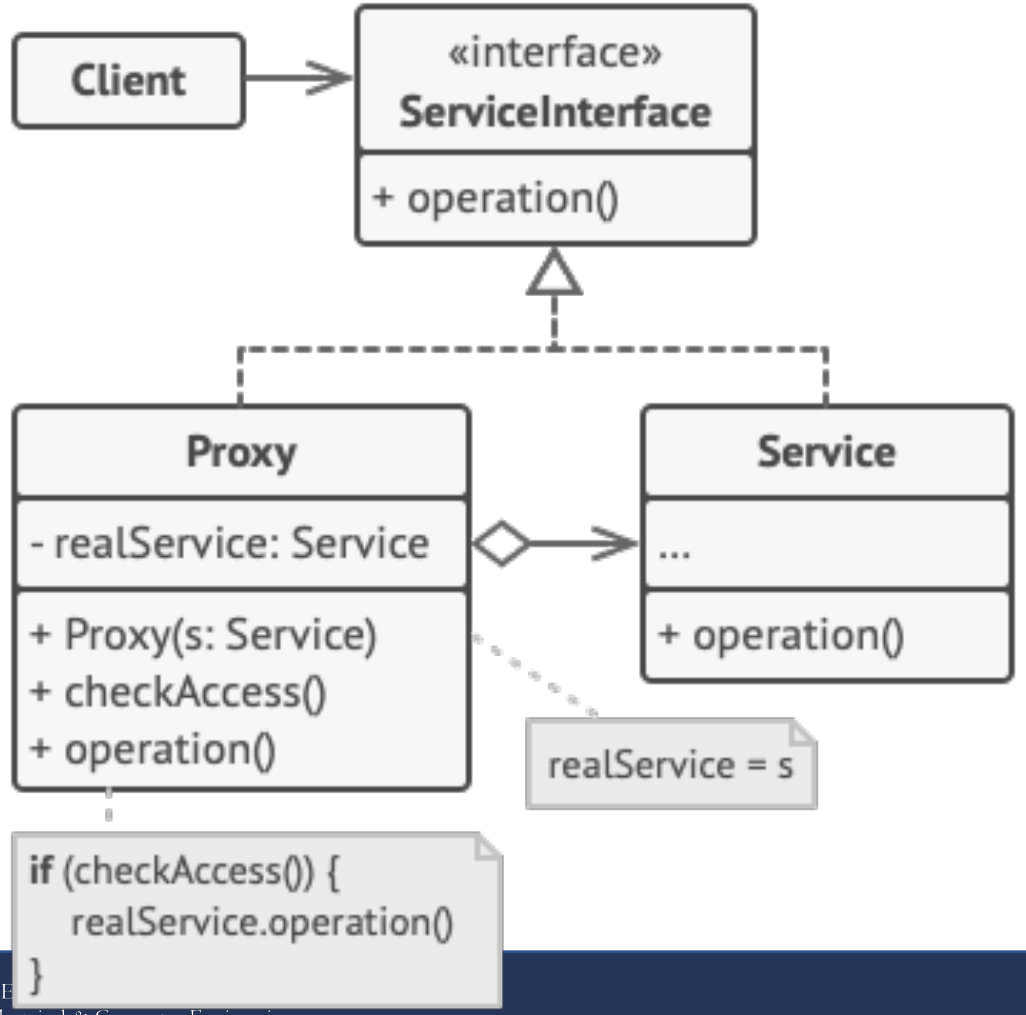
Proxy Pattern - Application

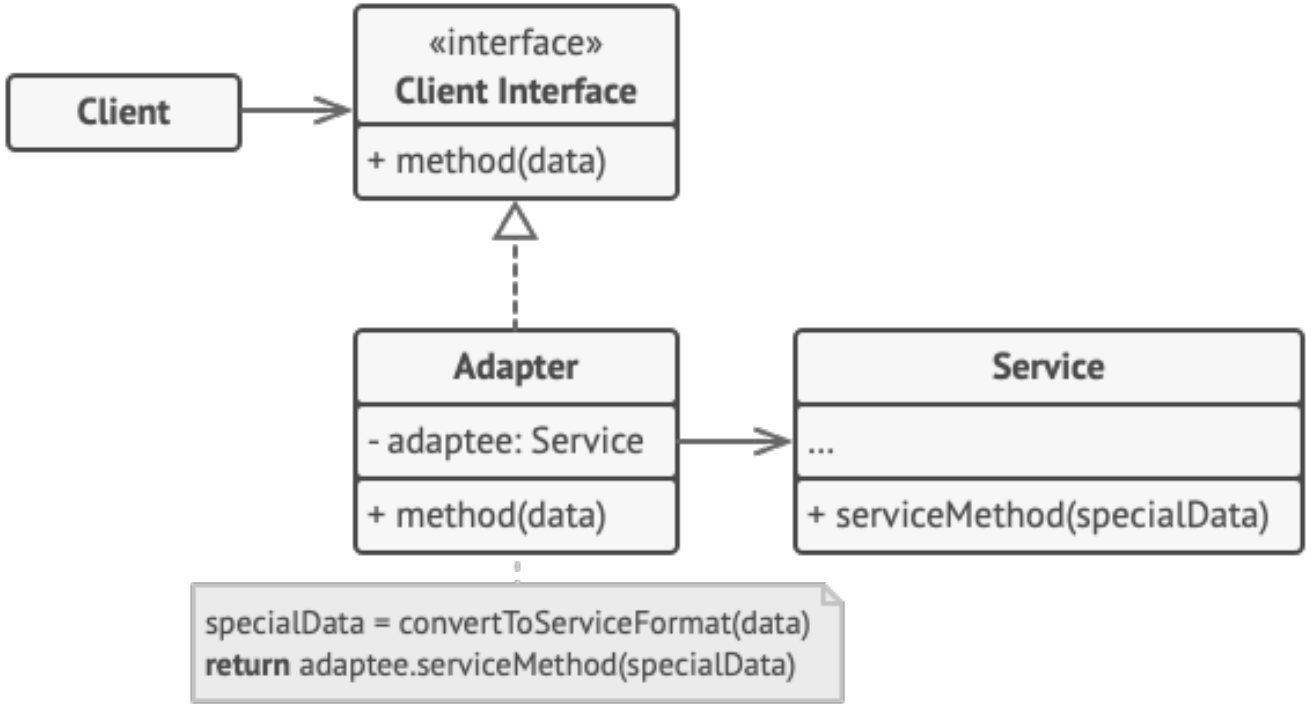
- Access control
- Logging requests



Proxy vs Adapter

- *Adapter provides a different interface to its subject. Proxy provides the same interface*
- *Adapter is meant to change the interface of an existing object*





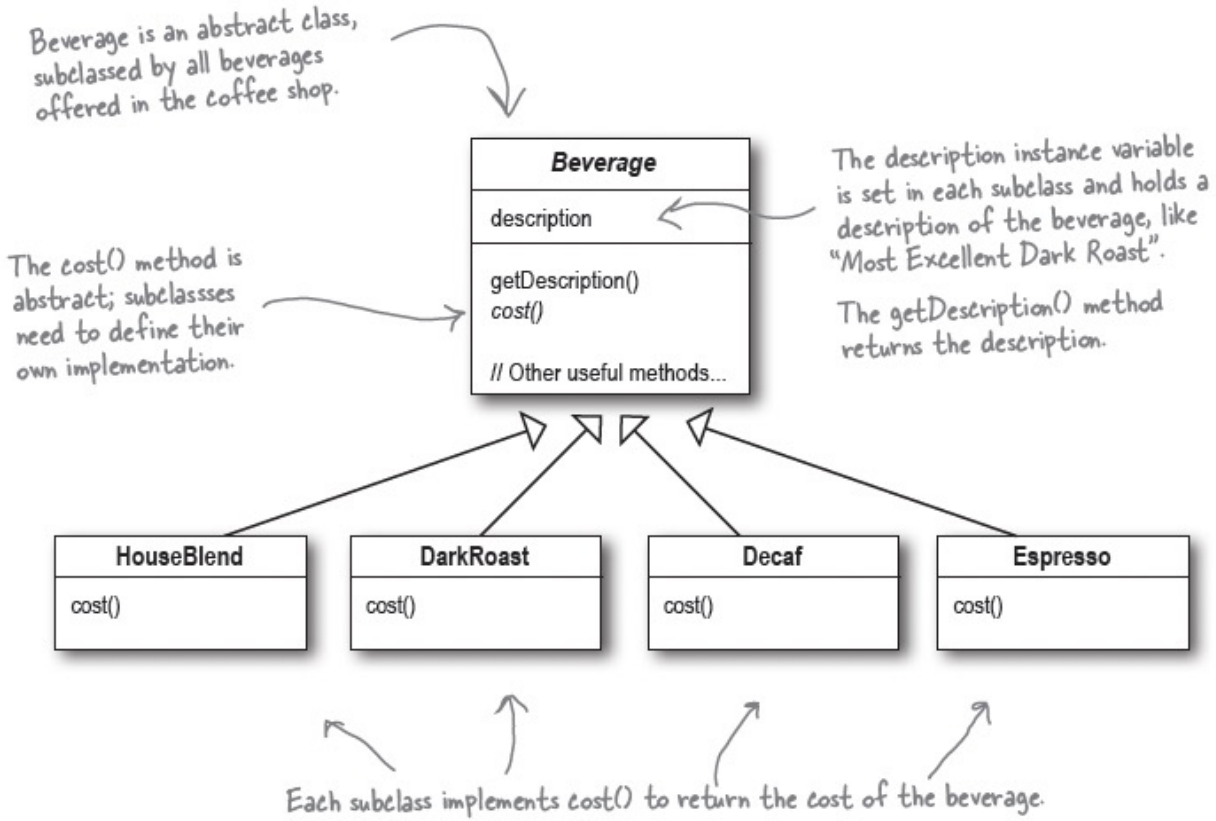
Classification of patterns

- **Creational patterns**
  - **Singleton**
  - **Factory Method**
- **Structural patterns**
  - **Composite**
  - **Adapter**
  - **Proxy**
  - **Decorator**
- **Behavioral patterns**
  - **Strategy**
  - **Observer**

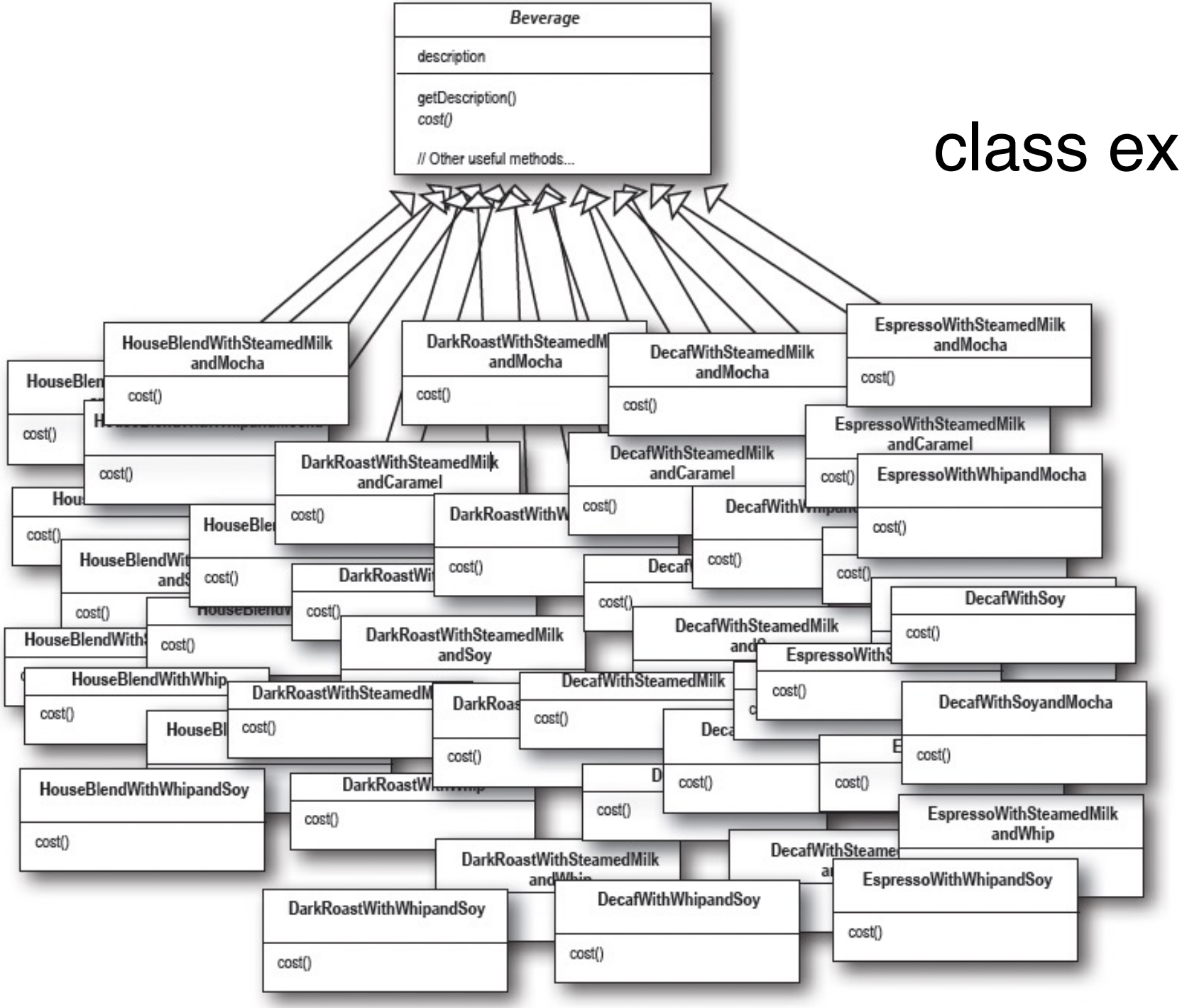
Decorator Pattern

- Starbuzz Coffee Ordering System

Scrambling to update their ordering systems to match their beverage offerings.



class explosion!



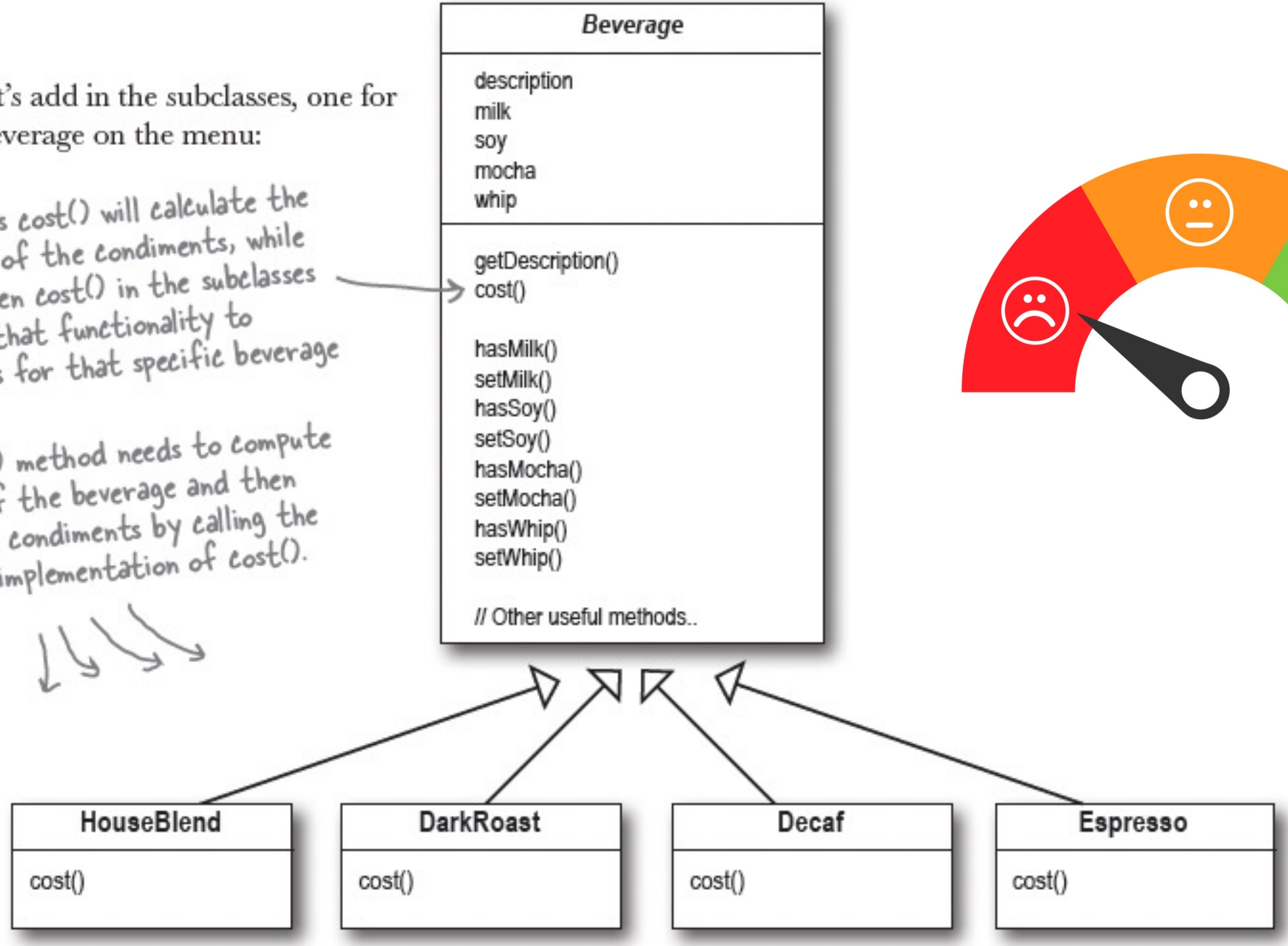
why do we need all these classes?

