Design Patterns and Architectural Styles

Thomas Fritz

Many thanks to Reid Holmes and Elisa Baniassad

Agenda

1. Design Patterns
2. Class Exercise
3. Architectural Styles
4. Code Exercise – Design Patterns [optional]
5. Quiz and more

Note:

- *Next week:*
  - *no in person lecture, but content and podcast (Code Smells & Refactoring) are online and will be part of the exam material*
  - *no quiz!*

Examinable skills

By the end of this lecture you should be able to…

- Explain what a design pattern and an architectural style is and why they are useful
- Justify and discuss the use of various design patterns, their participants, benefits, limitations and differences
- Apply the presented (class, videos, readings) design patterns in a given scenario
- Describe characteristics of selected architectural styles, their advantages and disadvantages and how they could apply to a problem domain
- Explain MV* and discuss their differences, benefits and limitations

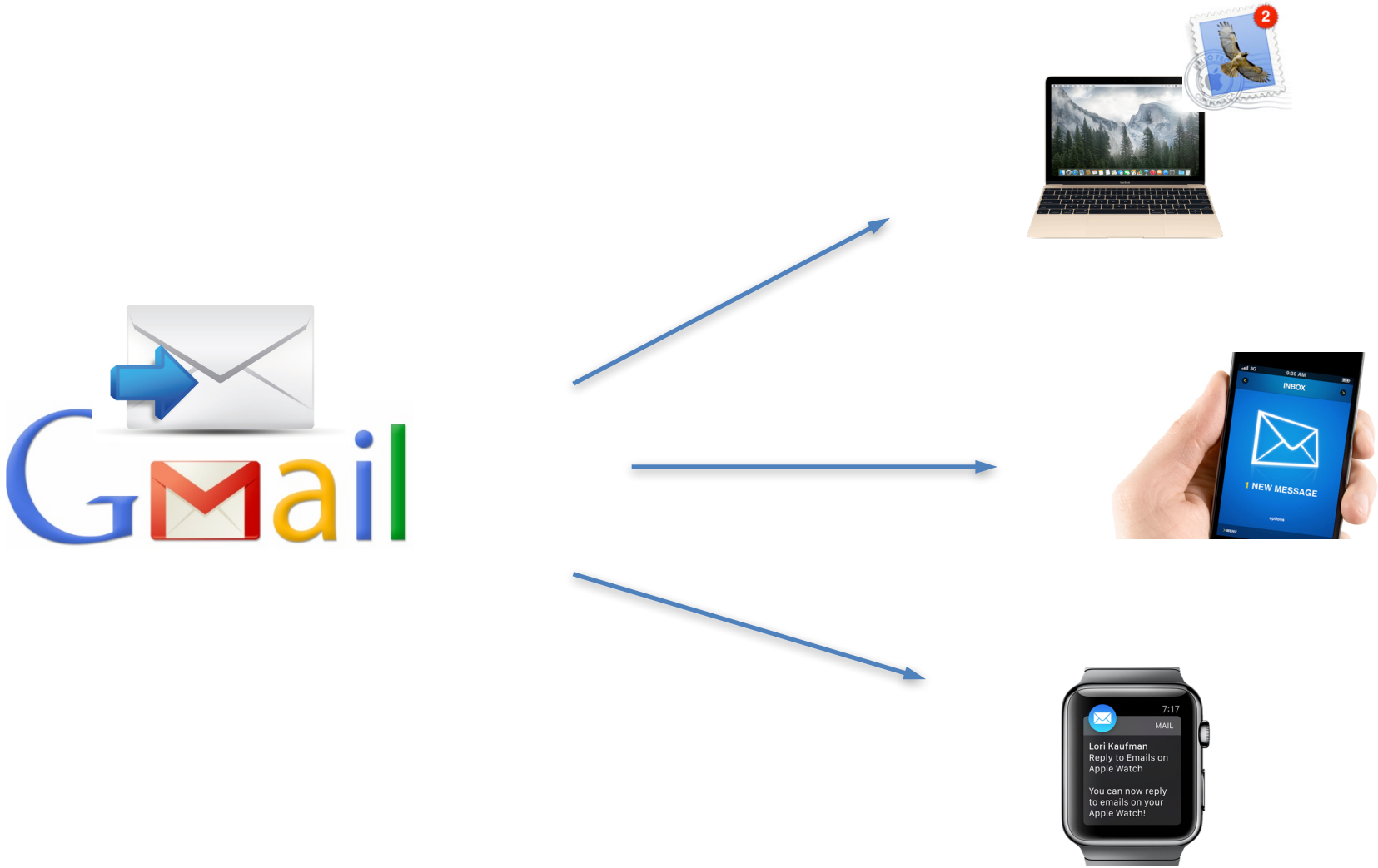
Design Patterns

Learning Objectives

Be able to:

- Explain what a design pattern is and why they are useful
- Justify, apply, and discuss the use of various design patterns, their participants, benefits, limitations and differences in given scenarios

Example – email notification



Email notification – possible design

GMailInboxIMAPFolder

+ appendMessages(Message[] msgs) : void
+ inboxChanged ( ) : void
+ getUnreadMessageCount( ) : int
+ getUnreadMessages( ) : Message[]

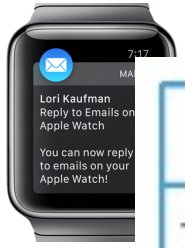
// other methods

MacBookMailApp

+ update (Message[] msgs) : void
// other methods

IPhoneMailApp

+ update (Message[] msgs) : void
// other methods



AppleWatchMailApp

+ update (Message[] msgs) : void
// other methods

Email notification – possible implementation

```
public class GMailInboxIMAPFolder {

	// ...

	public void inboxChanged() {
		macBookMail.update(getUnreadMessages());
		iPhoneMail.update(getUnreadMessages());
		appleWatchMail.update(getUnreadMessages());
	}

	// other methods
}
```

- What's not good about this implementation?

Modular Design – Recap

Eliminate effects between unrelated things and design components that are self-contained, independent, and have a single well-defined purpose.

- Designing software with good modularity is hard!

- Apply design principles
  - high cohesion, low coupling, information hiding
  - SOLID

- **Apply design patterns**

What is a design pattern?

A tried and true solution, to a commonly encountered problem

Think about what it feels like to solve a problem for the very first time...

You need to think about the problem from scratch

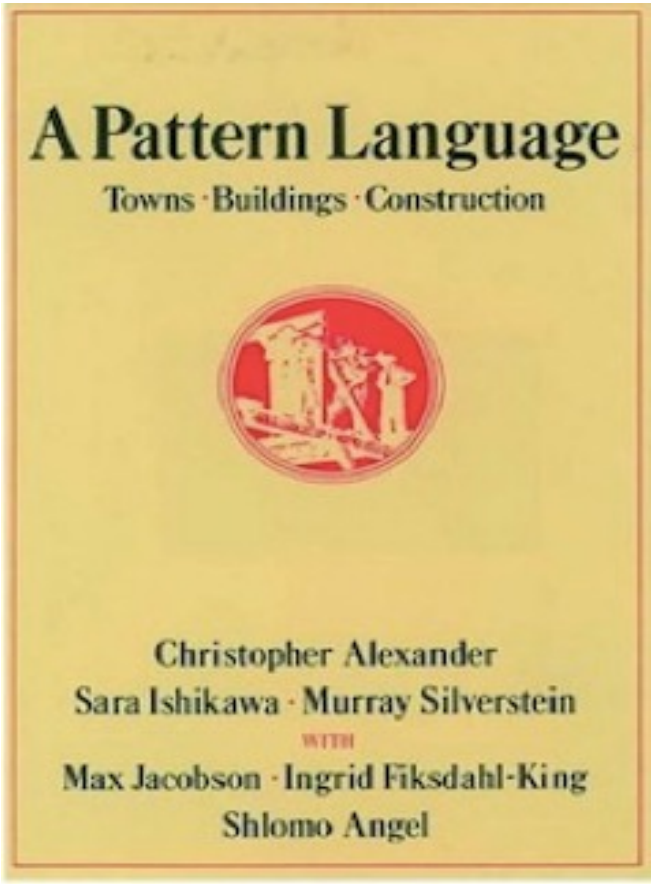
You have to test very thoroughly, and maybe won't even think of all possible cases to test!

- Now think about what it's like when someone tells you a possible solution
- That solution includes a lot of knowledge, experimentation, and testing!
- This saves a *lot* of time!!!

The “design pattern” name

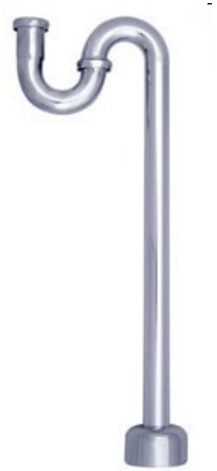
- original use comes from (building) architecture from Christopher Alexander

- was used for architectural idioms, to guide architectural design (a house is composed of a kitchen, bathroom, bedrooms etc... to be placed in certain basic configurations)



Real World Pattern Examples

Problem/ intent	sink stinks	highway crossing
Solution Structure	S-trap	clover leaf



Design Patterns in Software Engineering

A design pattern is a general **repeatable solution to a commonly occurring design problem**.

- *template* for how to solve a problem in a good modular way
- Distills/leverages design expertise
- names design structure explicitly (components, dependencies, interactions etc) and eases communication (shared vocabulary)

- **not a complete/finished design**

Design Patterns in SE

Main components

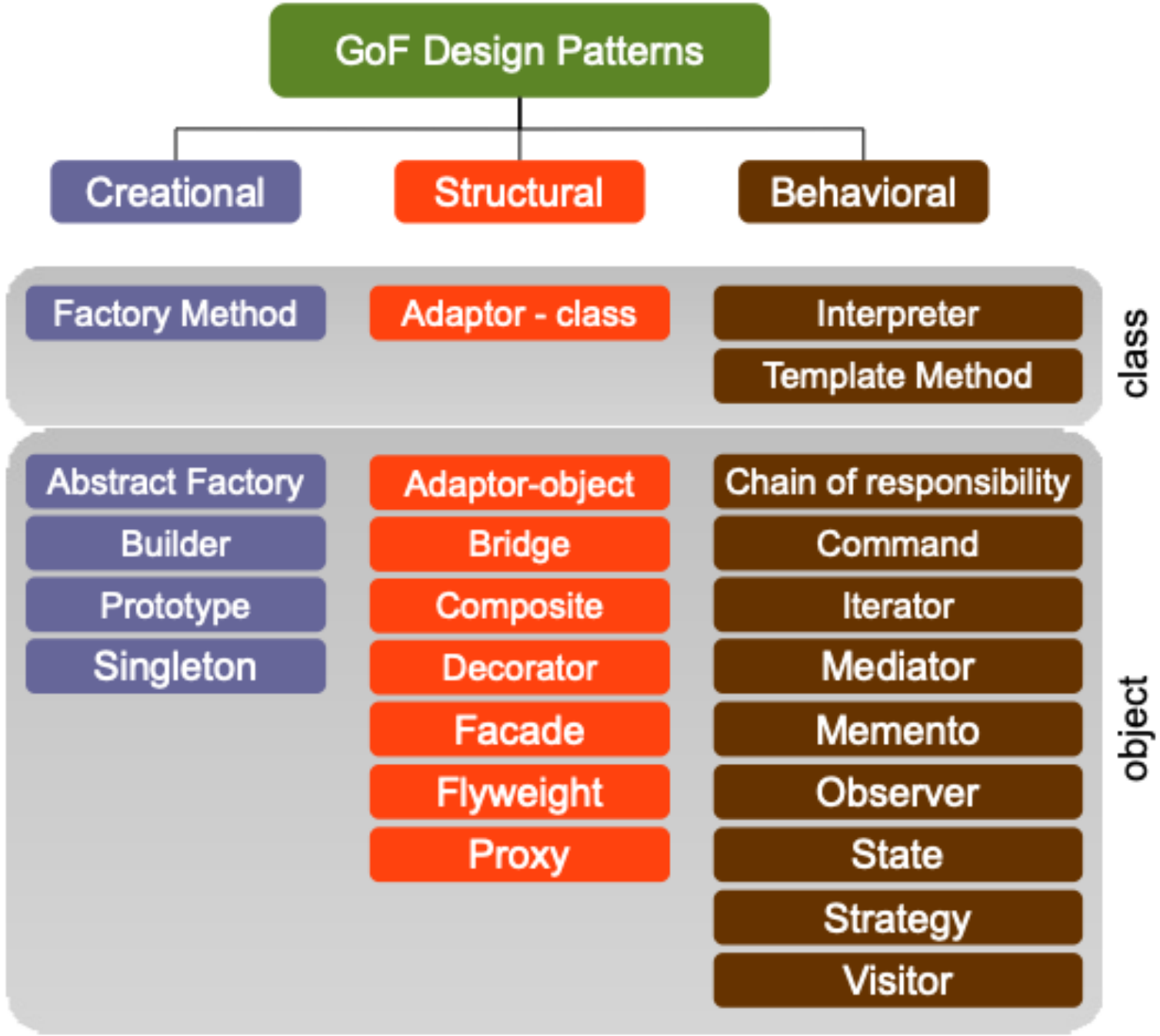
- Name
- Intent: what problem does it solve
- Solution: Participants & structure
- Consequences: known uses, related patterns, pros & cons