Can’t we just use instance variables and inheritance in the superclass to keep track of the condiments?

Now let's add in the subclasses, one for each beverage on the menu:

The superclass cost() will calculate the costs for all of the condiments, while the overridden cost() in the subclasses will extend that functionality to include costs for that specific beverage type.

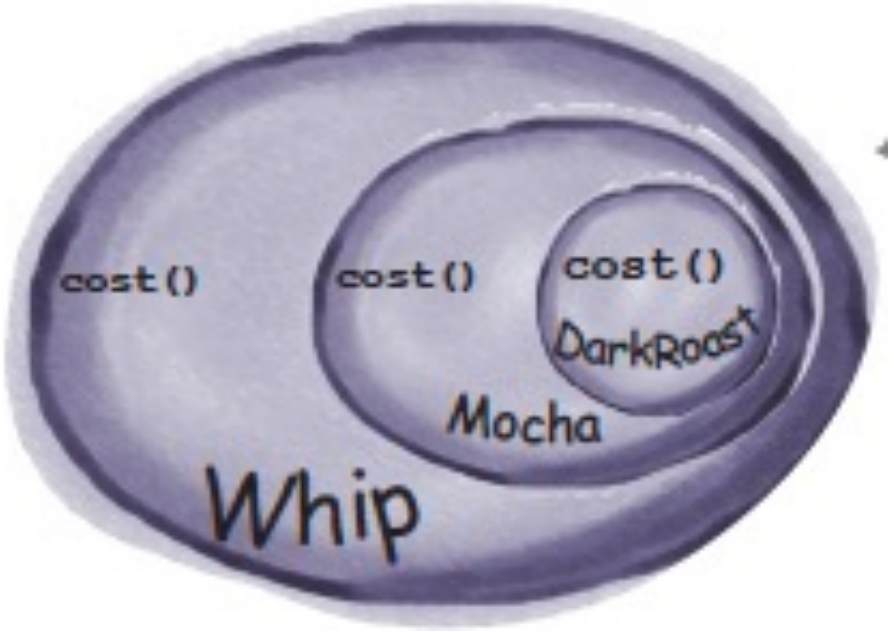
Each cost() method needs to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of cost().



- **S** — Single responsibility principle
- **O** — Open/closed principle
- **L** — Liskov substitution principle
- **I** — Interface segregation principle
- **D** — Dependency inversion principle

Decorator Pattern

- **Decorator** is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

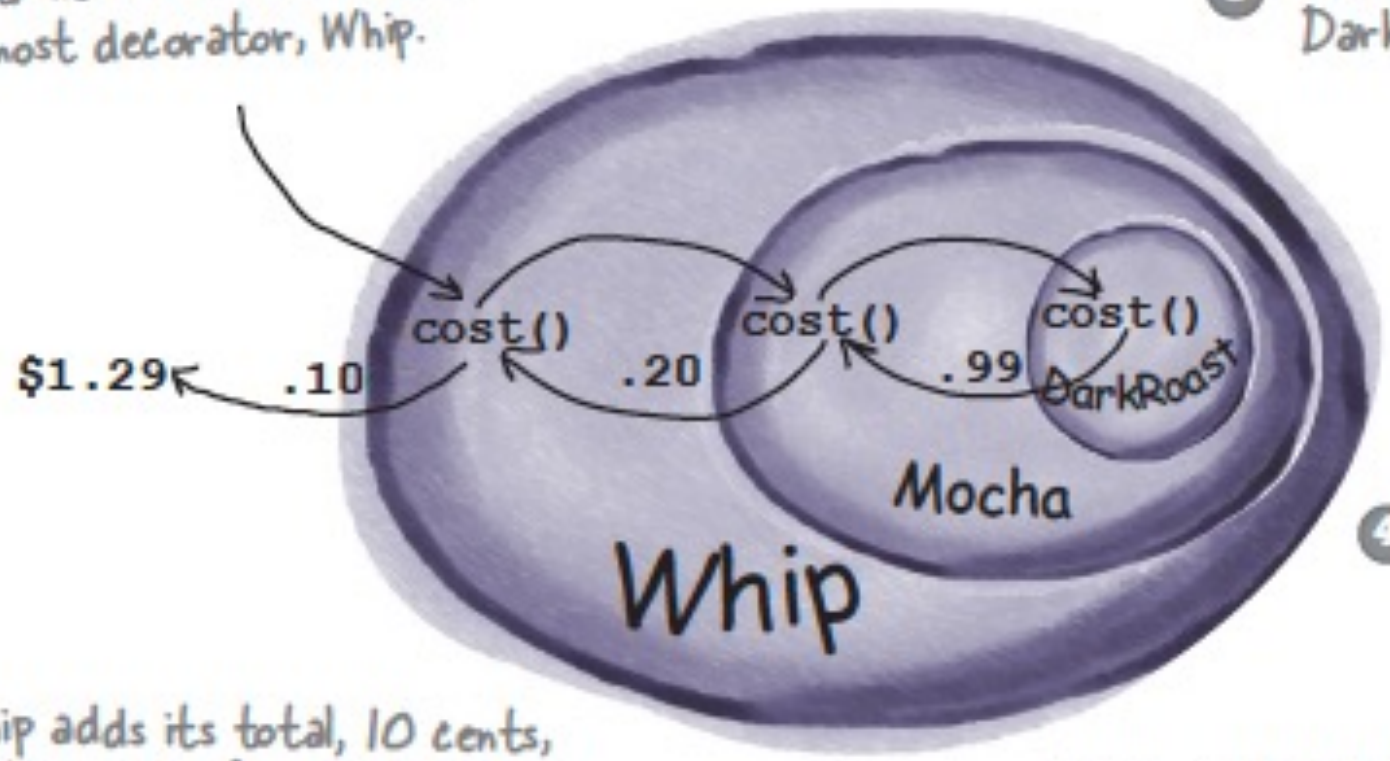


(You'll see how in a few pages.)

2 Whip calls cost() on Mocha

1 First, we call cost() on the outmost decorator, Whip.

3 Mocha calls cost() on DarkRoast

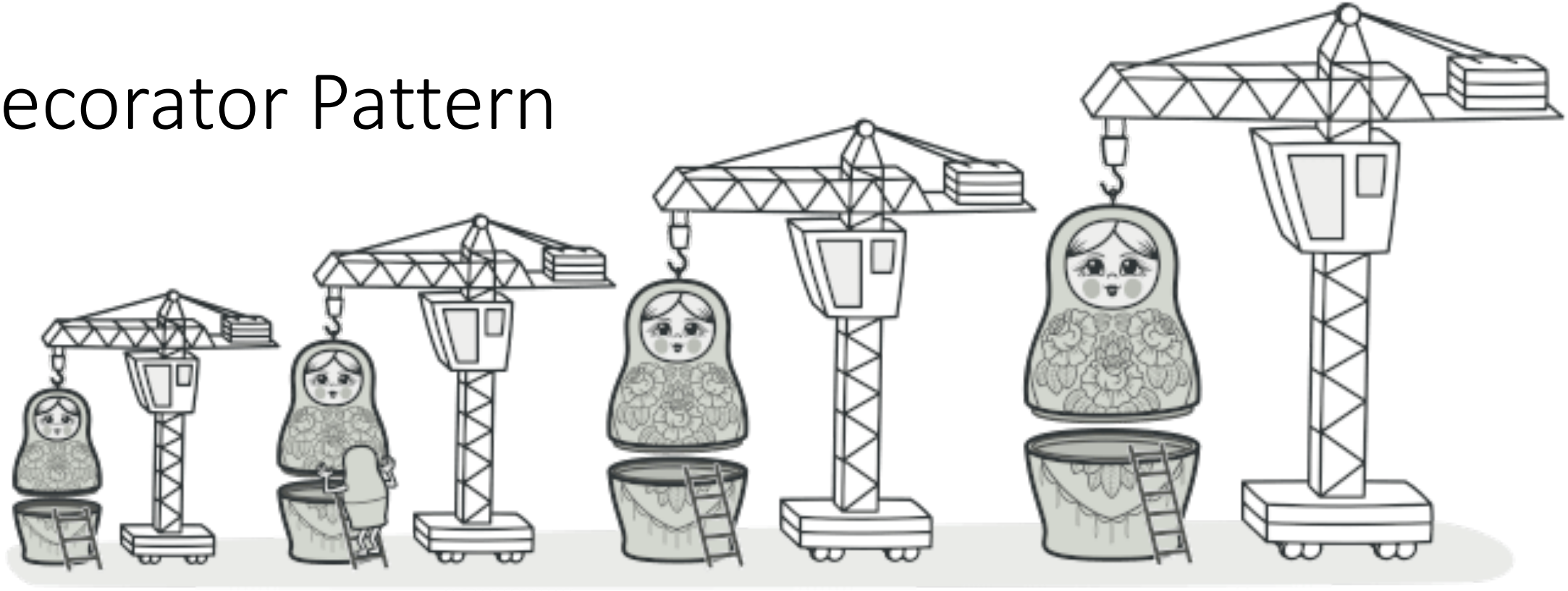


4 DarkRoast returns its cost, 99 cents.

6 Whip adds its total, 10 cents, to the result from Mocha, and returns the final result—$1.29.

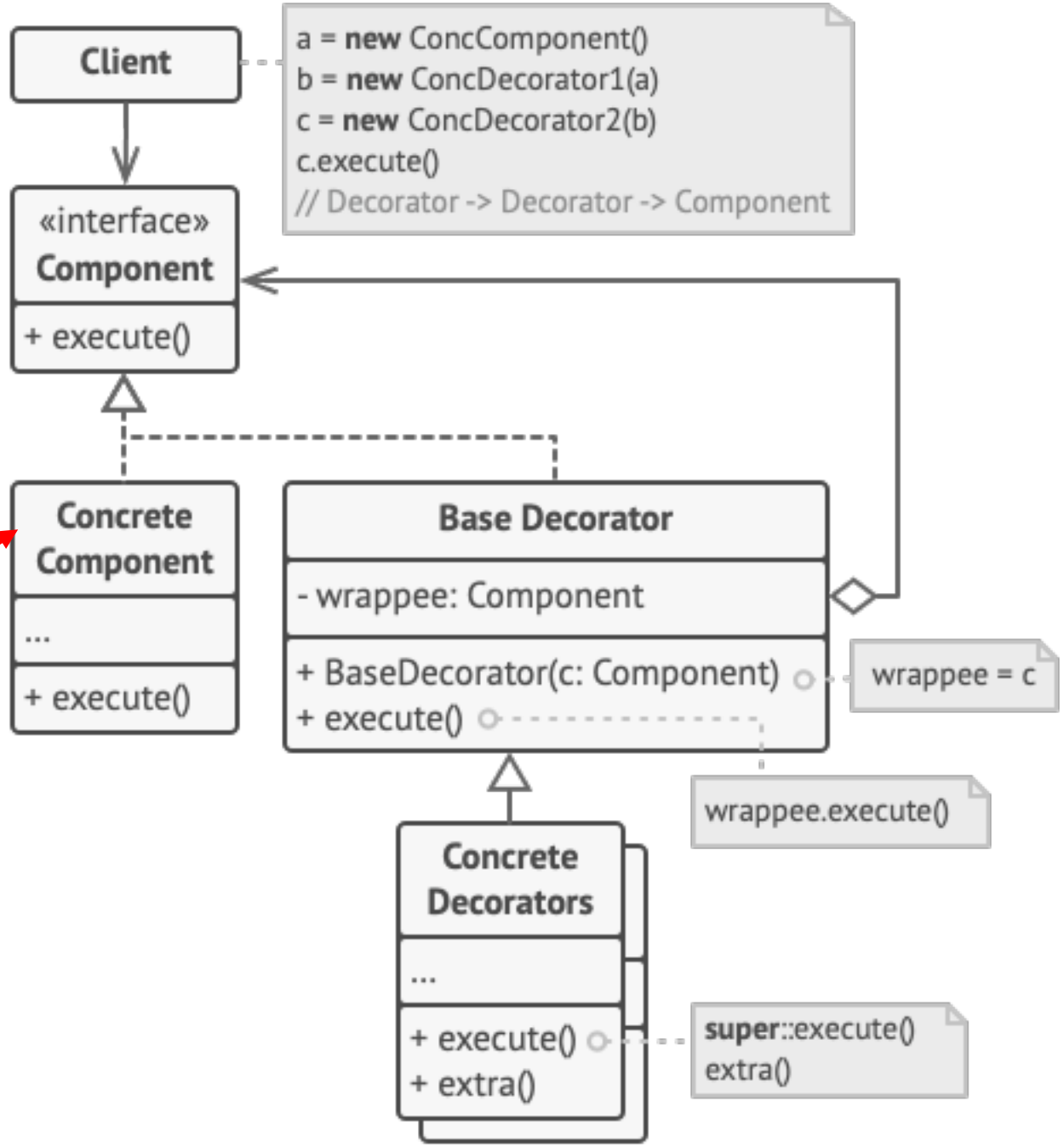
5 Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.

Decorator Pattern

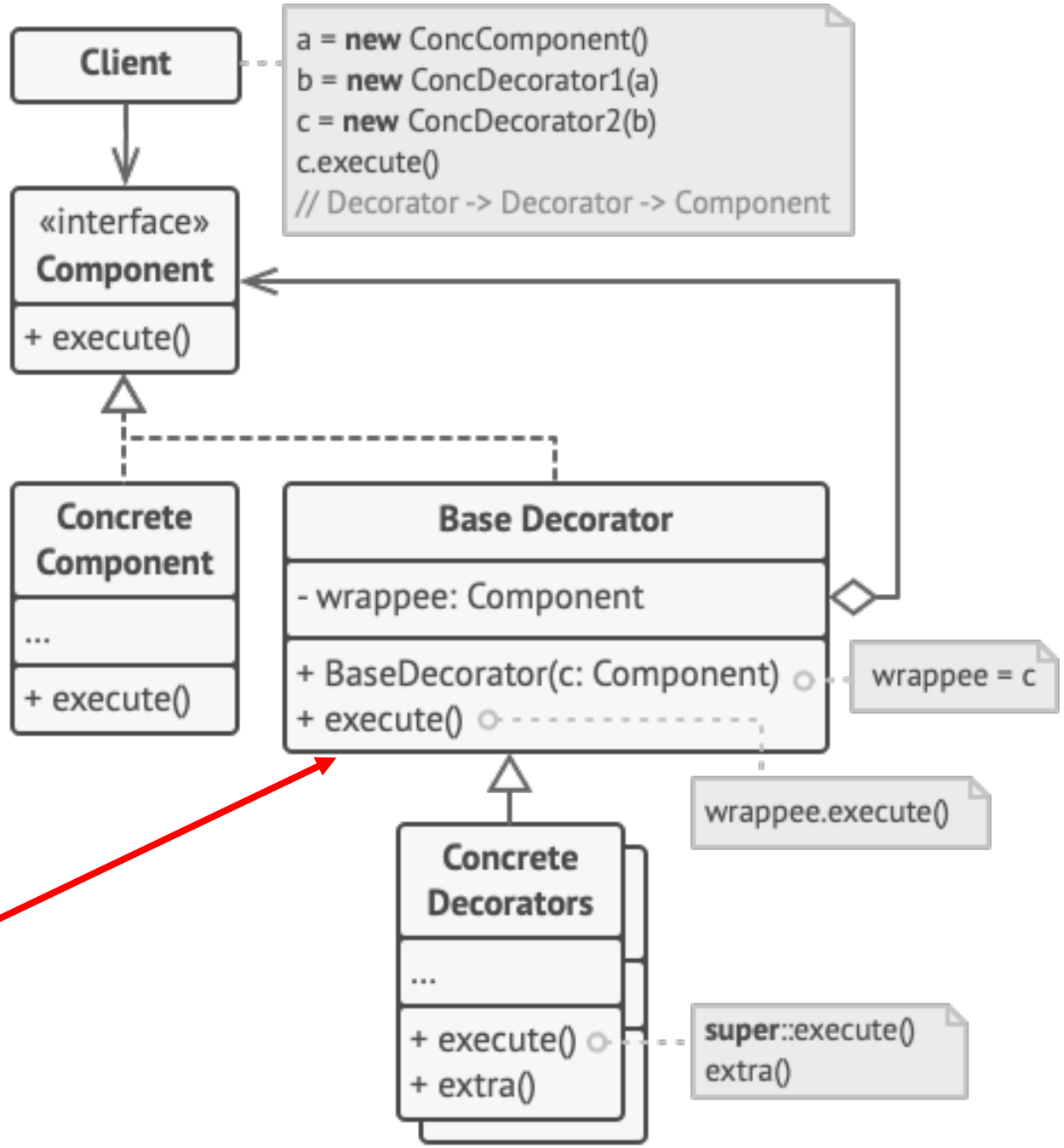


Each component can be used on its own, or wrapped by a decorator.

The **ConcreteComponent** is the object we're going to dynamically add new behavior to.



Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.

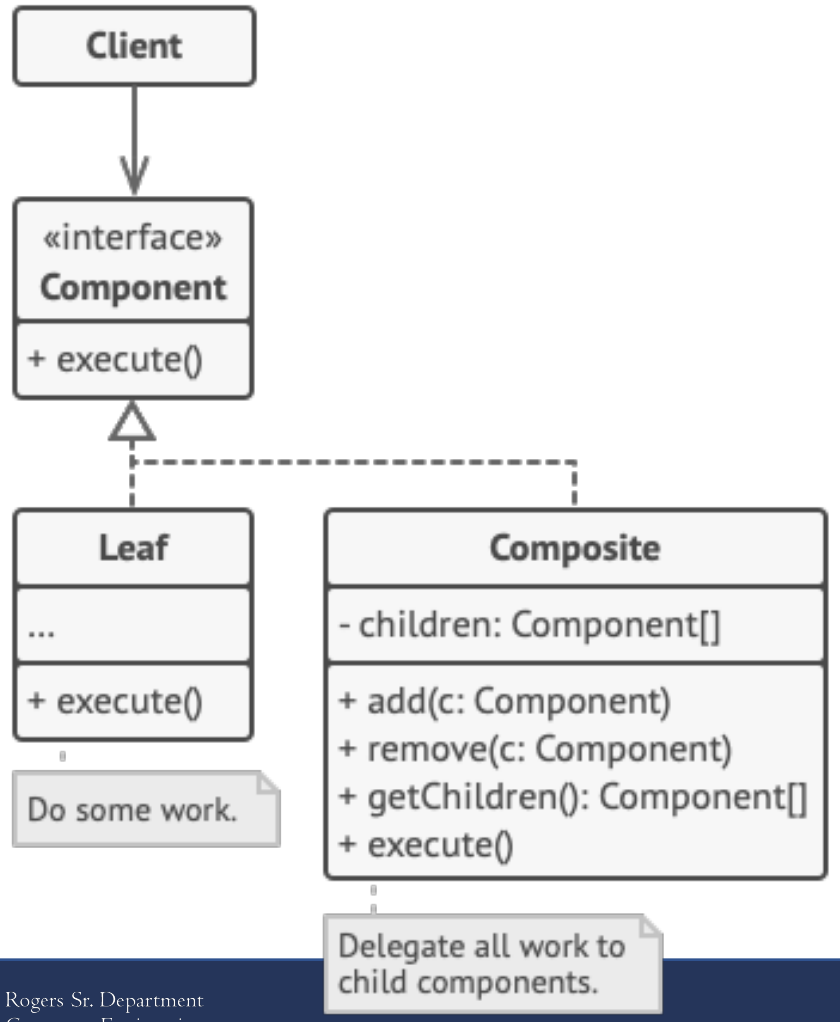


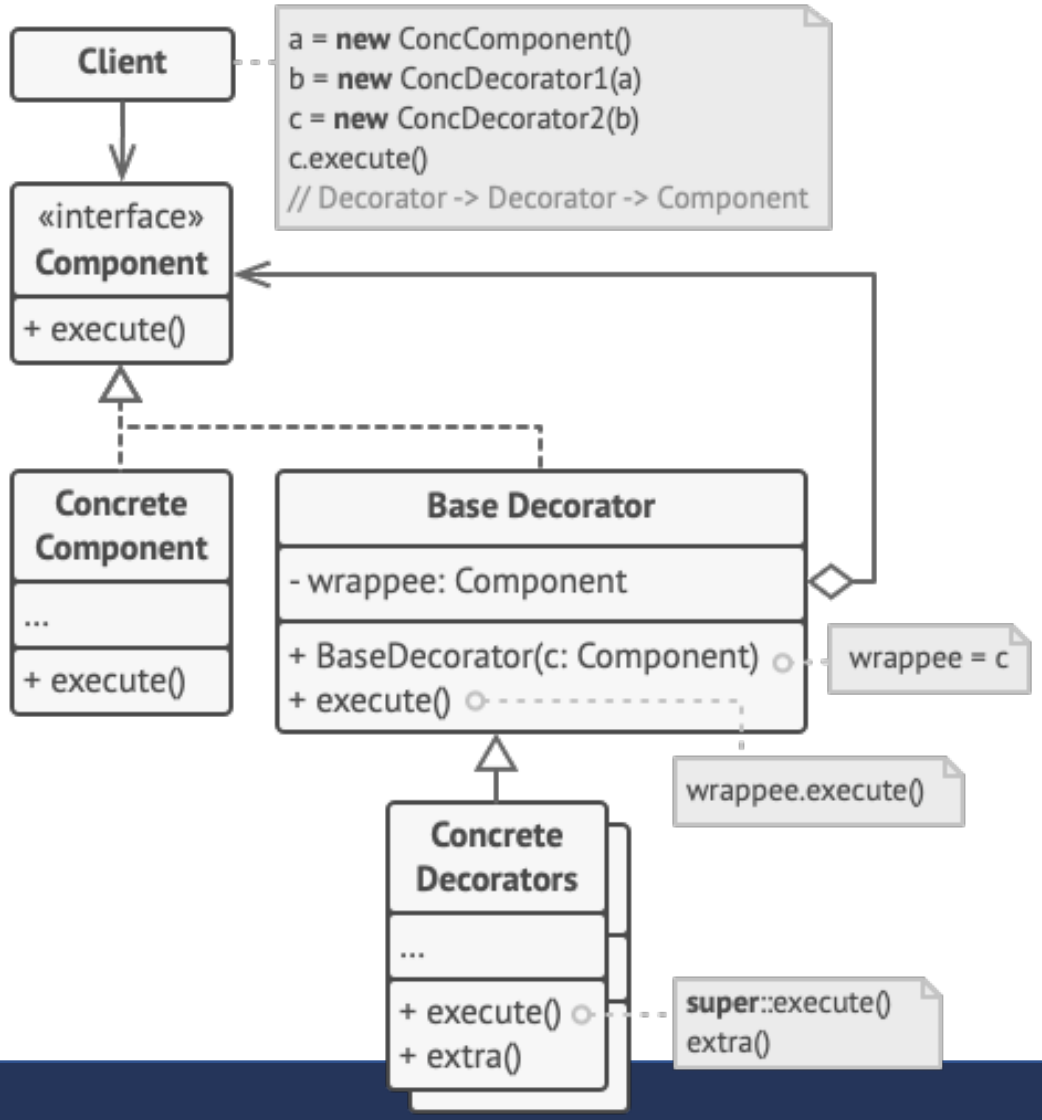
Decorator Pattern

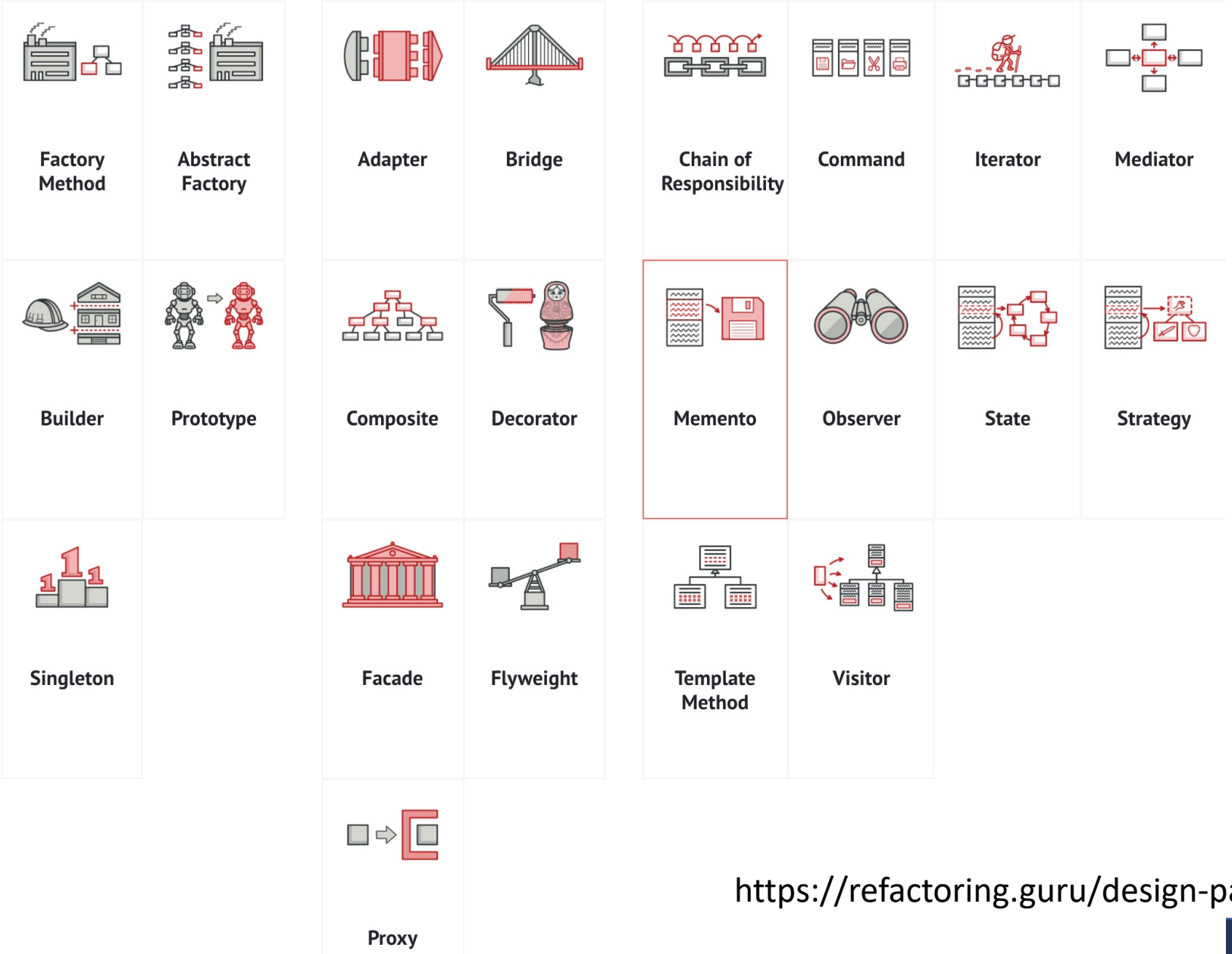
- ✓ You can extend an object's behavior without making a new subclass.
- ✓ You can add or remove responsibilities from an object at runtime.
- ✓ You can combine several behaviors by wrapping an object into multiple decorators.
- ✓ *Single Responsibility Principle*. You can divide a monolithic class that implements many possible variants of behavior into several smaller classes.

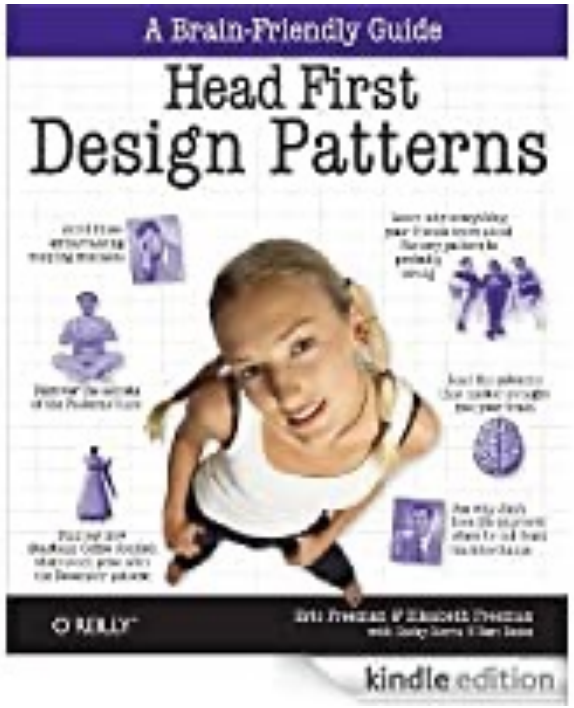
- ✗ It's hard to remove a specific wrapper from the wrappers stack.
- ✗ It's hard to implement a decorator in such a way that its behavior doesn't depend on the order in the decorators stack.
- ✗ The initial configuration code of layers might look pretty ugly.

Composite vs Decorator

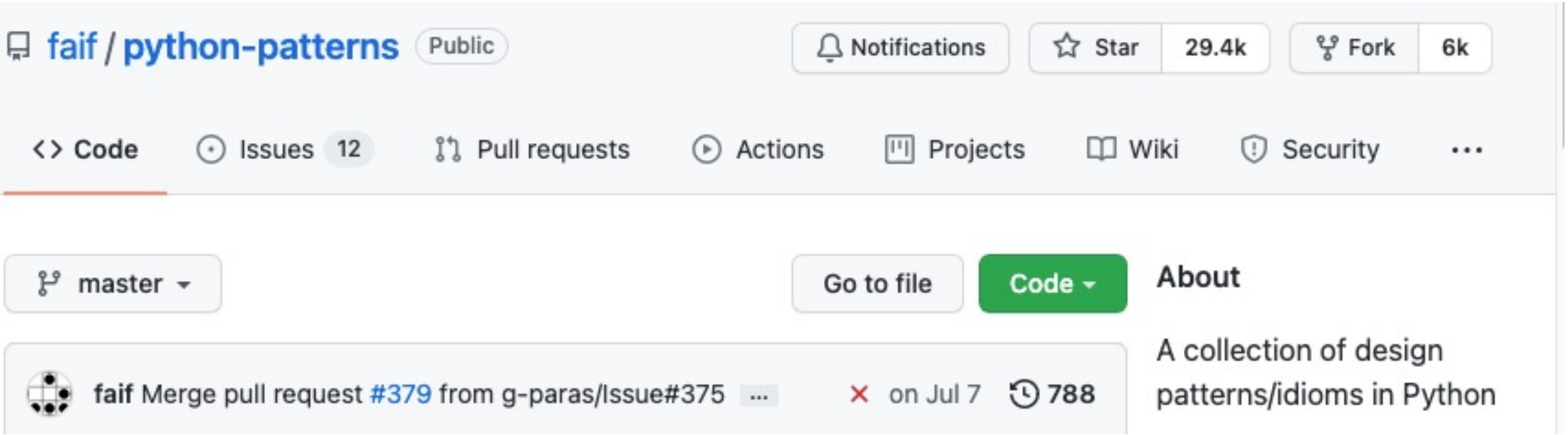








https://refactoring.guru/design-patterns/catalog



- https://github.com/faif/python-patterns

https://www.youtube.com/watch?v=bsyjSW46TDg

Criticism of Design Patterns

- **Kludges for a weak programming language**

Usually the need for patterns arises when people choose a programming language or a technology that lacks the necessary level of abstraction.

- **Inefficient solutions**

Patterns try to systematize approaches that are already widely used.