Are language-independent

Cannot be mechanically applied

- must be translated/transferred to a context by developer

Design Patterns



Design Pattern Classification

By purpose

- **creational patterns** concern the process of object creation
- **structural patterns** deal with composition of classes or objects
- **behavioural patterns** characterise the ways in which classes or objects interact and distribute responsibility

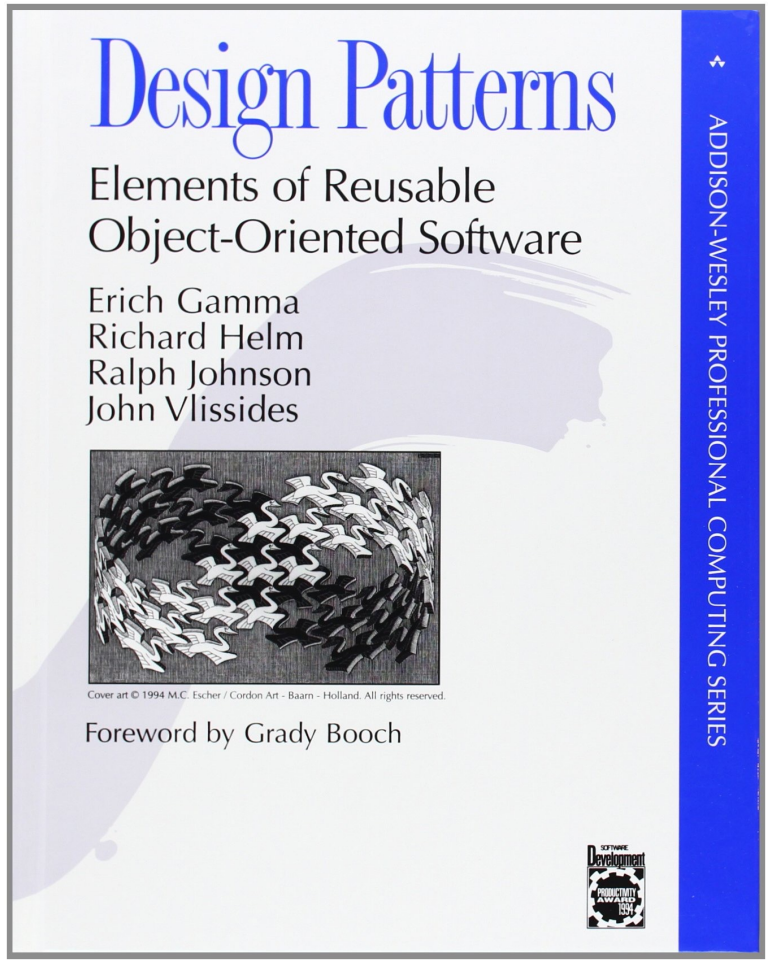
By scope

- **class:** deal with relationships between classes and their subclasses
- **object:** deal with object relationships that can be changed at run-time and are more dynamic

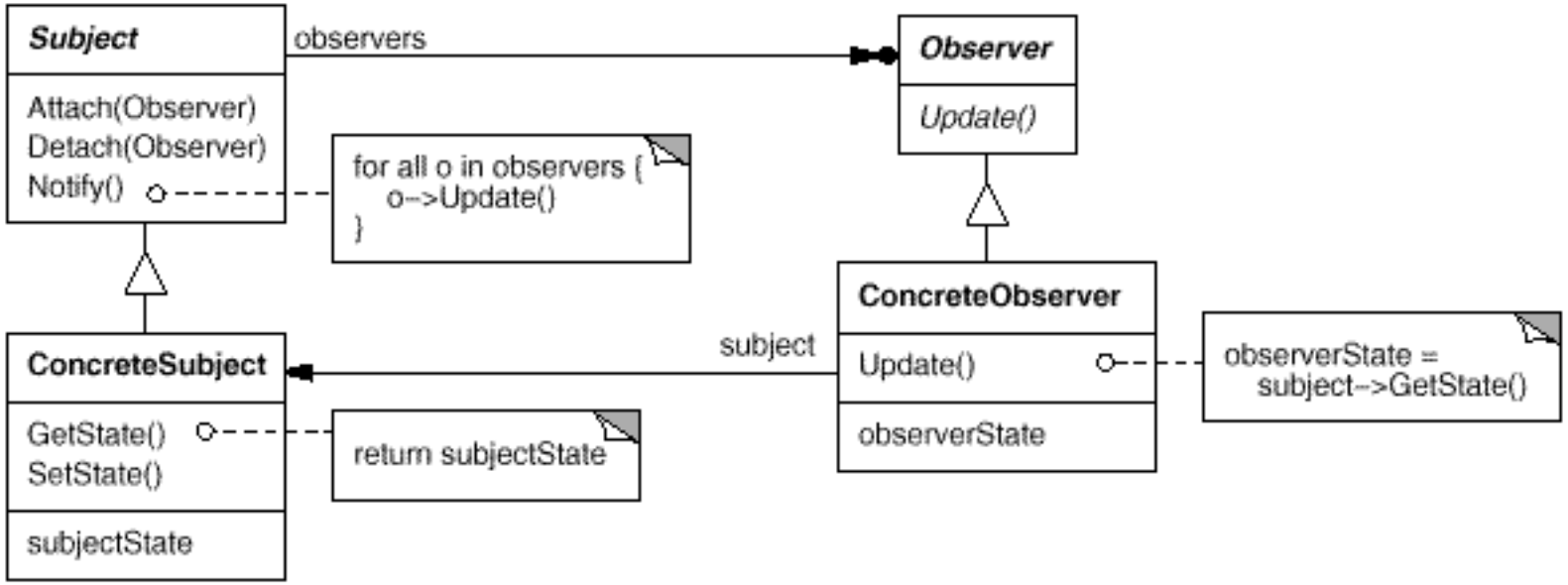
Gang of Four (GoF) Book

compilation of object-oriented, low-level software design patterns

published in 1994 by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.



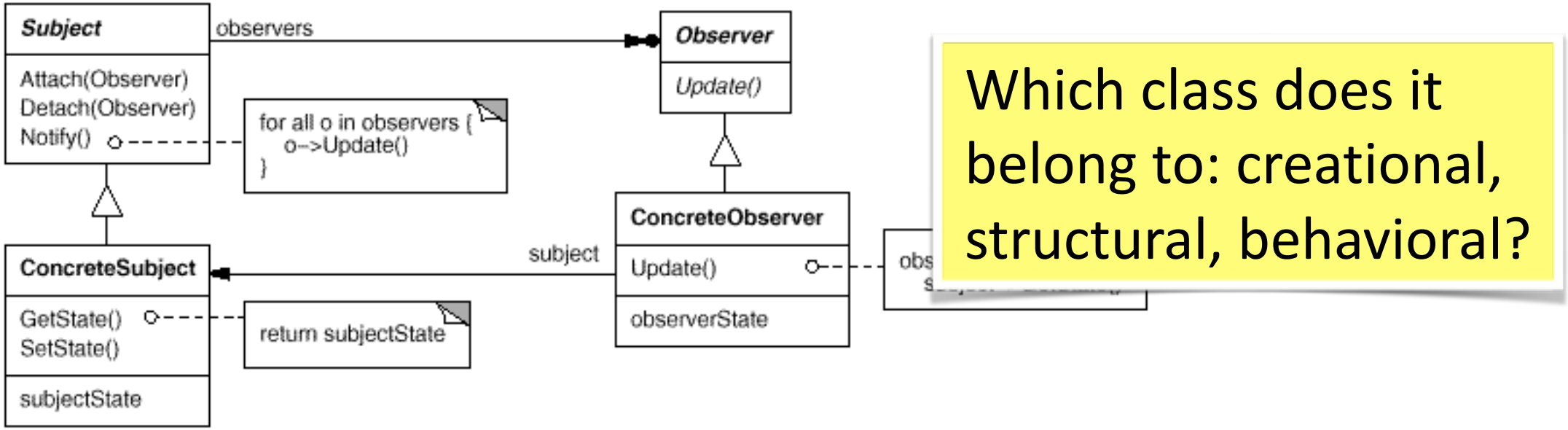
The Observer Design Pattern



*The **Observer Pattern** defines a one-to-many dependency between objects so that when one object (called the subject) changes state, all of its dependents (called observers) are notified and updated automatically.*

– Head First Design Patterns

The Observer Design Pattern – Question



*The **Observer Pattern** defines a one-to-many dependency between objects so that when one object (called the subject) changes state, all of its dependents (called observers) are notified and updated automatically.*

– Head First Design Patterns

Steps in applying a design pattern

Know pattern, recognize problem, apply & integrate

- Know the problems common design patterns can solve and identify which pattern fits for the given problem

- What are the critical roles in the Design Pattern?
- How are these roles mapped to classes in your scenario (possibly also add new ones)?
- How will the right methods be triggered (e.g. registering observers for subjects in observer design pattern)?