- **Unjustified use**

If all you have is a hammer, everything looks like a nail.

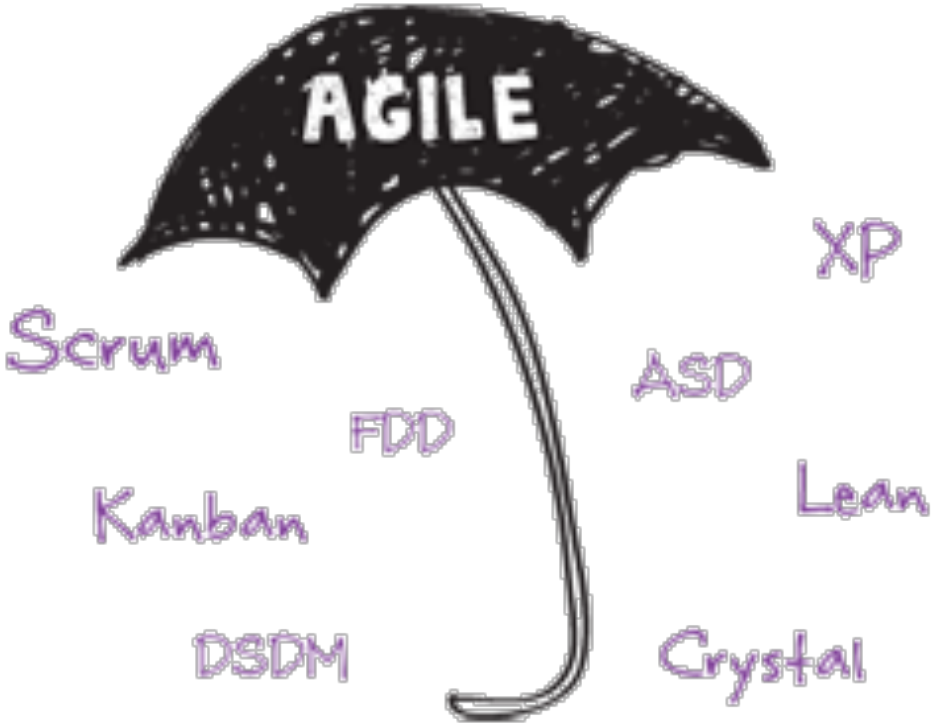
Cargo cult programming

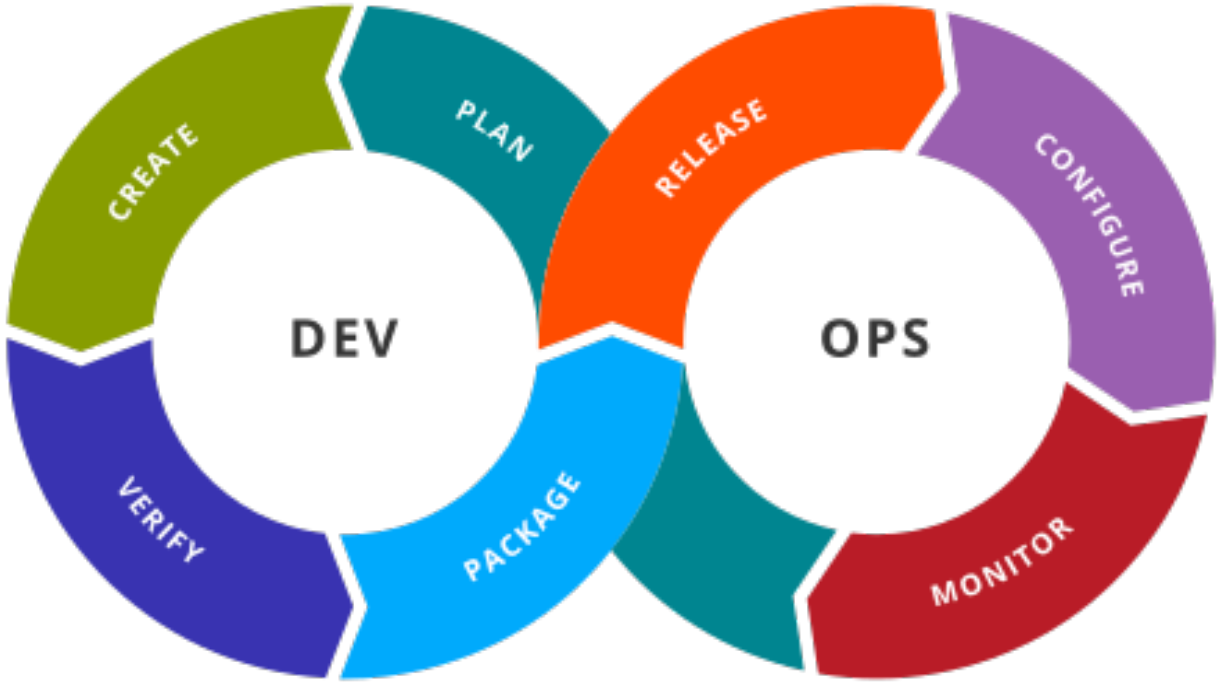
Are SOLID principles Cargo Cult?

It looks like a plane, but will it fly?

https://blog.ndepend.com/are-solid-principles-cargo-cult/

Developers + Operators = DevOps





Continuous Integration

- Merging in small code changes frequently

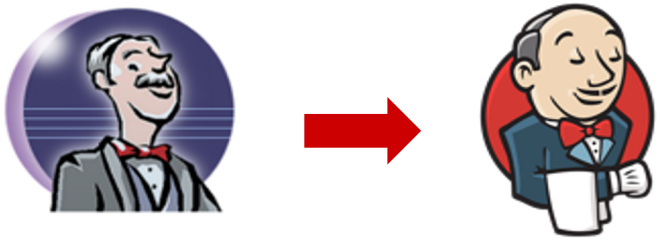
Continuous Delivery

- Add additional automation and testing, get the code nearly ready to deploy with almost no human intervention

Continuous Deployment

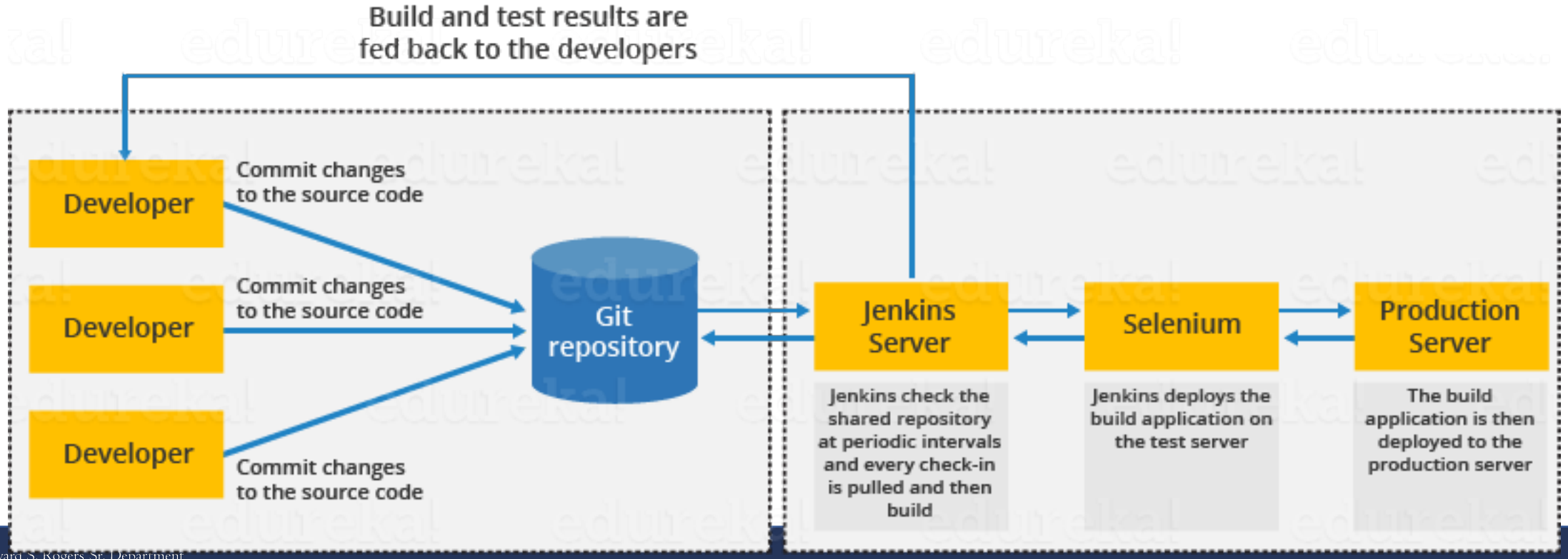
- Deploying all the way into production without any human intervention.

Tools - Continuous Integration



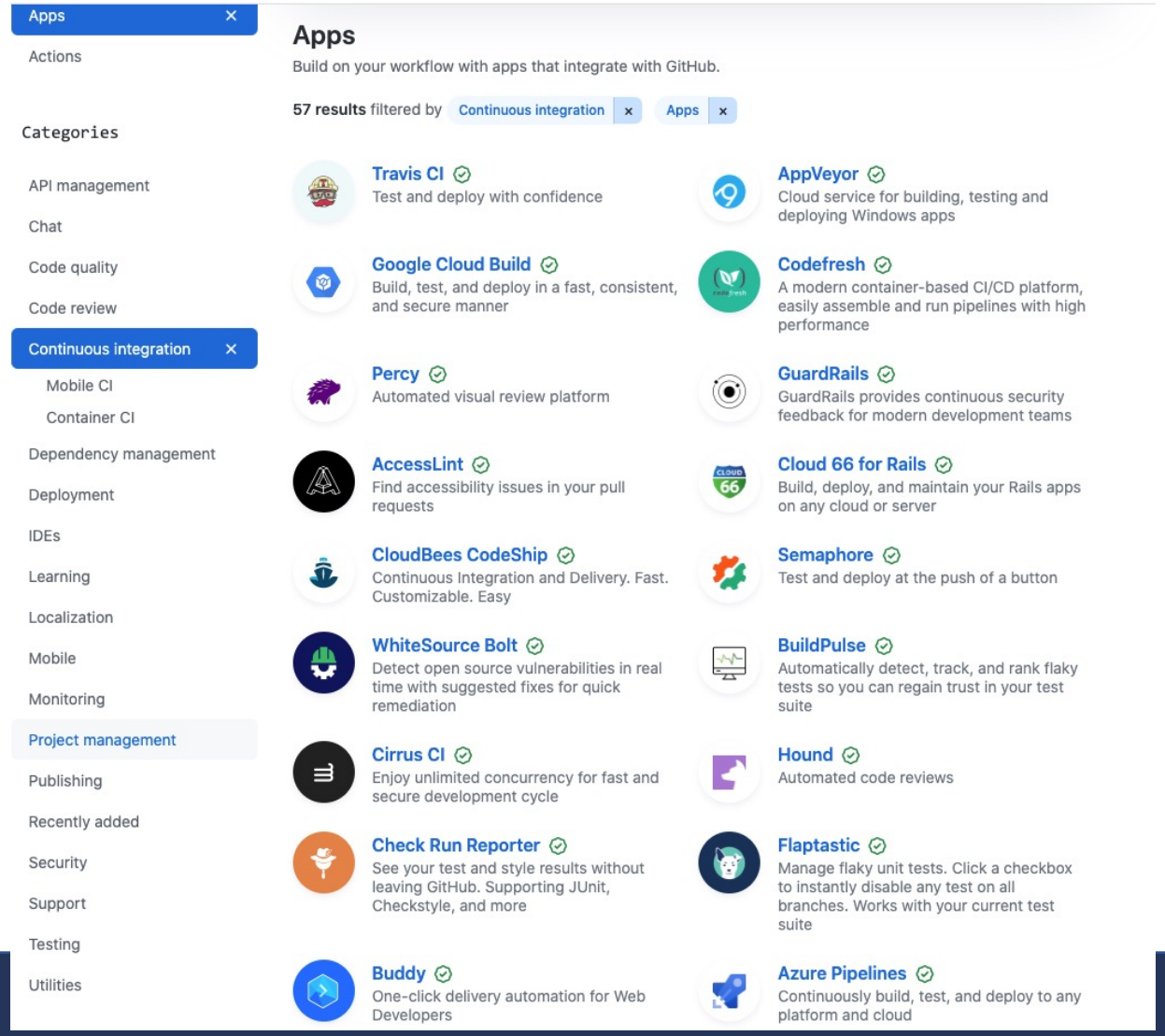
Hudson Jenkins

- Quickly integrating newly developed code with the main body of code that is to be released