Revisiting common problem: Email notification

GMailInboxIMAPFolder

+ appendMessages(Message[] msgs) : void
+ inboxChanged ( ) : void
+ getUnreadMessageCount( ) : int
+ getUnreadMessages( ) : Message[]

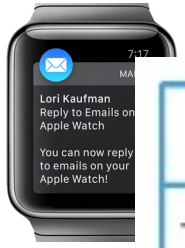
// other methods

MacBookMailApp

+ update (Message[] msgs) : void
// other methods

IPhoneMailApp

+ update (Message[] msgs) : void
// other methods



AppleWatchMailApp

+ update (Message[] msgs) : void
// other methods

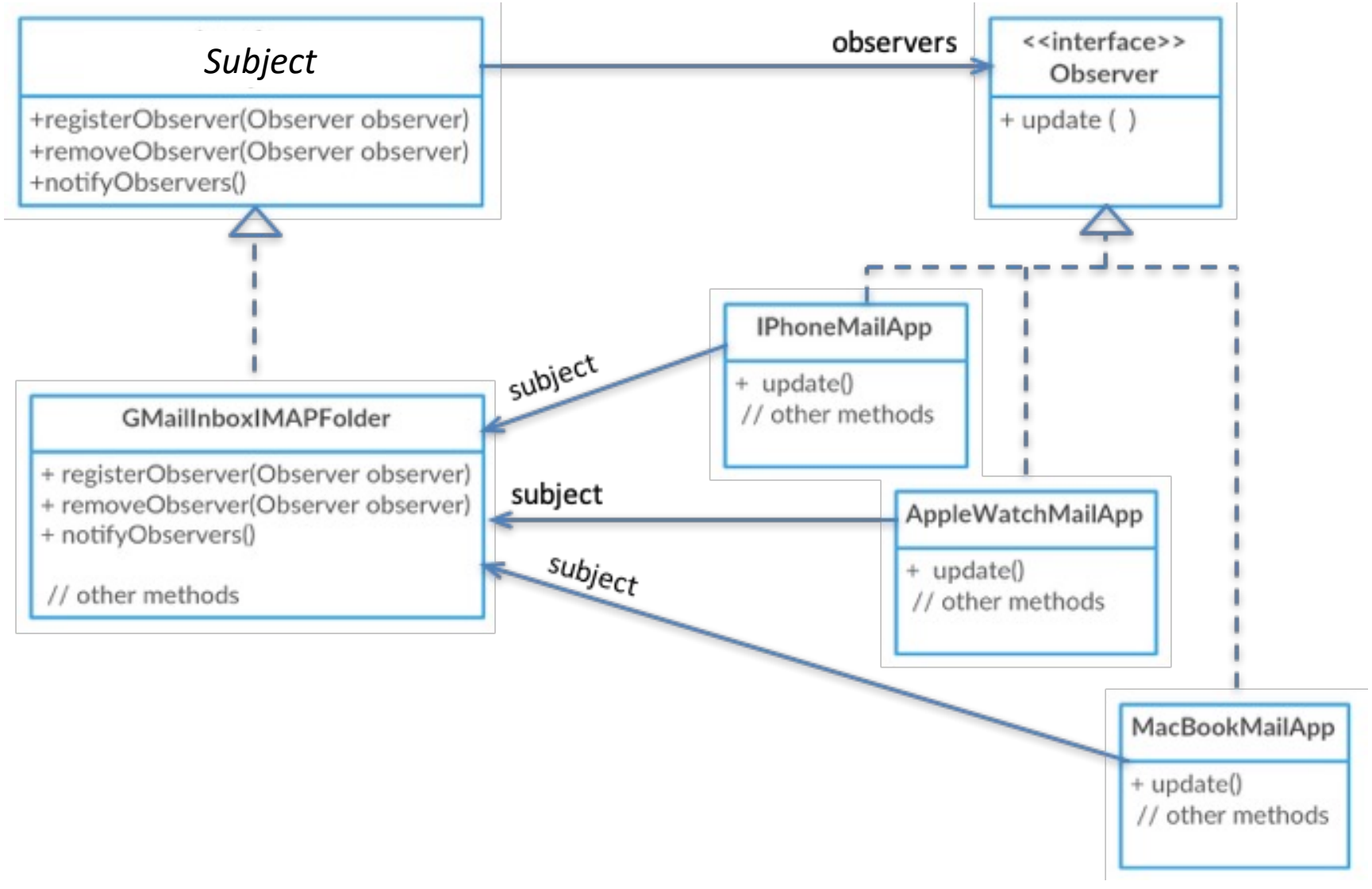
Exercise – Apply Observer DP (~10 mins) [https://bit.ly/3xLWy5H]

In groups, apply the Observer Design Pattern to the Email Notification scenario and sketch out the classes, methods and associations needed (UML class diagram)

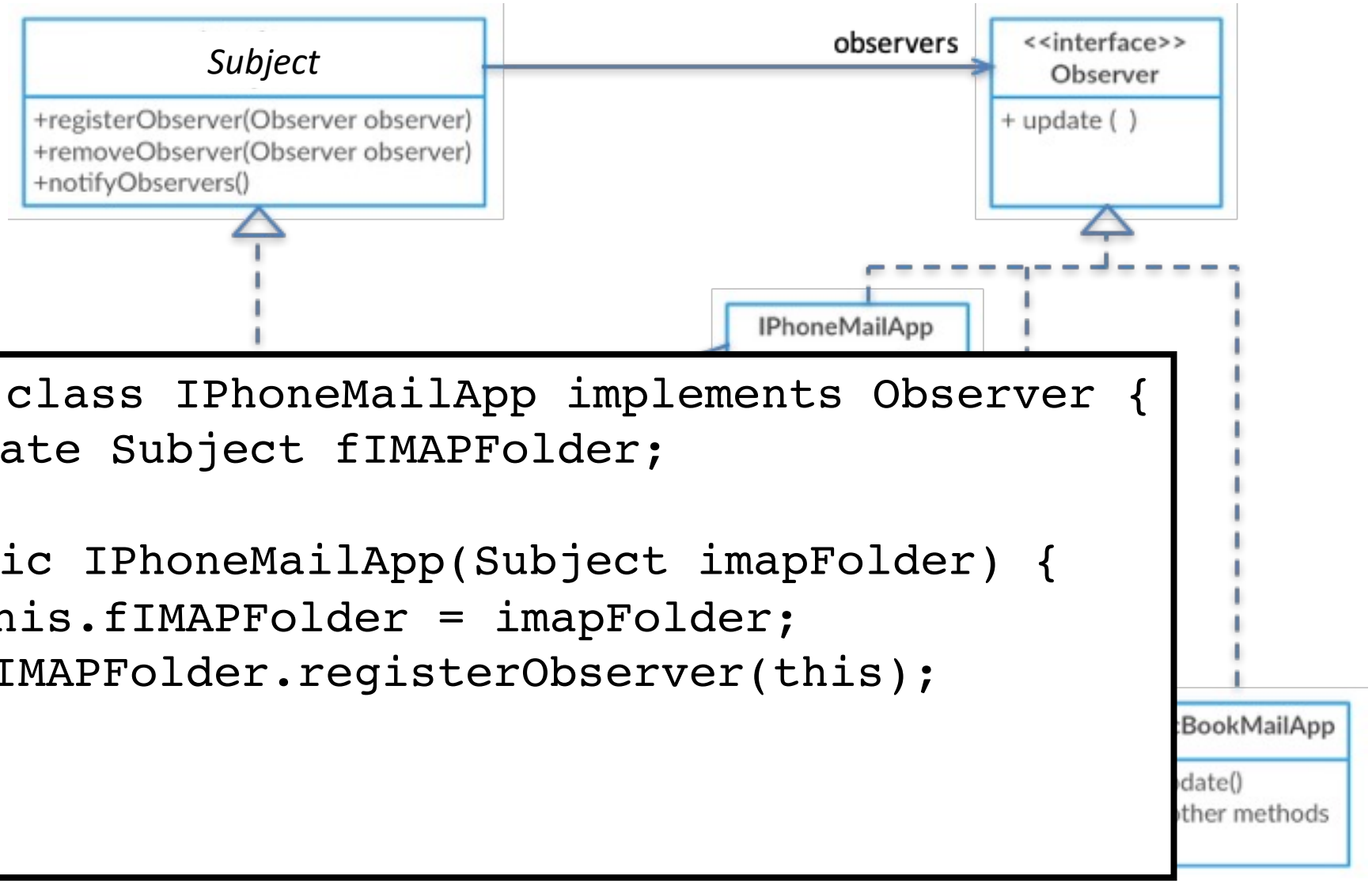
Remember the classes of the scenario:

- GMailInboxIMAPFolder
- MacBookMailApp
- IPhoneMailApp
- AppleWatchMailApp

Email notification – Observer DP



Email notification – Observer DP




```
public class IPhoneMailApp implements Observer {
   private Subject fIMAPFolder;

   public IPhoneMailApp(Subject imapFolder) {
      this.fIMAPFolder = imapFolder;
      fIMAPFolder.registerObserver(this);
   }
   ...
}
```

Highlights – Observer DP

+ Loose coupling between subjects and observers
  - subject only knows observer interface
  - new observers can be added at any time
  - changes to either will not affect the other
- High notification cost, if there are many observers
+ Different notification mechanisms can be used, i.e. push or pull

Observer DP – further scenarios

In an online auction, I want bidders to be notified when the auctioneer accepts a new bid

I want different views on weather data to be updated when the data changes

Employed by Model-View-Controller & MVP

Design Patterns Recap

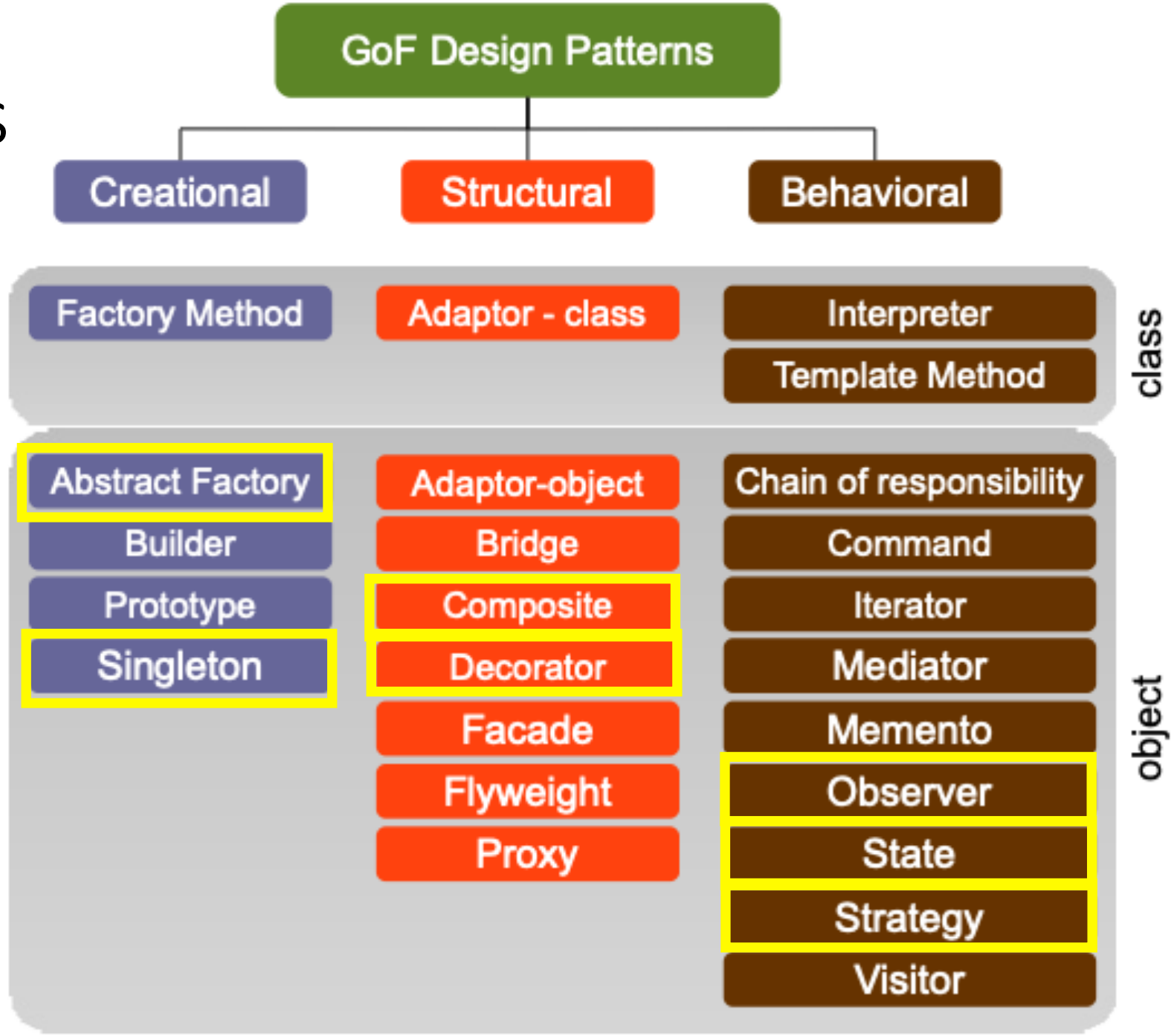
[mostly reference, just some examples in more detail]

Learning Objectives

Be able to:

- Justify, apply, and discuss the use of various design patterns, their participants, benefits, limitations and differences in given scenarios

Design Patterns

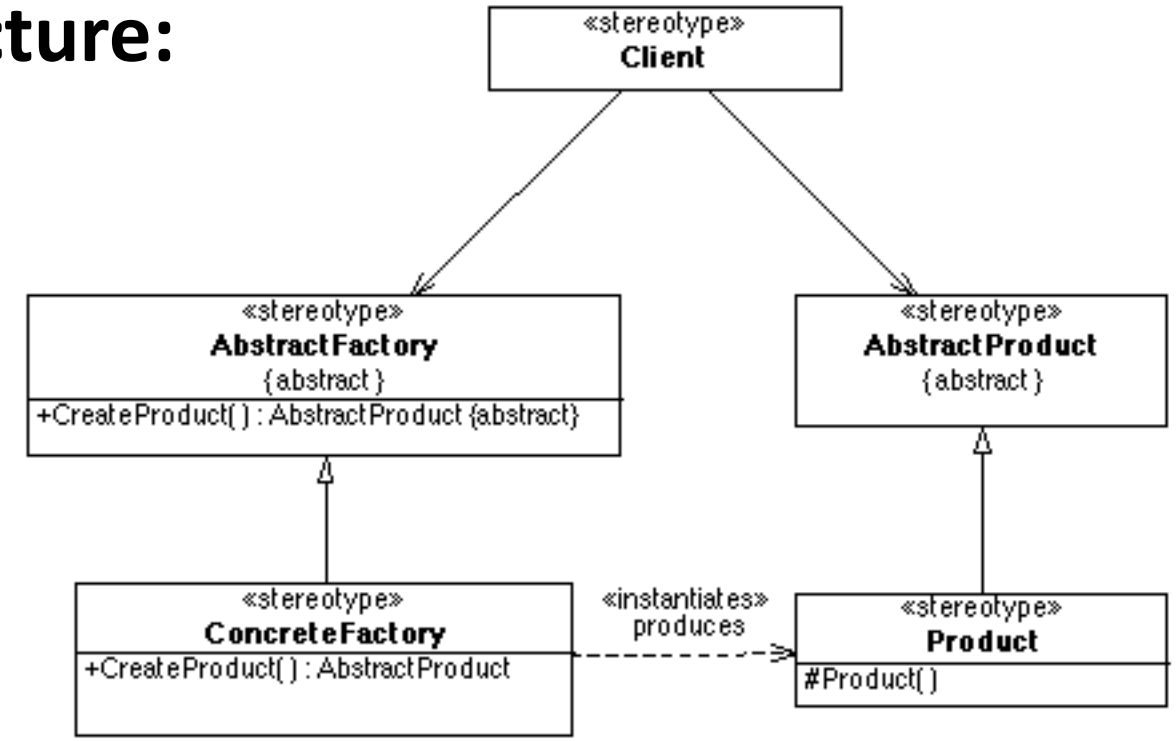


Abstract Factory DP

Name: Abstract Factory

Intent: Interface for creating families of related or dependent objects without specifying their concrete class

Participants & Structure:

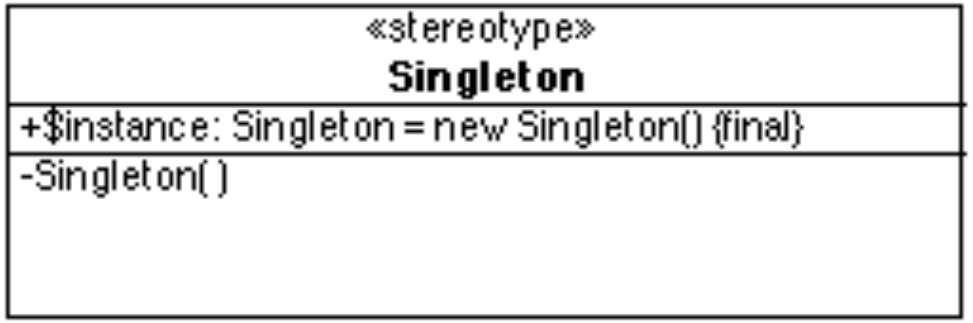


Singleton Design Pattern

Name: Singleton

Intent: Make sure a class has a single point of access and is globally accessible (*i.e. Filesystem, Display, PreferenceManager…*)

Participants & Structure:




```
private static Singleton uniqueInstance = null;

public static Singleton getInstance() {
        if (uniqueInstance == null)
                uniqueInstance = new Singleton();
        return uniqueInstance;
}

// Make sure constructor is private!
private Singleton() {…}
```

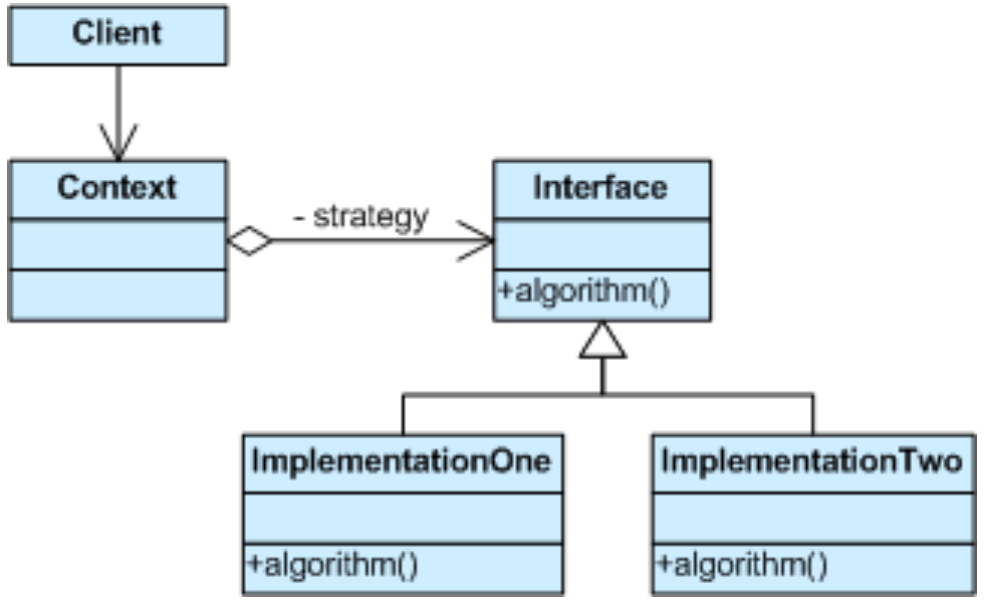
Strategy design pattern

Intent: Define a family of algorithms that can be easily interchanged with each other.

Motivation: support open/closed principle by abstracting algorithms behind an interface; clients use the interface while subclasses provide the functionality

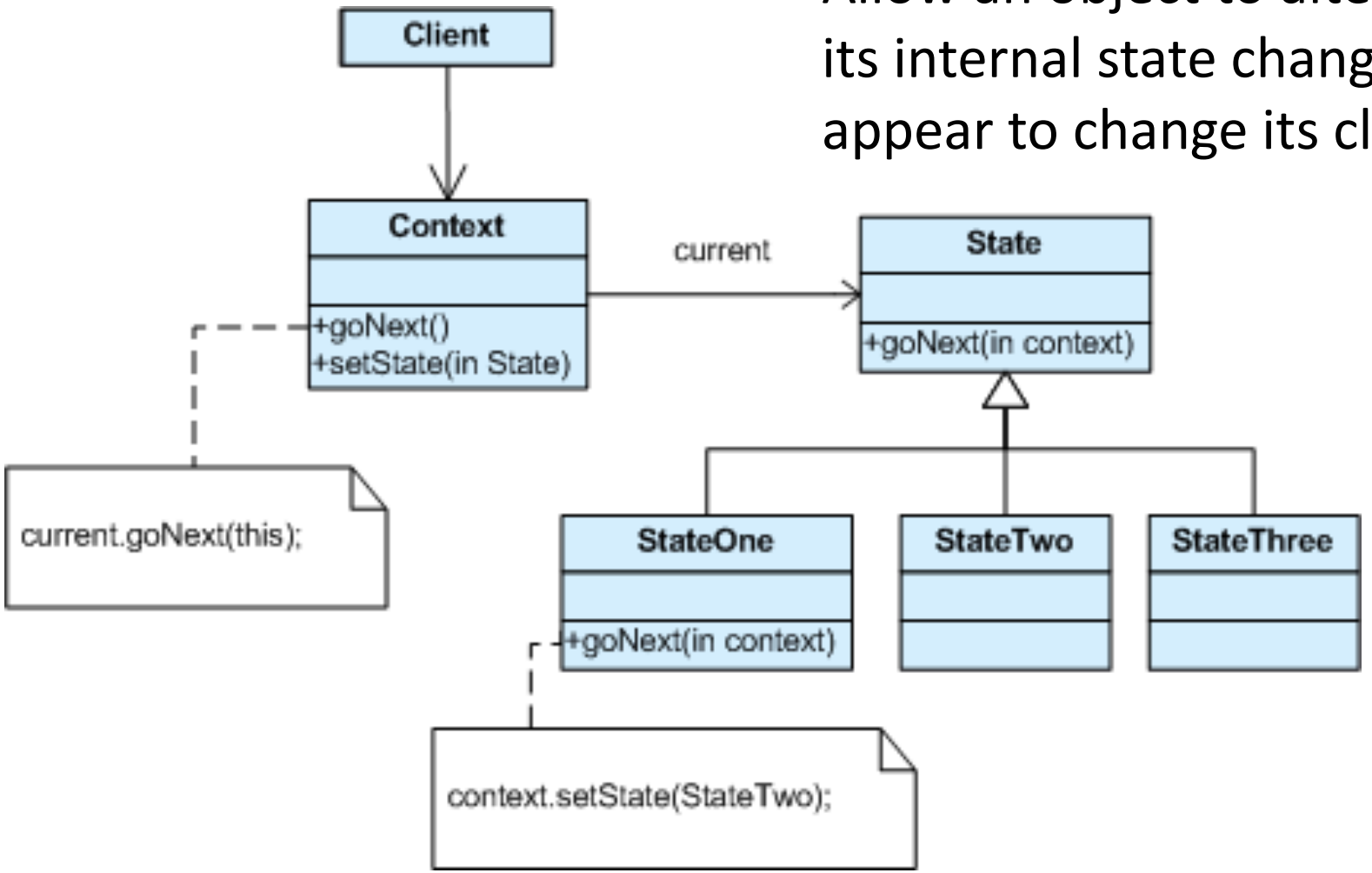
Applicability: when you want to replace behavior at runtime;