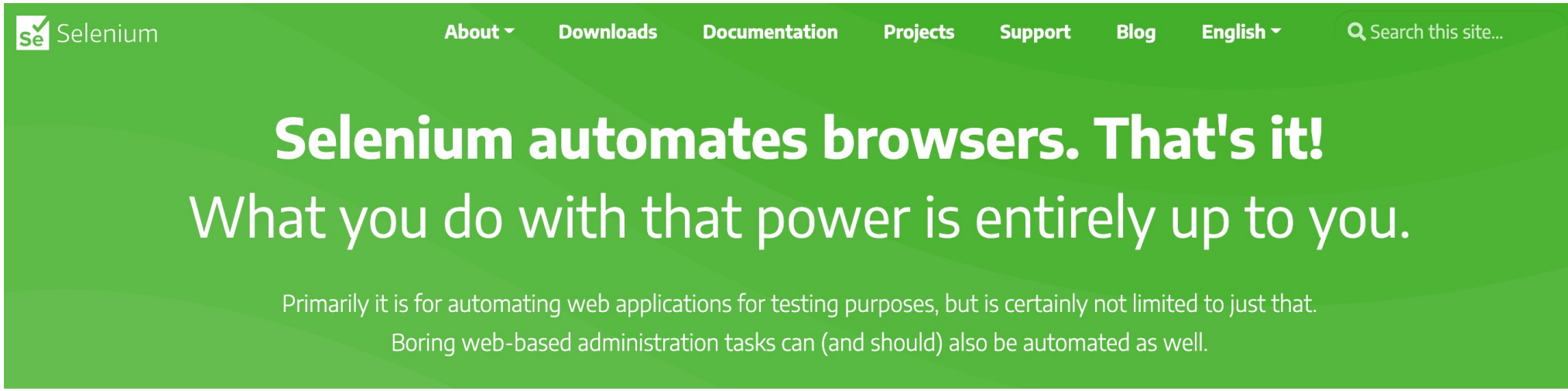
Continuous Integration

https://martinfowler.com/articles/continuousIntegration.html

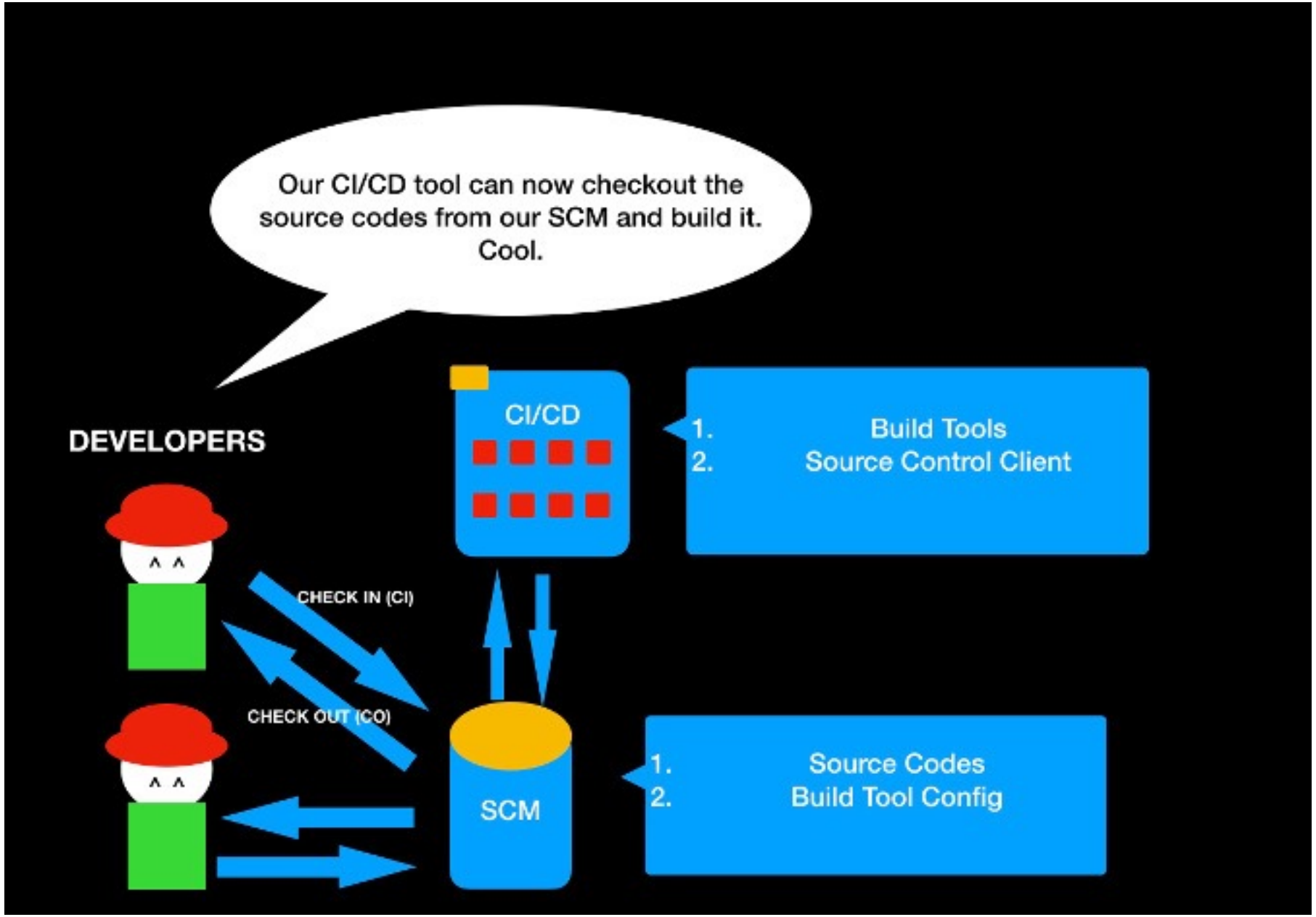


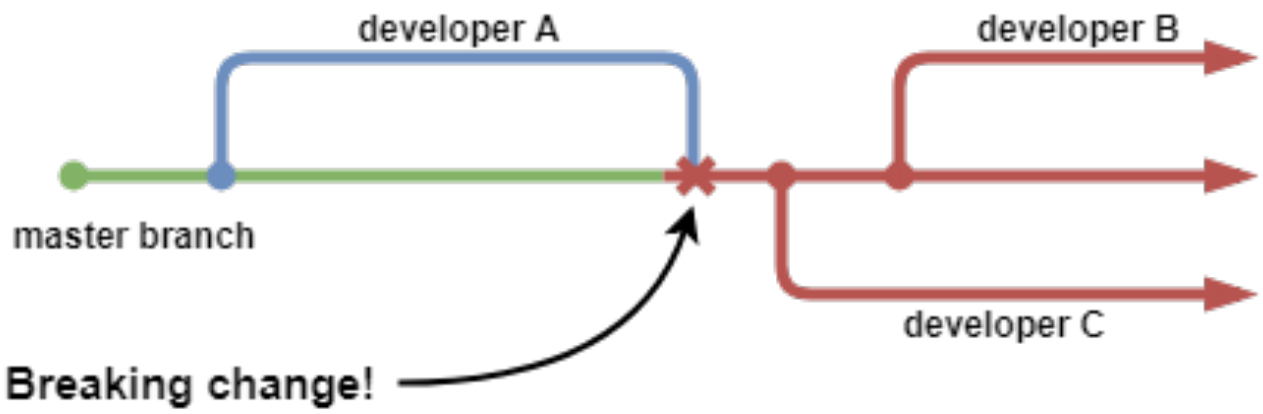
Continuous Testing

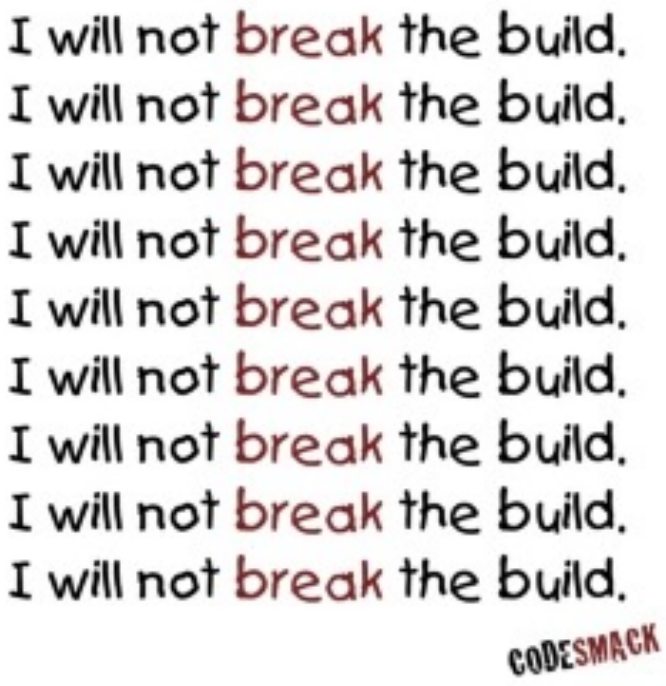
- **Selenium**



Build


Our CI/CD tool can now checkout the source codes from our SCM and build it. Cool.
DEVELOPERS
CI/CD
1. Build Tools
2. Source Control Client
CHECK IN (CI)
CHECK OUT (CO)
SCM
1. Source Codes
2. Build Tool Config


developer A
developer B
master branch
developer C
Breaking change!


I will not break the build.
I will not break the build.
I will not break the build.
I will not break the build.
I will not break the build.
I will not break the build.
I will not break the build.
I will not break the build.
I will not break the build.
CODESMACK

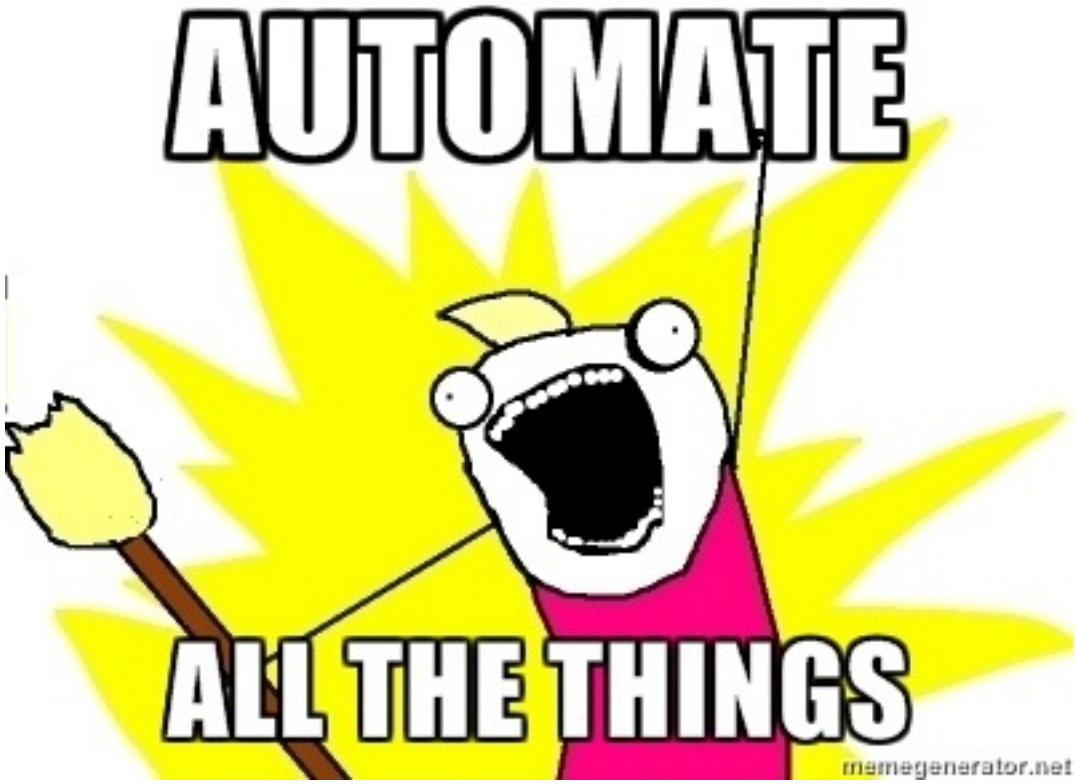

ALL I HAVE TO DO IS CHECK IN MY LATEST CHANGES TO THE CURRENT BUILD
AND... IT'S BROKE!
imgflip.com

Brian the Build Bunny

http://www.woodwardweb.com/gadgets/000434.html

- Lightweight virtualization
- Separate docker images for separate services (web server, business logic, database, …)

Automate all the things



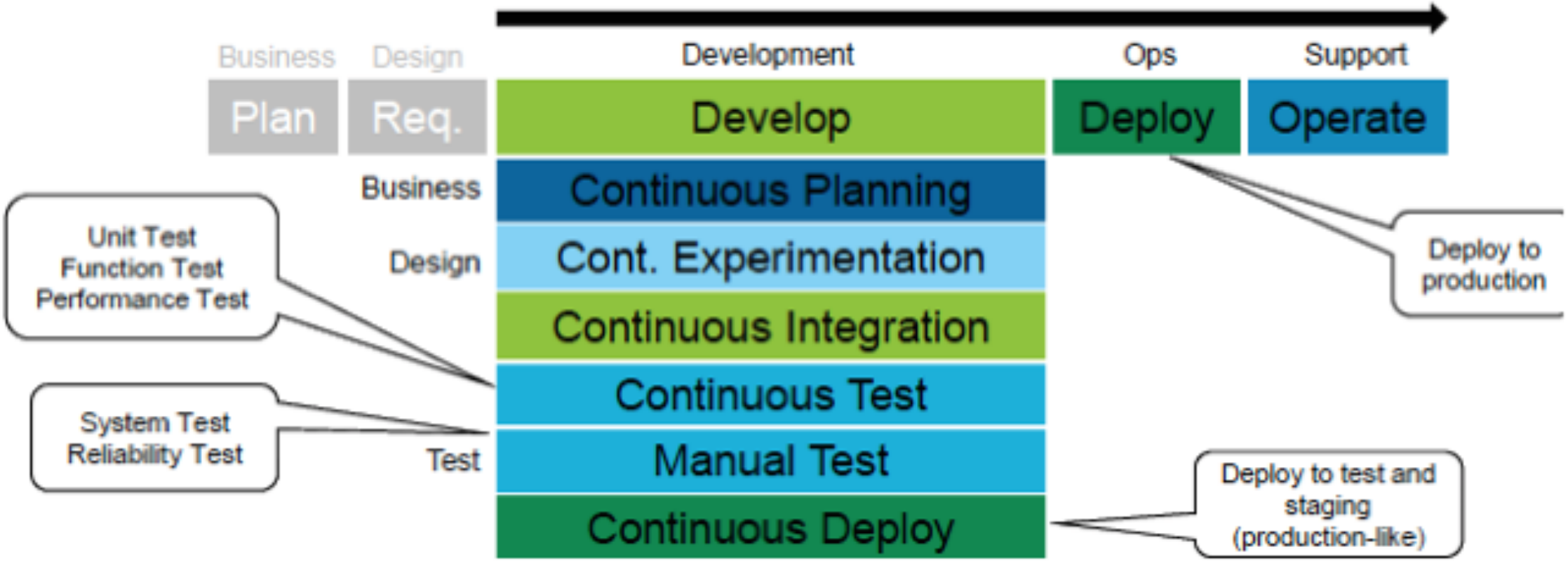
INSTALL.SH

```
#!/bin/bash

pip install "$1" &
easy_install "$1" &
brew install "$1" &
npm install "$1" &
yum install "$1" & dnf install "$1" &
docker run "$1" &
pkg install "$1" &
apt-get install "$1" &
sudo apt-get install "$1" &
steamcmd +app_update "$1" validate &
git clone https://github.com/"$1"/"$1" &
cd "$1";./configure;make;make install &
curl "$1" | bash &
```

Continuous Deployment

Continuous * (Perpetual Development)



Continuous Deployment of Mobile Software at Facebook (Showcase)

Chuck Rossi
Facebook Inc.
1 Hacker Way
Menlo Park, CA USA 94025
chuckr@fb.com

Elisa Shibley
University of Michigan
2260 Hayward Street
Ann Arbor, MI USA 48109
eshibley@umich.edu

Shi Su
Carnegie Mellon University
PO Box 1
Moffett Field, CA USA 94035
shis@andrew.cmu.edu

Kent Beck
Facebook Inc.
1 Hacker Way
Menlo Park, CA USA 94025
kbeck@fb.com

Tony Savor
Facebook Inc.
1 Hacker Way
Menlo Park, CA USA 94025
tsavor@fb.com

Michael Stumm
University of Toronto
10 Kings College Rd
Toronto, Canada M8X 2A6
stumm@eecg.toronto.edu

Release engineering and push karma: Chuck Rossi

Facebook Engineering
April 5, 2012 · 5 min read ·

Quality Assurance 3

Testing, Analysis

The Edward S. Rogers Sr. Department
of Electrical & Computer Engineering
UNIVERSITY OF TORONTO

Definition: software analysis

The **systematic** examination of a software artifact to determine its properties.

- Attempting to be comprehensive, as measured by, as examples:

  Test coverage, inspection checklists, exhaustive model checking.

Type	ID	Checkpoint	Yes/No	Comments
General	1	Identify the potential target users of the system		
		- Demographics		
		- User groups		
	2	What aspects of the application is sensitive to HW and SW differences		
	3	Are there any universal standards and guidelines, to which the application should adhere [E.g. iPhone]		
OS	1	Create OS compatibility matrix		
	2	Get client confirmation for OS compatibility matrix		
	3	Identify testing scope [domain specific]		
	4	Setup multiple virtual machines for each OS		
Browser	1	Create Browser compatibility matrix		
	2	Get client confirmation for Browser compatibility matrix		
	3	Identify testing scope [domain specific] - Include most navigable and most frequently accessible pages		
	4	Whether to use Downgradable Browser Versions		
	5	Setup multiple virtual machines if applicable		
Device	1	Create Device compatibility matrix		
	2	Get client confirmation for Device compatibility matrix		
	3	Identify testing scope [Domain specific + UI aspects + Configurations]		
	4	Setup simulators [For Mobile Devices]		
	5	Should application work on jail-broken/rooted devices?		
Network	1	Create scope on possible access points to system [Dial-up, wireless, 4G, low bandwidth, with proxy, without proxy..etc.]		
	2	Create scope on possible access points from system [Printer in same network, access to internet, access external network via firewall]		
	3	Get client confirmation on the possible access points identified		
	4	Environment setup for each network configuration		

https://rochanaqa.wordpress.com/2015/10/05/how-to-plan-and-test-compatibility-using-simple-checklists/

Classic Testing (Functional Correctness)

Testing

- Executing the program with selected inputs in a controlled environment (dynamic analysis)
- Goals:
  - Reveal bugs (main goal)
  - Assess quality (hard to quantify)
  - Clarify the specification, documentation
  - Verify contracts

"Testing shows the presence, not the absence of bugs

Edsger W. Dijkstra 1969

What are we covering?