Participants & Structure:

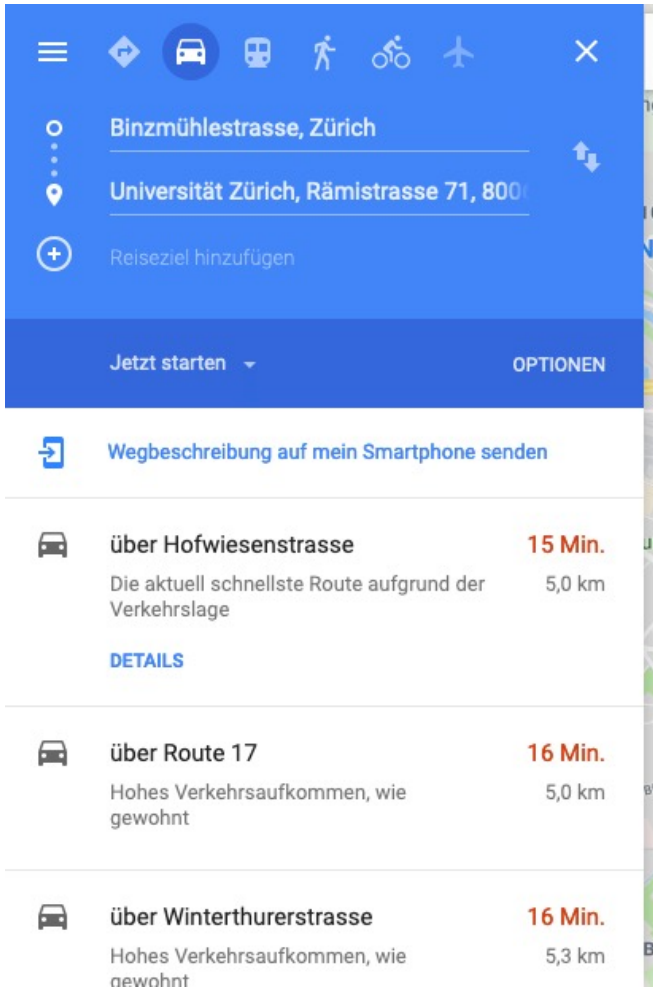


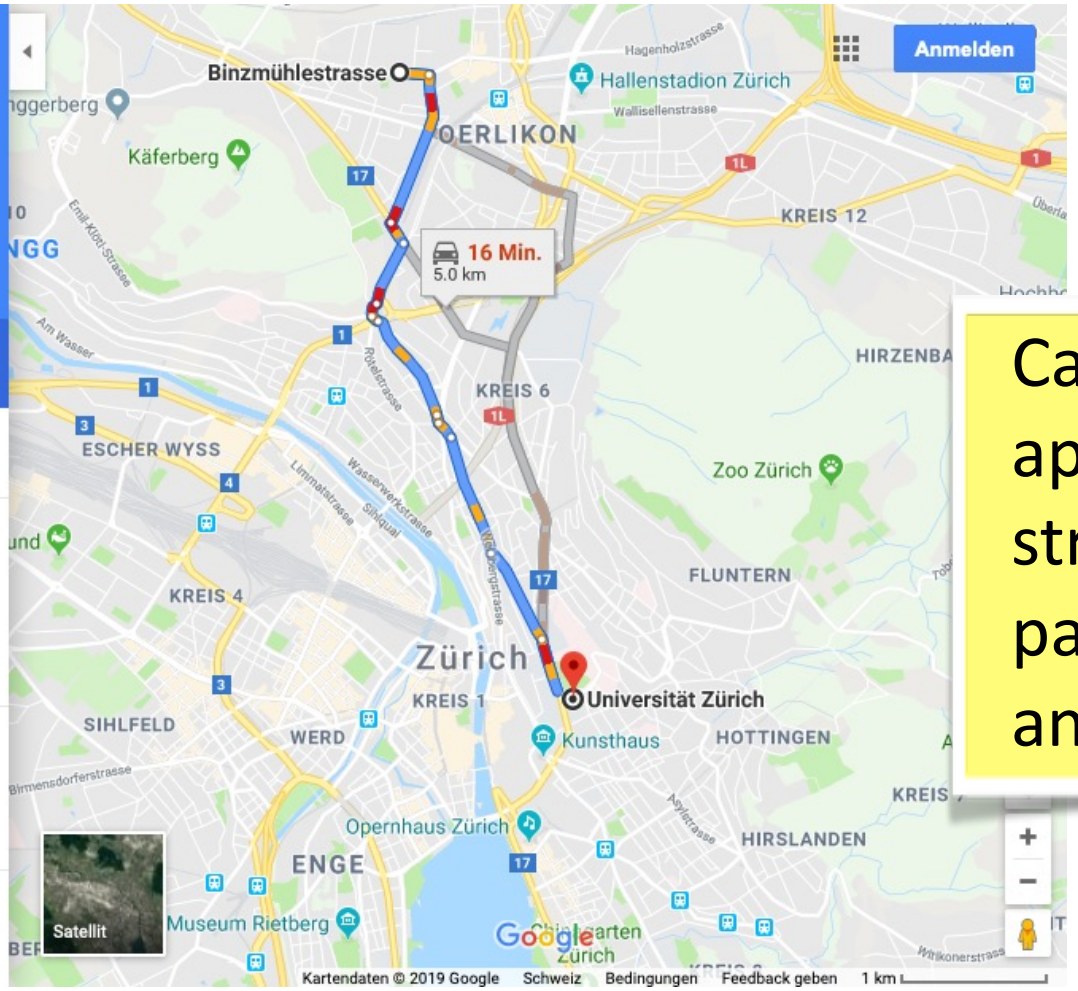
State design pattern

Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.



Design pattern example – Question





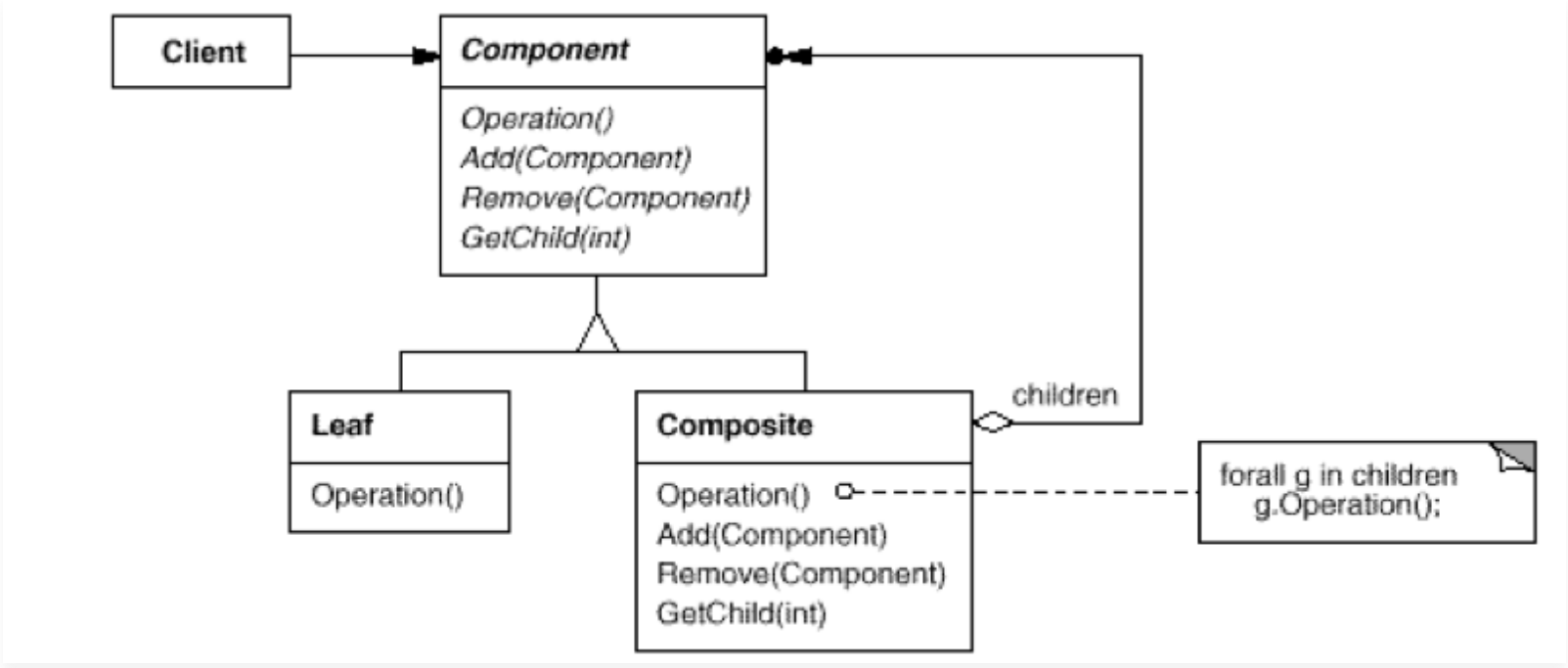
Can you spot a potential application of either the strategy or the state pattern: which of the two and where?

Composite design pattern

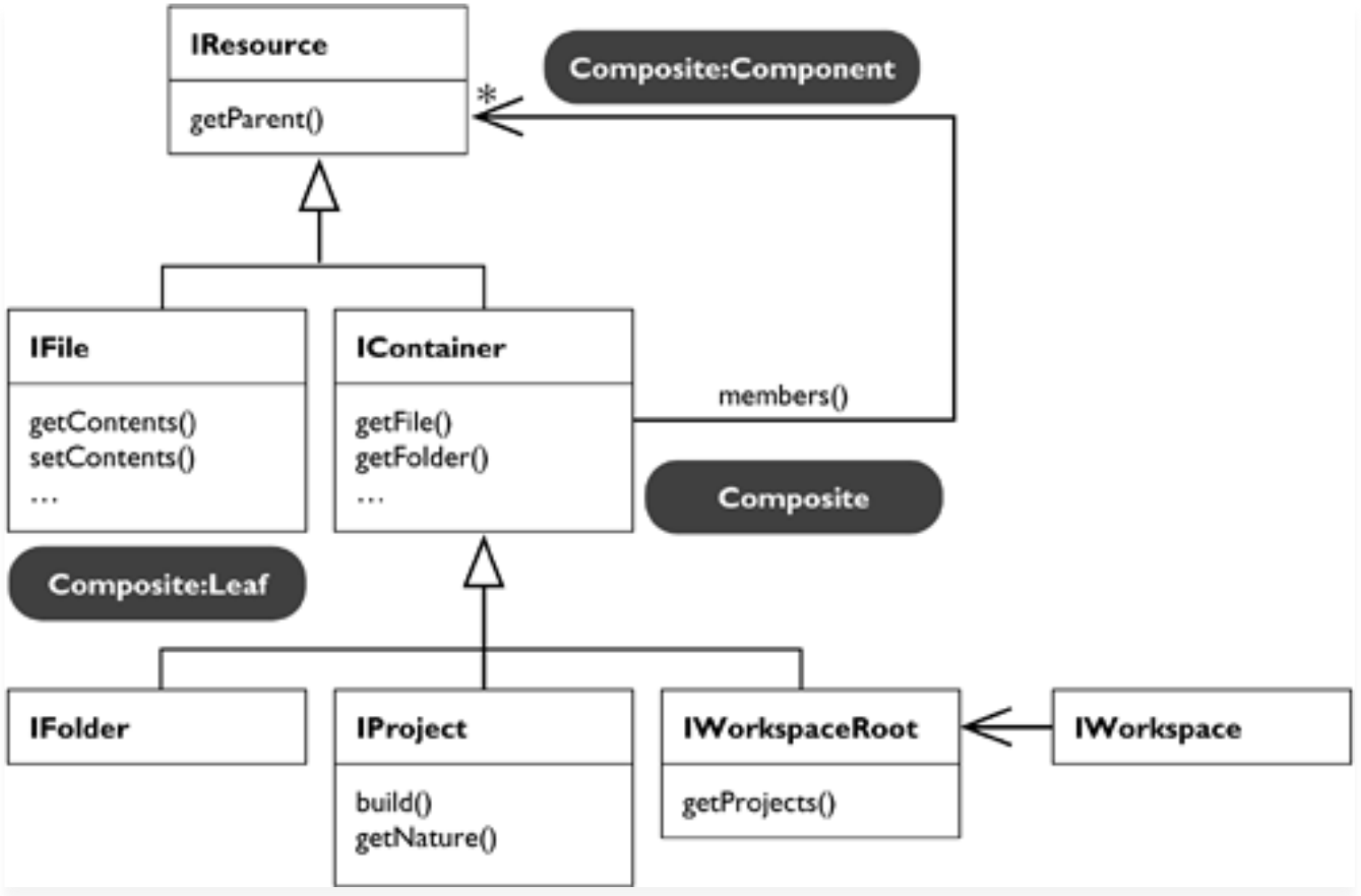
Name: Composite

Intent: Compose objects into tree structures. Let's clients treat individual objects and compositions uniformly.

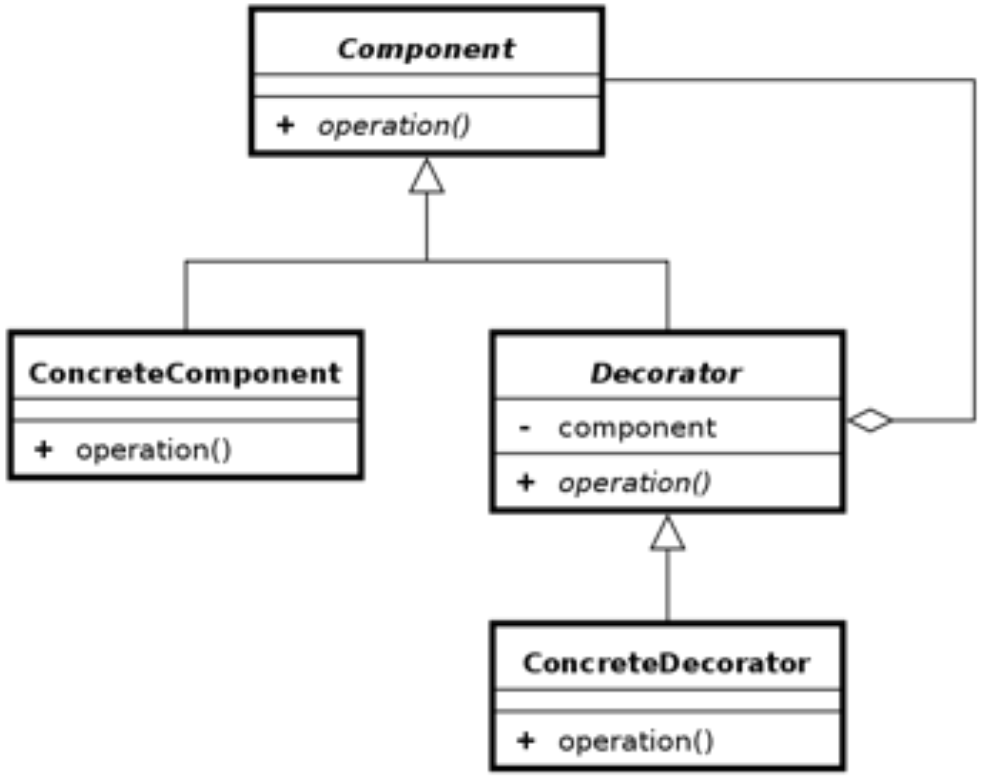
Participants & Structure:



Composite – Example



Decorator design pattern



*The **Decorator Pattern** attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.*

– Head First Design Patterns

Decorator design pattern example

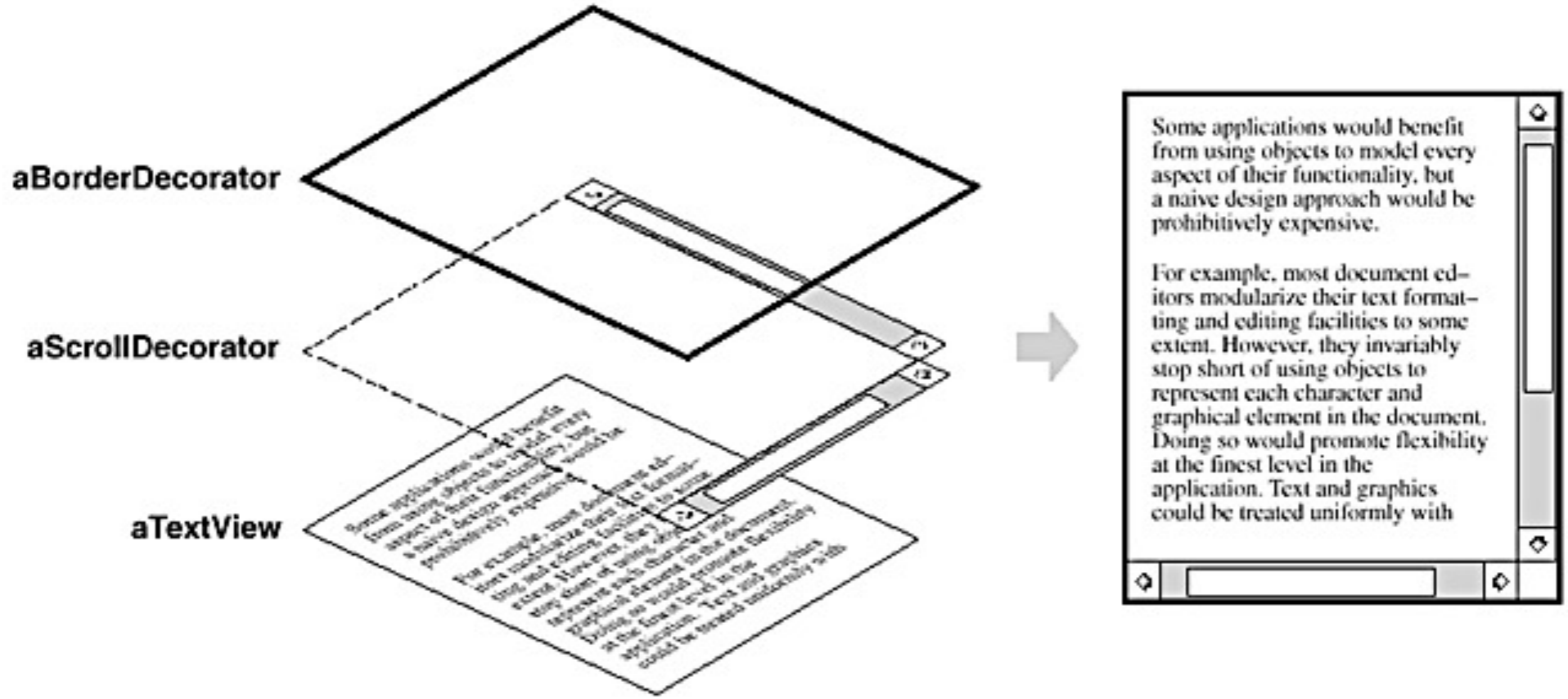
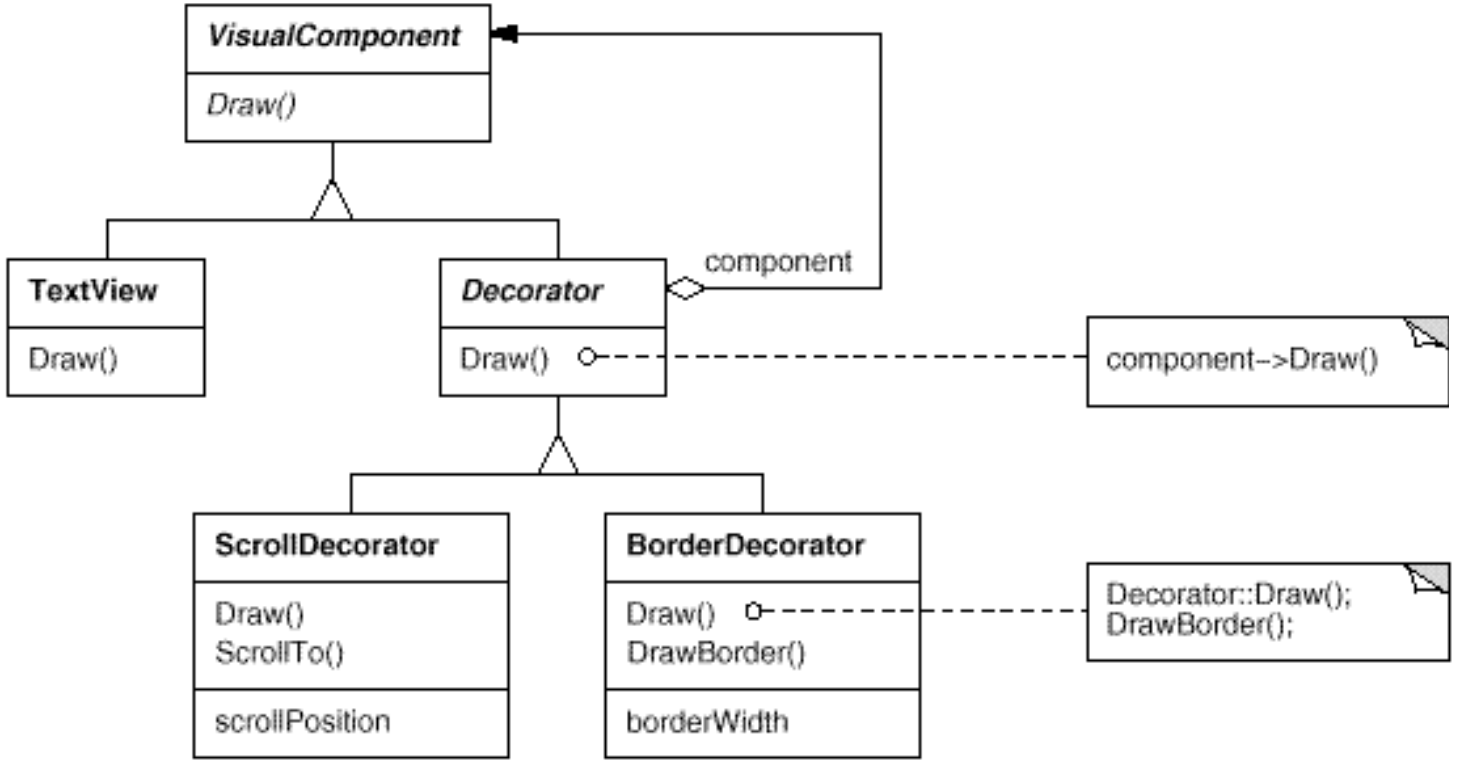