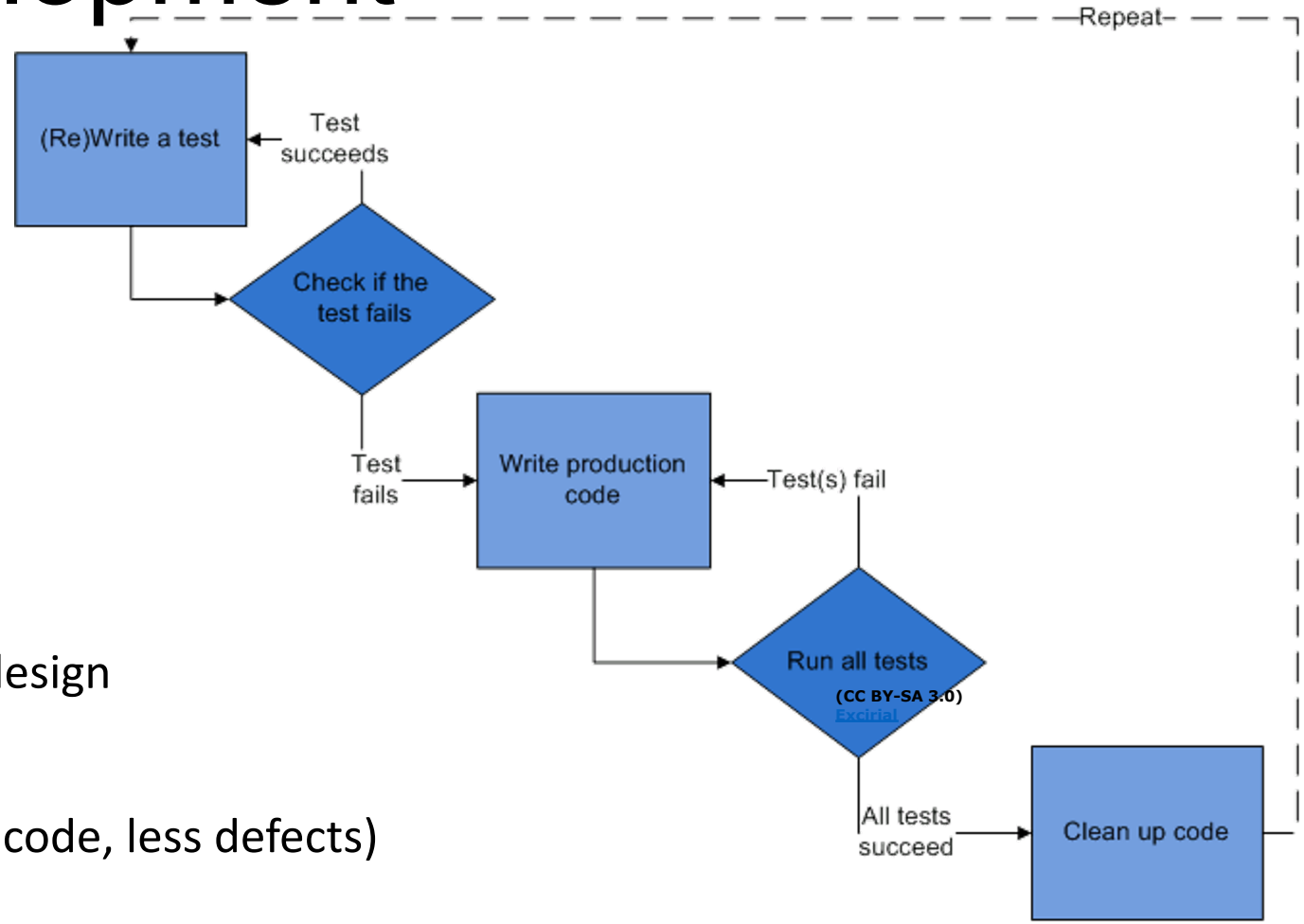
- Program/system functionality:
  - Execution space (white box!).
  - Input or requirements space (black box!).
- The expected user experience (usability).
  - GUI testing, A/B testing
- The expected performance envelope (performance, reliability, robustness, integration).
  - Security, robustness, fuzz, and infrastructure testing.
  - Performance and reliability: soak and stress testing.
  - Integration and reliability: API/protocol testing

Testing Levels

- Unit testing
- Integration testing
- System testing

Test Driven Development

- Tests first!
- Popular agile technique
- Write tests as specifications before code
- Never write code without a failing test
- Claims:
  - Design approach toward testable design
  - Think about interfaces first
  - Avoid writing unneeded code
  - Higher product quality (e.g. better code, less defects)
  - Higher test suite quality
  - Higher overall productivity



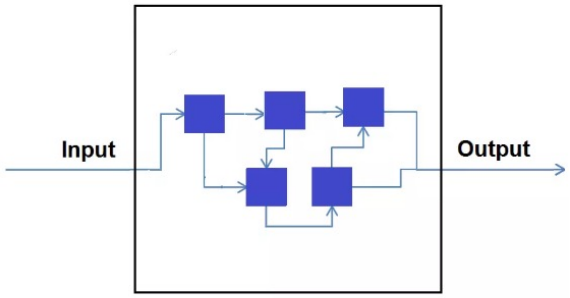
“Traditional” coverage

- **Statement**: Has each statement in the program been executed?
- **Branch**: Has each of each control structure been executed?
- **Function**: Has each function in the program been called?
- **Path**: requires that all paths through the Control Flow Graph are covered.
- …

We can measure coverage on almost anything

Common adequacy criteria for testing approximate **full "coverage"** of the program execution or specification space.

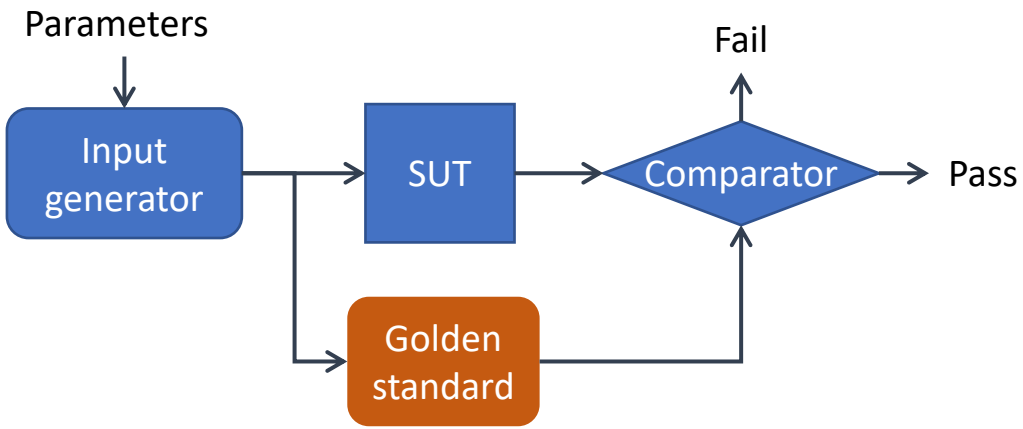
White box testing

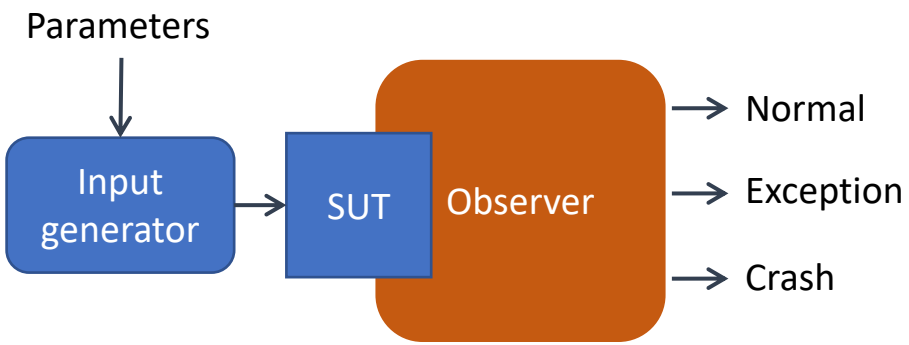


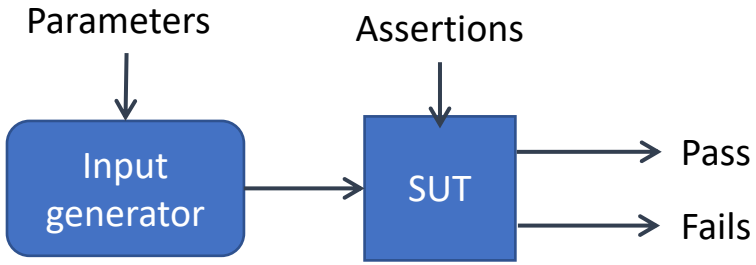
Tests internal structures or workings of an application, as opposed to its functionality.

- Unit Test
- Testing for Memory Leaks
- Penetration Testing
  - ***"What would a cybercriminal do to harm my organization' computer systems, applications, and network?"***

The Oracle Problem



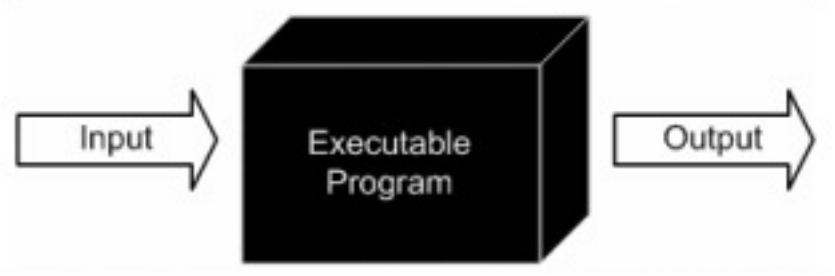




System under test (SUT)

https://www.youtube.com/watch?v=q2t91jLmh3k

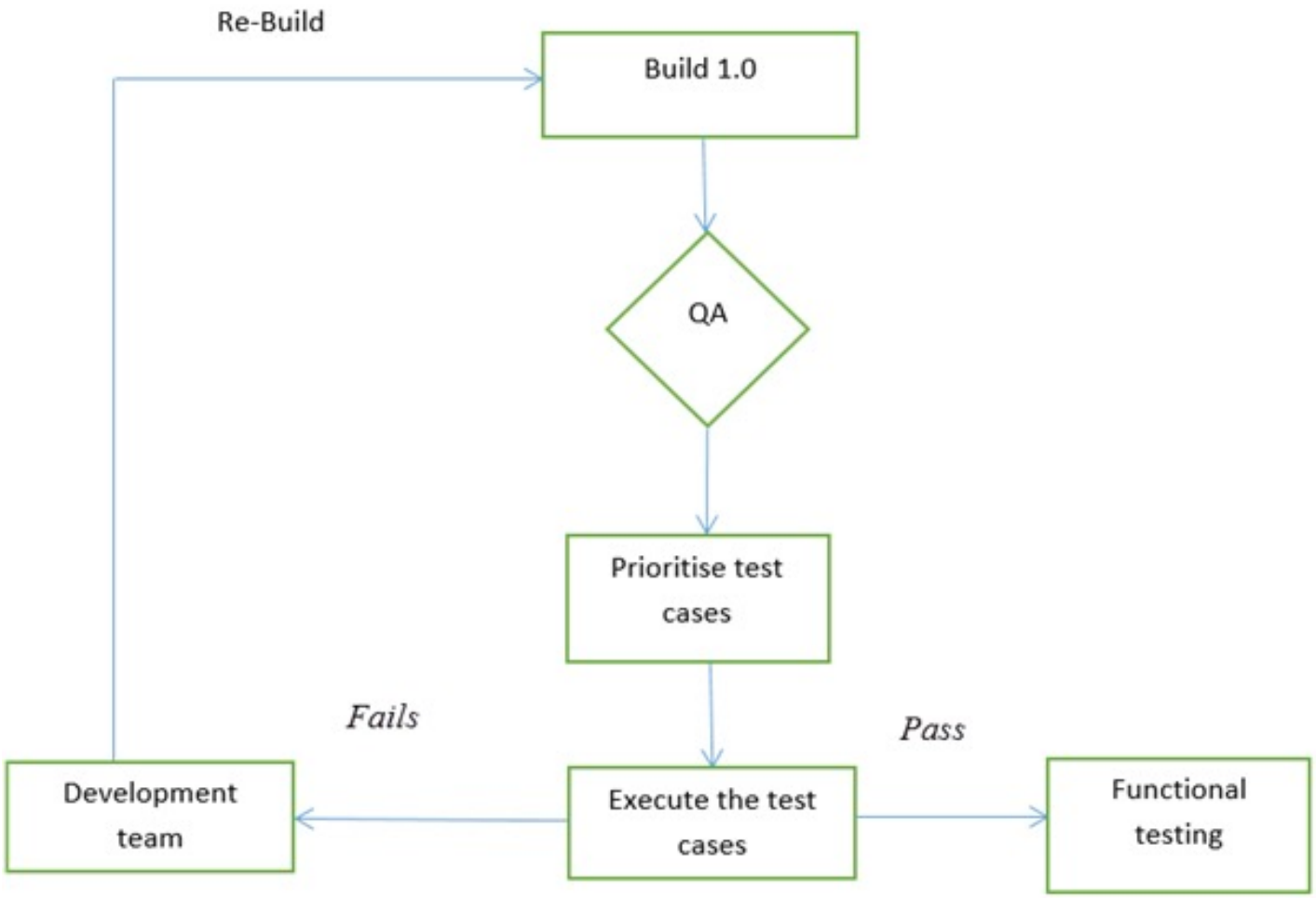
Black box testing



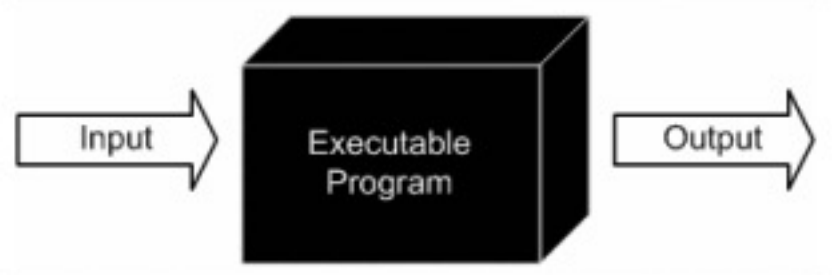
- Functionality of application is tested without looking at the implementation details
- Types
  - **Functional Testing**
    - Smoke Testing
    - Regression Testing
    - …
  - **Non-Functional Testing**
    - Performance Testing
    - Compatibility Testing
    - Stress Testing

Smoke Testing

- Determines whether the deployed software build is stable or not.
- We perform smoke testing on a new build.



Black box testing



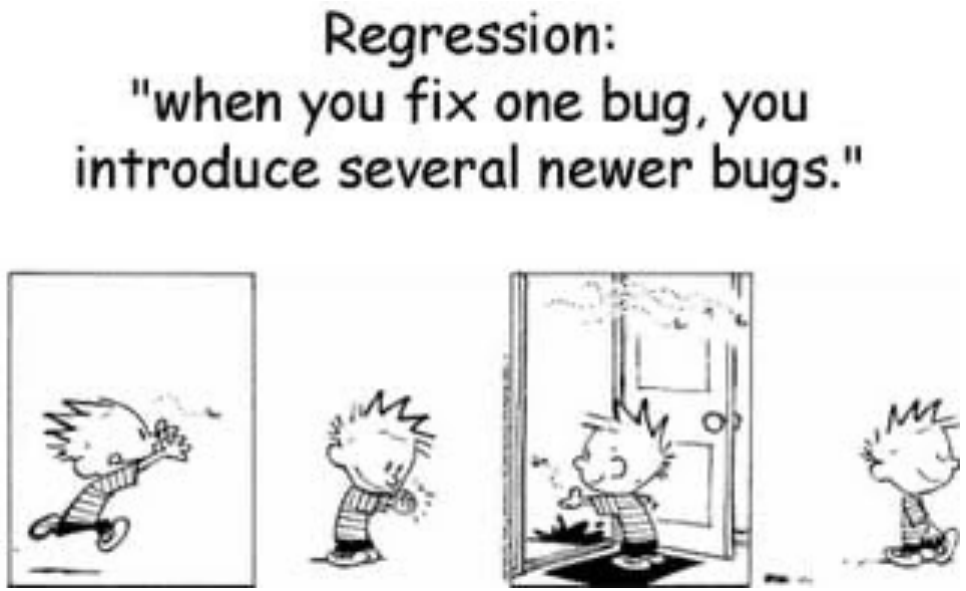
- Functionality of application is tested without looking at the implementation details
- Types
  - **Functional Testing**
    - Smoke Testing
    - Regression Testing
    - …
  - **Non-Functional Testing**
    - Performance Testing
    - Compatibility Testing
    - Stress Testing

Regression testing

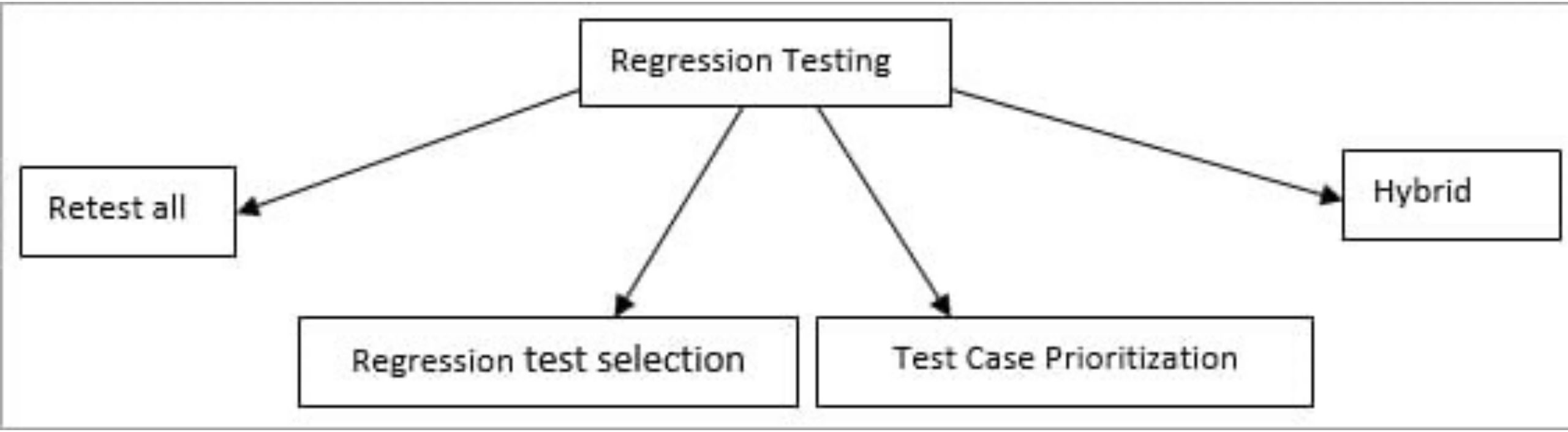
- Ensure that a small change in one part of the system does not break existing functionality elsewhere in the system.