Diagram by Jan Klunder- https://commons.wikimedia.org/w/index.php?curid=1797712

Decorator design pattern example



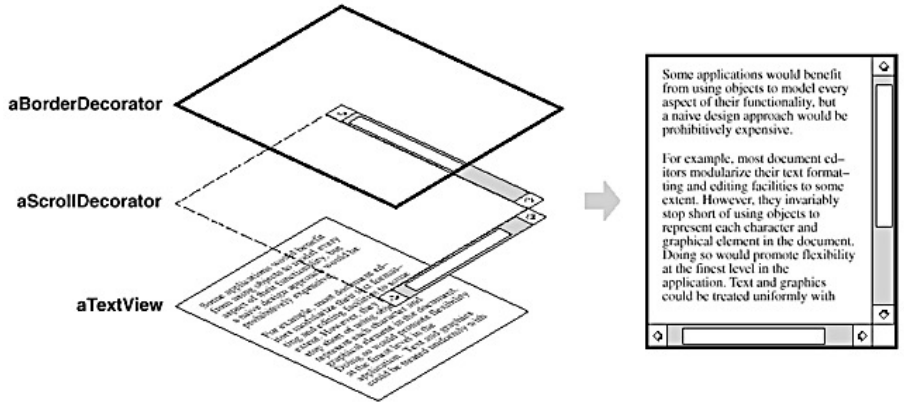
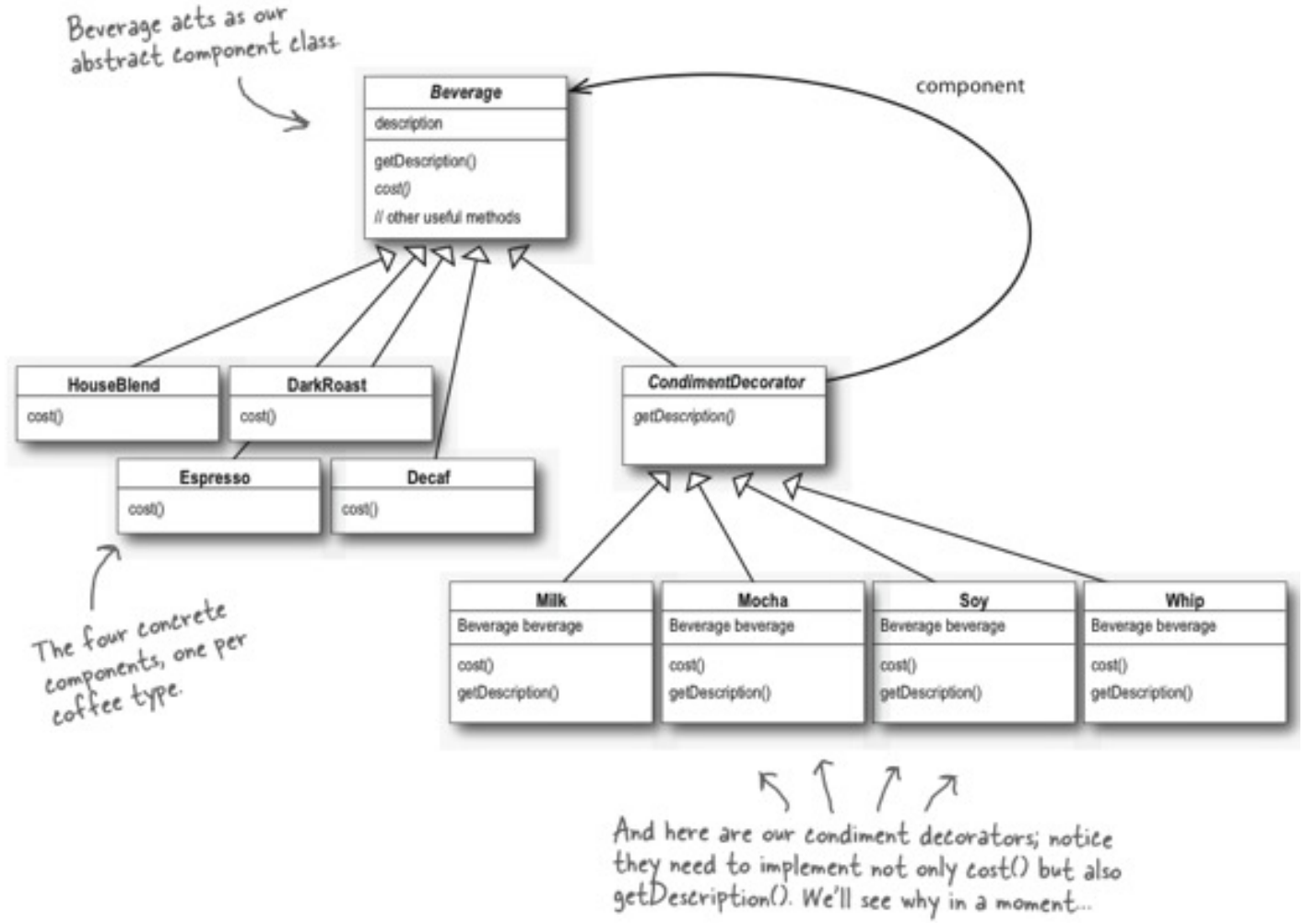


Diagram by Jan Klunder- https://commons.wikimedia.org/w/index.php?curid=1797712

Coffee ordering with the Decorator DP



Freeman, et al. *Head First Design Patterns*

Highlights – Decorator DP

+ Adheres to Open-Closed principle
  - Classes are open for extension but closed for modification
  - No need to modify a class for extending the behavior
+ Extends functionality of object without affecting any other object
- Can create lots of small classes, and overuse can be complex
- Increases complexity of code needed to instantiate a component

Questions & Design Pattern Exercise

Learning Objectives

Be able to:

- Justify, apply, and discuss the use of various design patterns, their participants, benefits, limitations and differences in given scenarios

Design Problem: Bags and Boxes [https://bit.ly/4aP4bXy] (~15mins)

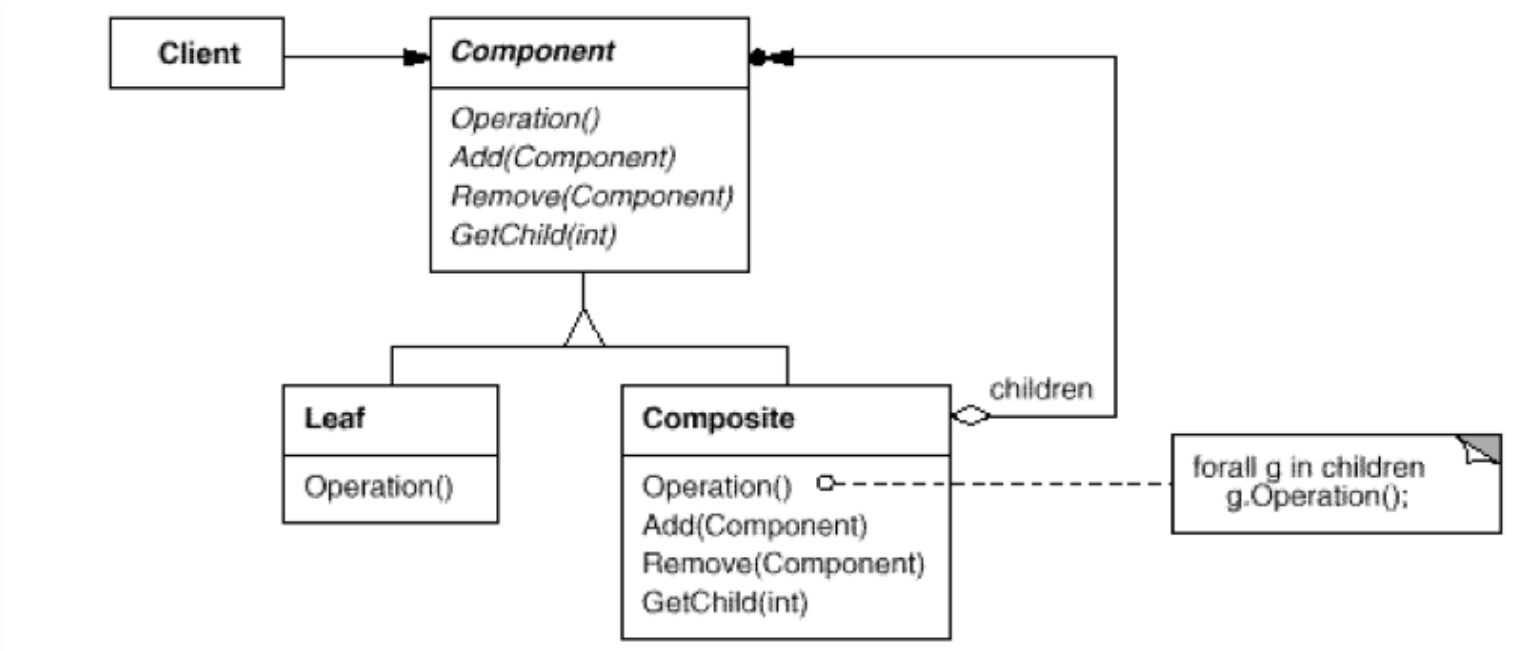
The designer of an adventure game wants a player to be able to take (and drop) various items found in the rooms of the game. Two of the items found in the game are bags and boxes. Both bags and boxes can contain individual items as well as other bags and boxes. Bags and boxes can be opened and closed and items can be added to or removed from a bag or box.

Choose a pattern and adapt it to this situation.