Regression testing

- Ensure that a small change in one part of the system does not break existing functionality elsewhere in the system.
- Application scenario:
  - When new functionalities are added
  - In case of change requirements
  - When there is a defect fix
  - When there are performance issues
  - In case of environment changes
  - When there is a patch fix



4 Types of Regression Testing



What are we covering?

- Program/system functionality:
  - Execution space (white box!).
  - Input or requirements space (black box!).
- The expected user experience (usability).
  - GUI testing, A/B testing
- The expected performance envelope (performance, reliability, robustness, integration).
  - Security, robustness, fuzz, and infrastructure testing.
  - Performance and reliability: soak and stress testing.
  - Integration and reliability: API/protocol testing

Manual Testing?

GENERIC TEST CASE: USER SENDS MMS WITH PICTURE ATTACHED.

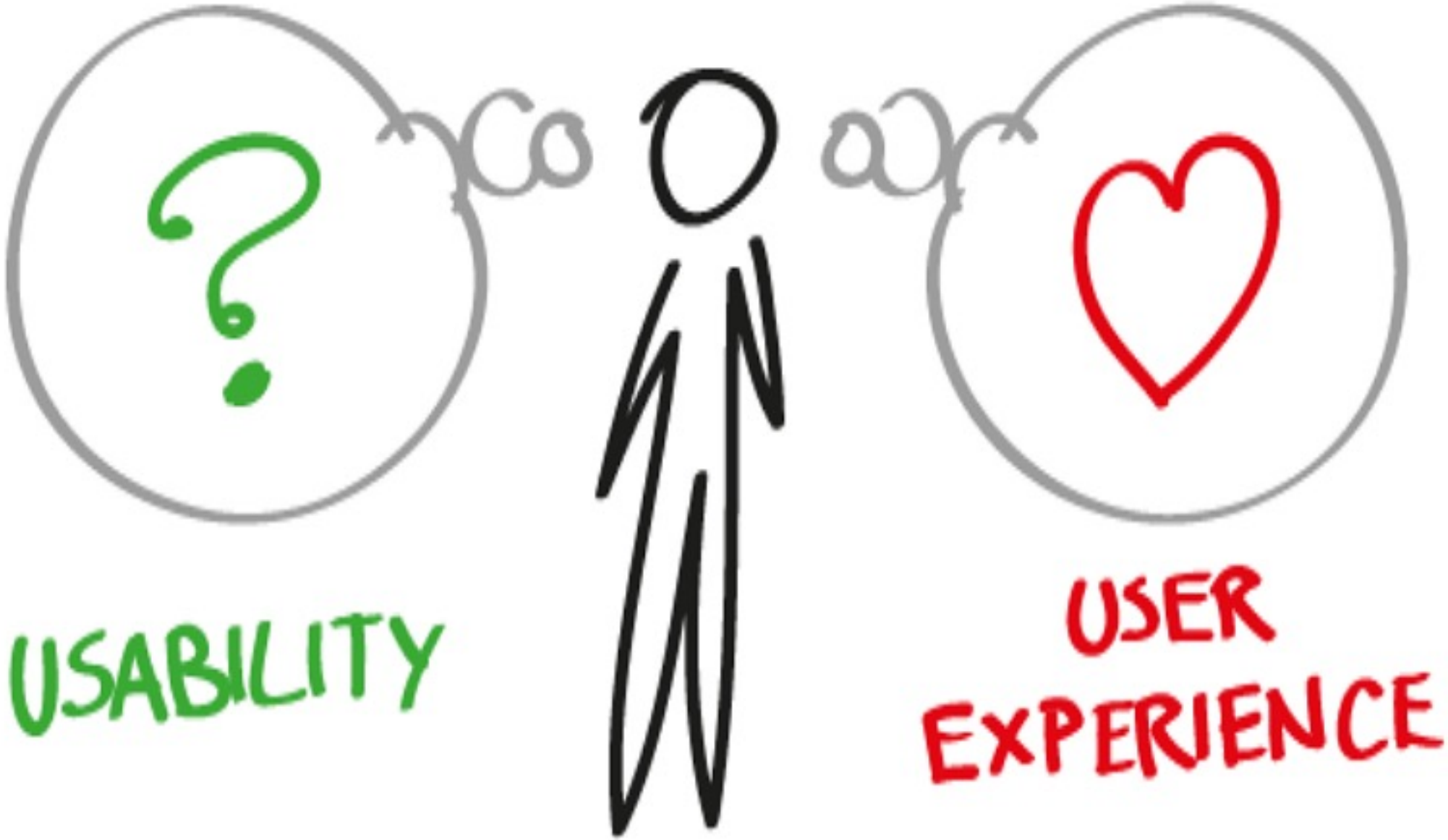
Step ID	User Action	System Response
1	Go to Main Menu	Main Menu appears
2	Go to Messages Menu	Message Menu appears
3	Select "Create new Message"	Message Editor screen opens
4	Add Recipient	Recipient is added
5	Select "Insert Picture"	Insert Picture Menu opens
6	Select Picture	Picture is Selected
7	Select "Send Message"	Message is correctly sent

- Live System?
- Extra Testing System?
- Check output / assertions?
- Effort, Costs?
- Reproducible?

https://www.youtube.com/watch?v=UmAa8UJATkE

Automating GUI/Web Testing

- First: why is this hard?
- Capture and Replay Strategy
  - mouse actions
  - system events
- Test Scripts: (click on button labeled "Start" expect value X in field Y)
- Lots of tools and frameworks
  - e.g. JUnit + Jemmy for Java/Swing


USABILITY
USER EXPERIENCE

A/B testing

Usability: A/B testing

- Controlled randomized experiment with two variants, A and B, which are the control and treatment.
- One group of users given A (current system); another random group presented with B; outcomes compared.
- Often used in web or GUI-based applications, especially to test advertising or GUI element placement or design decisions.

Example

- A company sends an advertising email to its customer database, varying the photograph used in the ad...

Example: group A (99% of users)

- Act now! Sale ends soon!

Example: group B (1%)

- Act now! Sale ends soon!

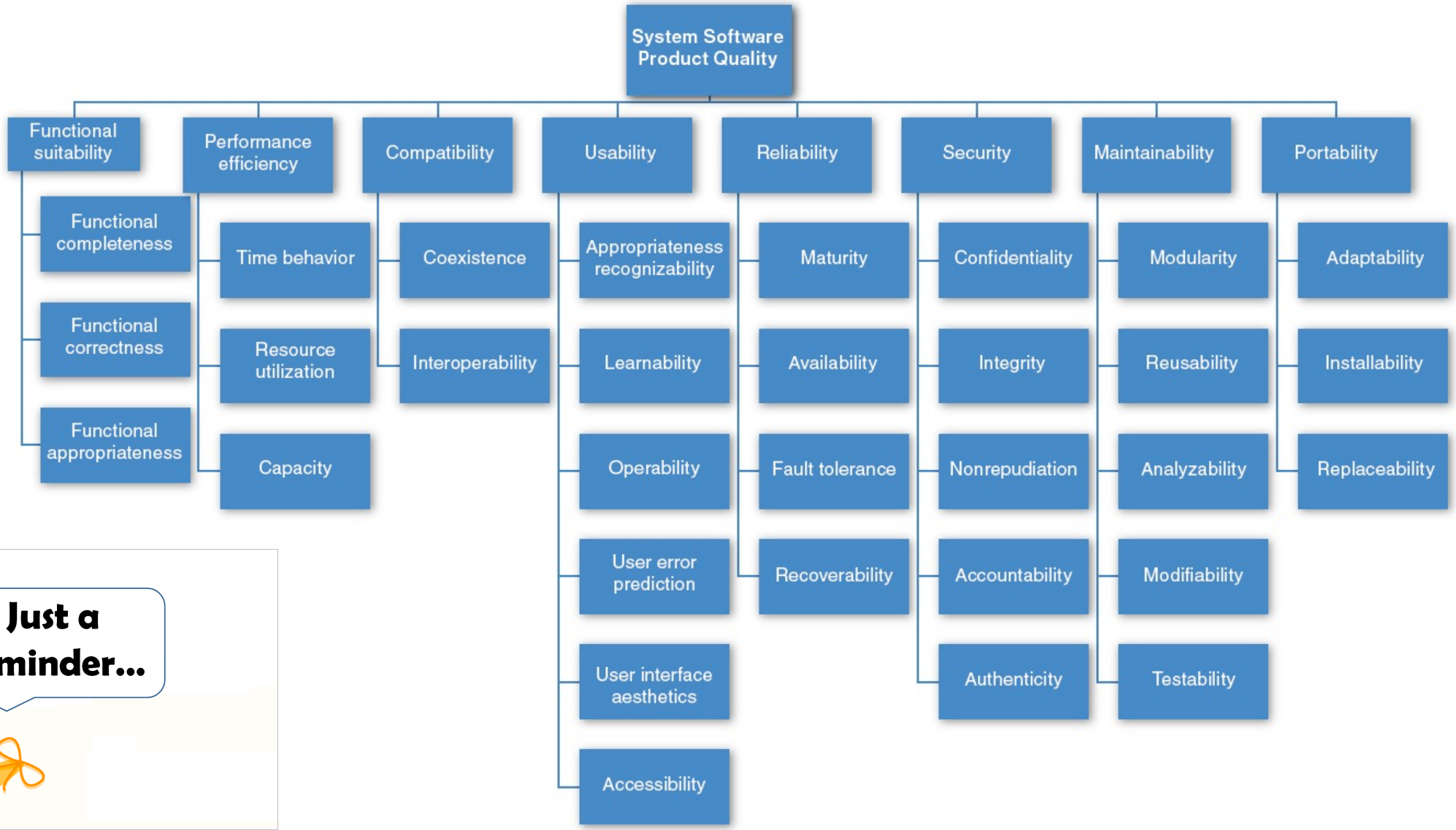
Usability: A/B testing

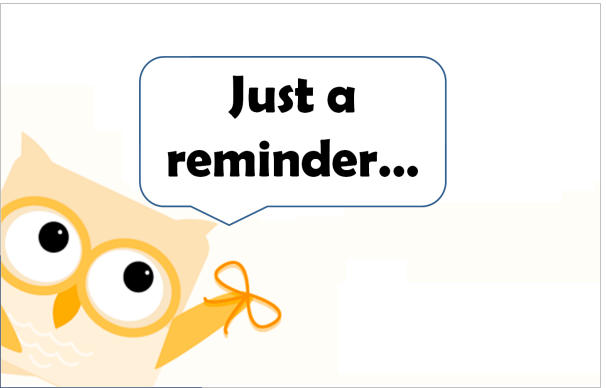
- However, it cannot..
  - Tell you why
  - Let you test drastic redesigns of your website or app.
  - Tell you if you're solving the right/wrong problem.

What are we covering?

- Program/system functionality:
  - Execution space (white box!).
  - Input or requirements space (black box!).
- The expected user experience (usability).
  - GUI testing, A/B testing
- The expected performance envelope (performance, reliability, robustness, integration).
  - Security, robustness, fuzz, and infrastructure testing.
  - Performance and reliability: soak and stress testing.
  - Integration and reliability: API/protocol testing

Quality Attributes



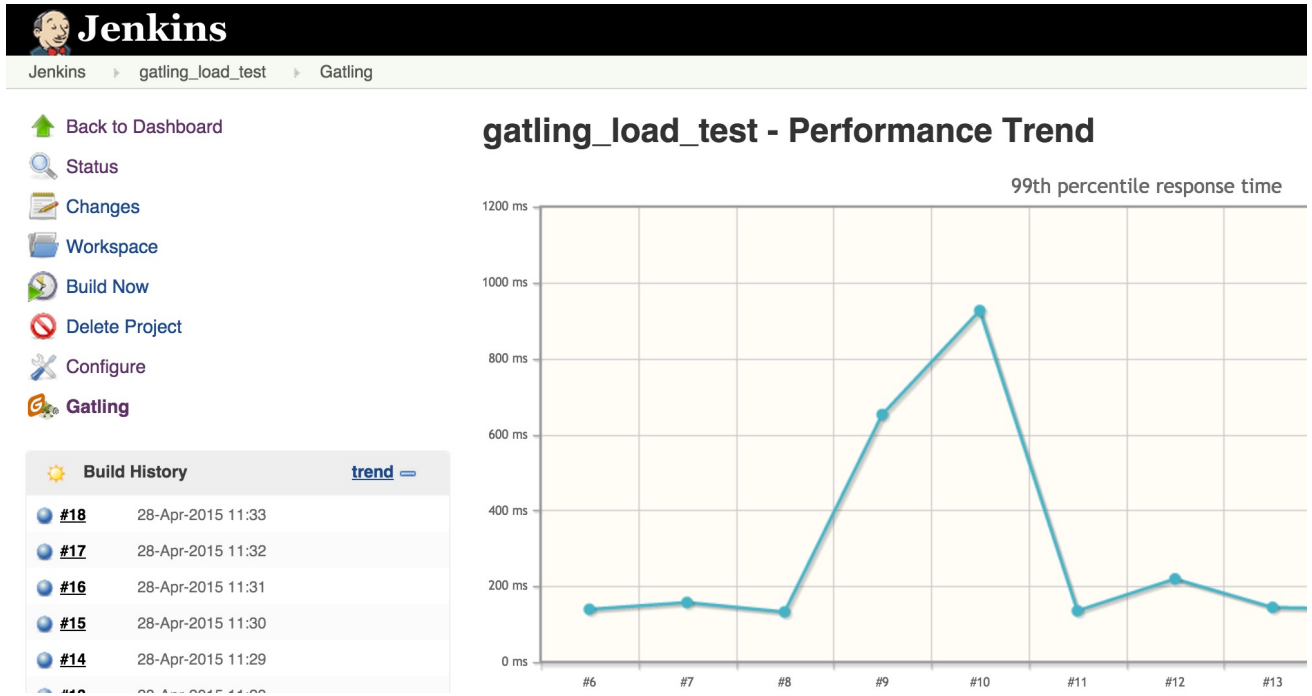


Performance Testing

- Specification? Oracle?
- Test harness? Environment?
- Nondeterminism?
- Unit testing?
- Automation?
- Coverage?

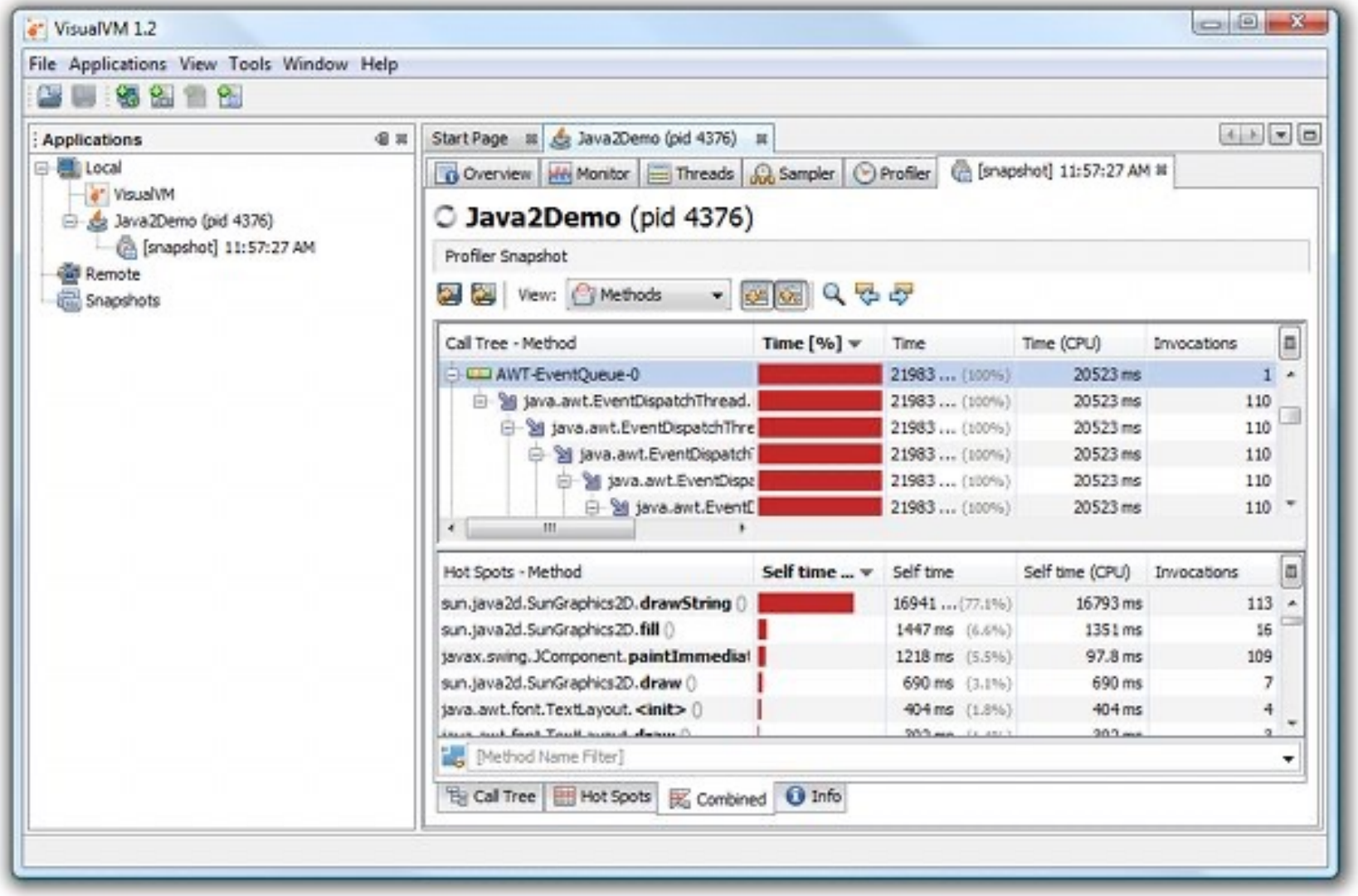
Unit and regression testing for performance

- Measure execution time of critical components
- Log execution times and compare over time



Profiling

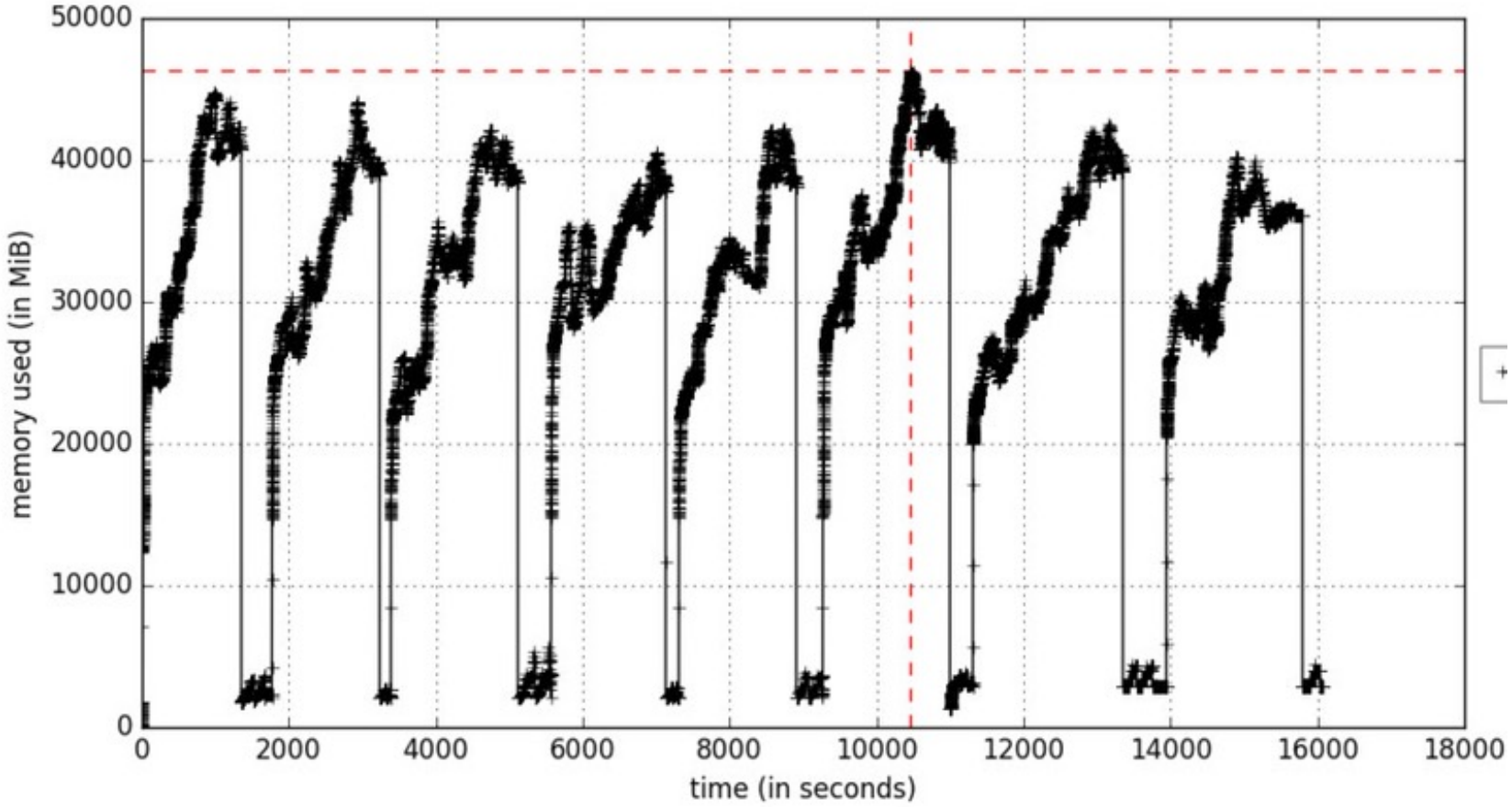
- Finding bottlenecks i execution time and memory

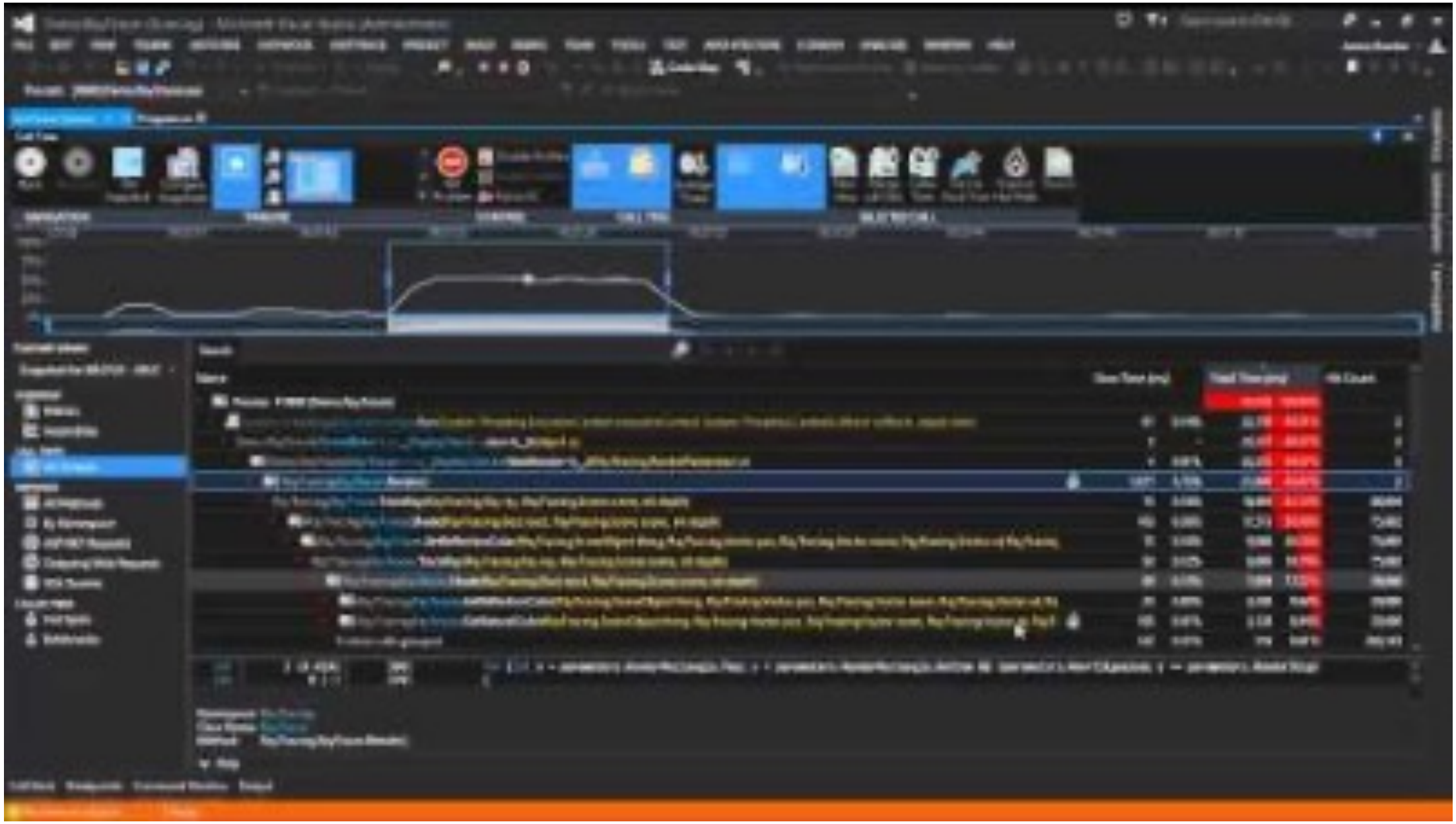


Profiling

- Memory profile as a function of time

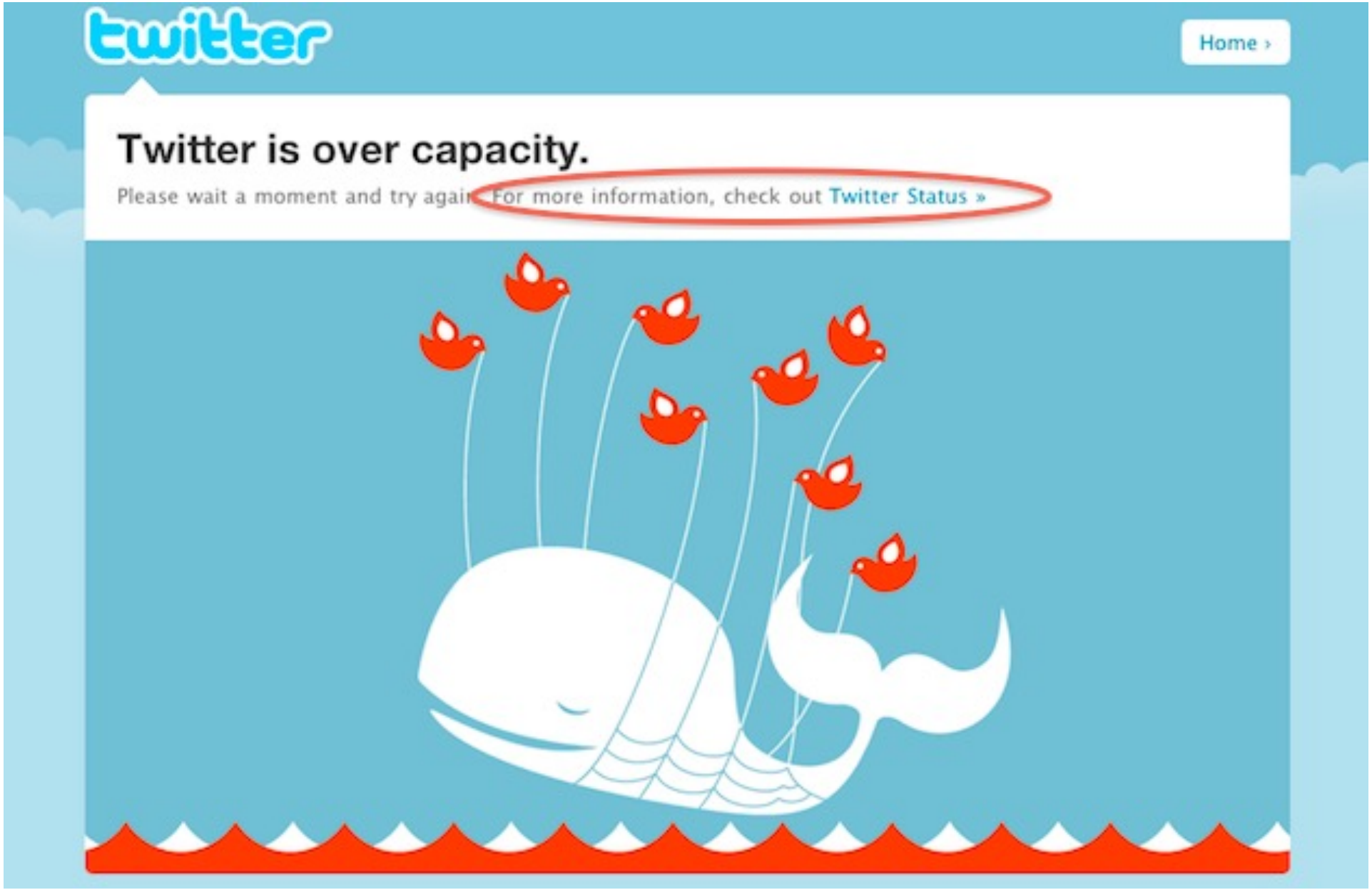
memory-profile package



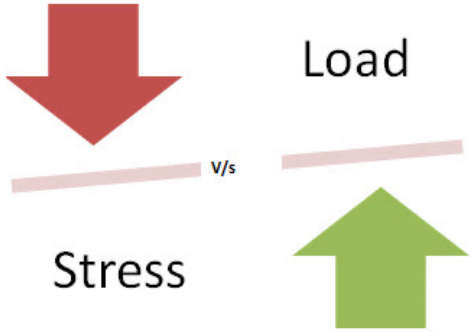


https://www.youtube.com/watch?v=0sEgkZ27gtY https://www.telerik.com/

Robustness Testing


twitter
Home ›
Twitter is over capacity.
Please wait a moment and try again. For more information, check out Twitter Status »

Robustness: Stress Testing



- Robustness testing technique: test beyond the limits of normal operation.
- Can apply at any level of system granularity.
- Stress tests commonly put a greater emphasis on robustness, availability, and error handling under a heavy load, than on what would be considered “correct” behavior under normal circumstances.

Soak testing

- **Problem:** A system may behave exactly as expected under artificially limited execution conditions.
  - E.g., Memory leaks may take longer to lead to failure
- **Soak testing:** testing a system with a significant load over a significant period of time
- Used to check reaction of a subject under test under a possible simulated environment for a given duration and for a given threshold.

Testing purposes - 1

Technique	Description
Baseline testing	• Execute a single transaction as a single virtual user for a set period of time or for a set number of transaction iterations • Carried out without other activities under otherwise normal conditions • Establish a point of comparison for further test runs
Load testing	• Test application with target maximum load but typically no further • Test performance targets (i.e. response time, throughput, etc.) • Approximation of expected peak application use
Scalability testing	• Test application with increasing load • Scaling should not require new system or software redesign

Testing purposes - 2

Technique	Description
Soak (stability) testing	• Supply load to application continuously for a period of time • Identify problems that appear over extended period of time, for example a memory leak
Spike testing	• Test system with high load for short duration • Verify system stability during a burst of concurrent user and/or system activity to varying degrees of load over varying time periods
Stress testing	• Overwhelm system resources • Ensure the system fails and recovers gracefully