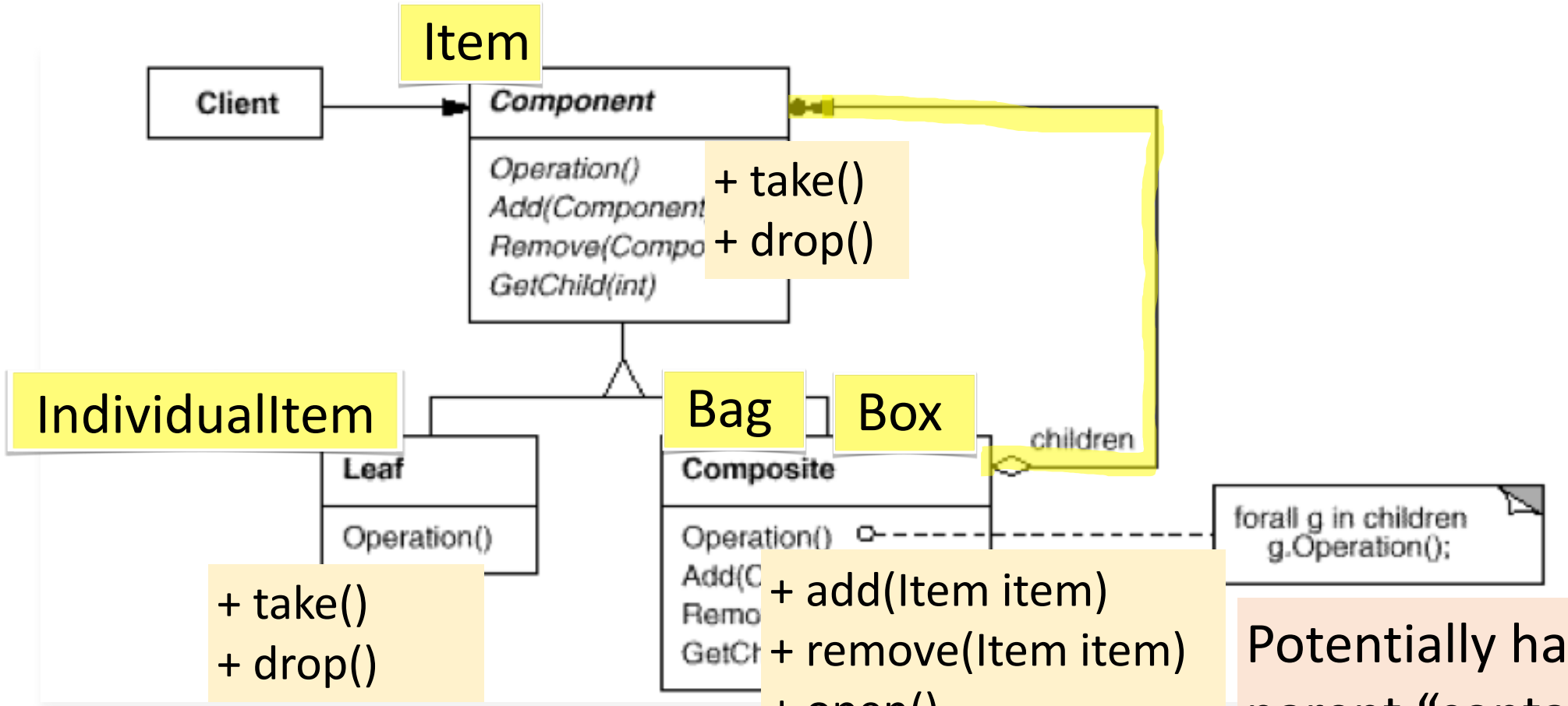
(What are advantages and disadvantages of this design?)

Design Problem: Bags and Boxes

The designer of an adventure game wants a player to be able to take (and drop) various items found in the rooms of the game. Two of the items found in the game are bags and boxes. Both bags and boxes can contain individual items as well as other bags and boxes. Bags and boxes can be opened and closed and items can be added to or removed from a bag or box.



Design Problem: Bags and Boxes



Discussion Question

Look-and-Feel:
a GUI framework should support several look and feel standards, such as Motif and Windows look, for its widgets. The widgets are the interaction elements of a user interface such as scroll bars, windows, boxes, buttons. Each style defines different looks and behaviors for each type of widget.

Which pattern is most applicable:

A. Observer

B. Decorator

C. Composite

D. Abstract Factory

Architectural Styles

Learning Objectives

Be able to:

- Describe what is meant by an architectural style and their purpose
- Describe characteristics of selected architectural styles
- Apply architectural styles to a problem domain and describe advantages and disadvantages

Software Architecture – Recap

- The set of principal design decisions about the system.

- Encompasses
  - structure, behaviour, non-functional properties

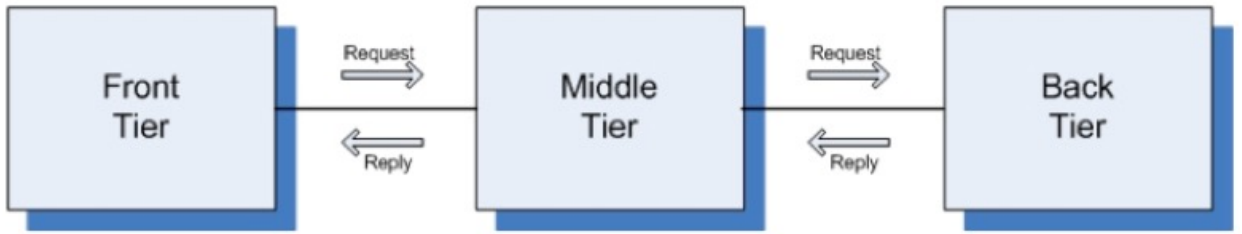
- Defines
  - components, connectors, configurations

[Tailor et al.]

Architectural styles

An architectural style is a named collection of “good” architectural design decisions applicable to a recurring design problem (based on experience)

- Parameterized to account for different contexts in which that problem appears
- constrains design decision
- elicits beneficial qualities in resulting systems
- e.g. three-tier architectural style/pattern



[Czarnecki]

Architectural styles

Defines a family of architectures that are constrained by

- Component/connector vocabulary; topology; semantic constraints

Diagrammatically

- Nodes == components (e.g. procedures, modules, processes,…)
- Edges == connectors (e.g. procedure calls, events, db queries,...)

Understanding a style

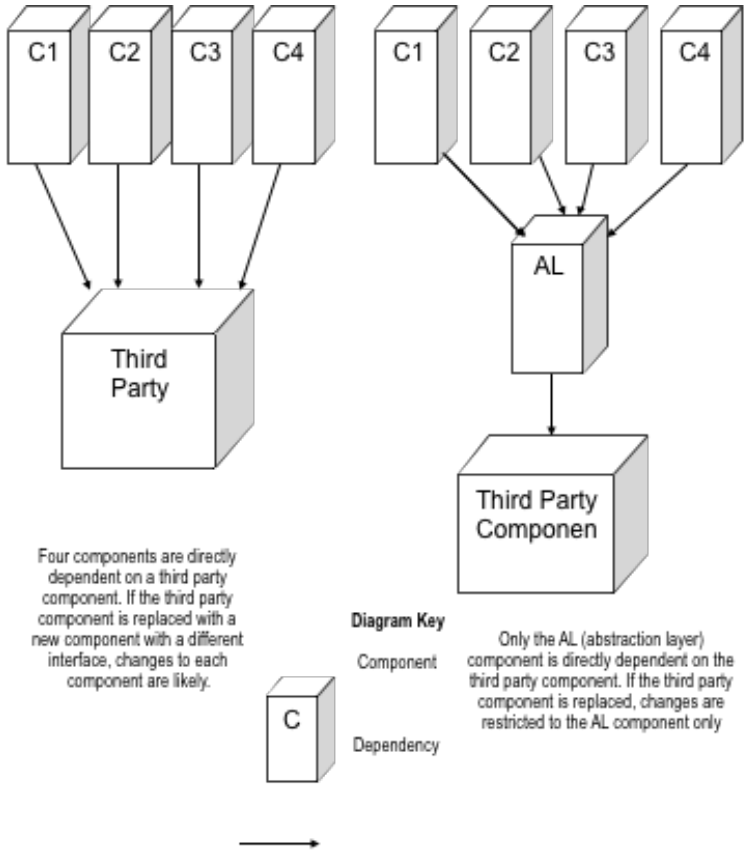
- What is the structural pattern? What is the underlying computational model? What are the essential invariants of the style? What are some common usage examples? What are the style's advantages and disadvantages?

[Gorton]

Structure and Dependencies

- All styles minimize coupling in a sepcific way

- Example diagram:
  - Identify likely change points
  - Reduce direct dependencies on these points



[Czarnecki]

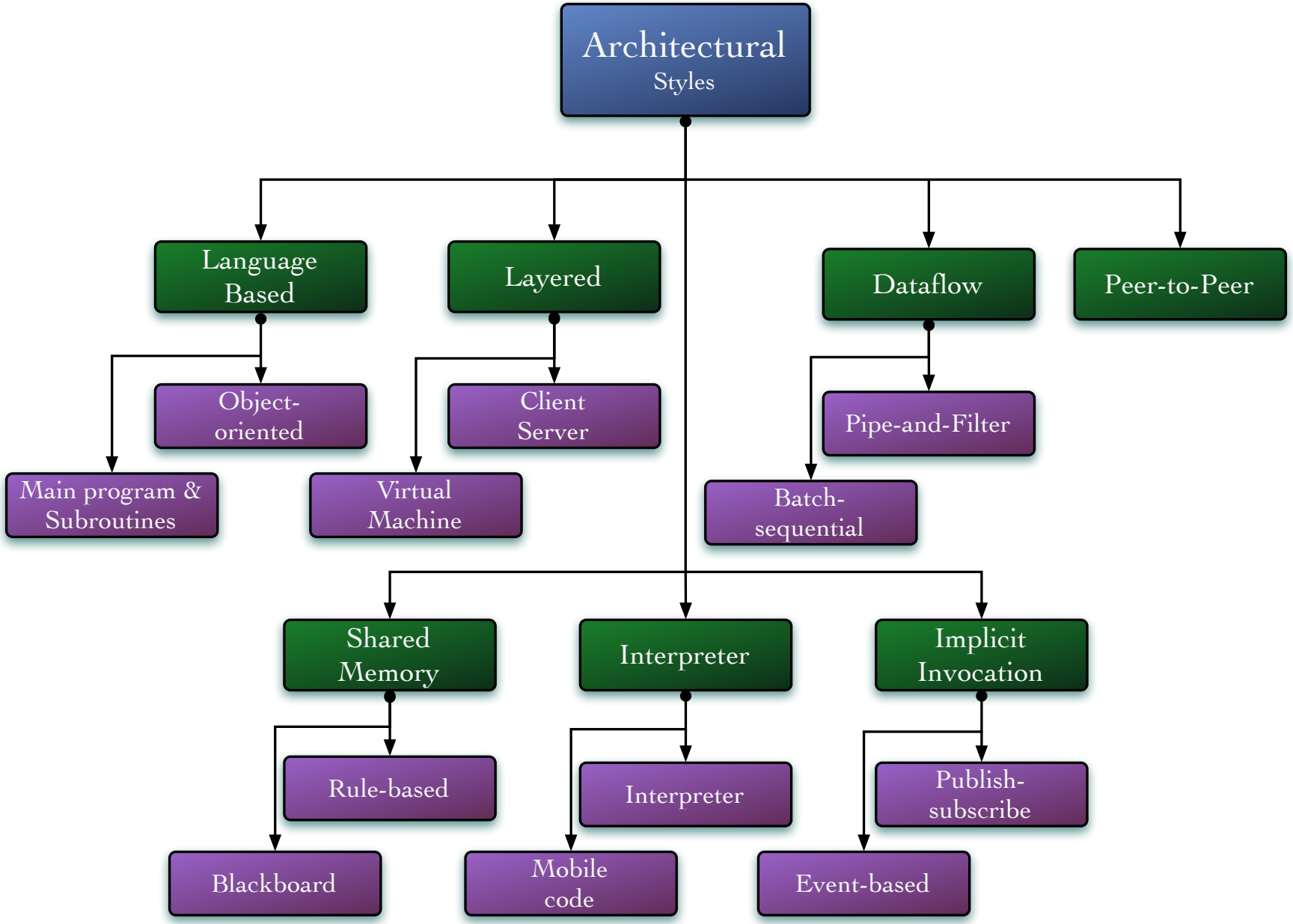
Good properties of an architectural style

- Result in a consistent set of principled techniques
- Resilient in the face of (inevitable) changes
- Source of guidance through product lifetime
- Reuse of established engineering knowledge

“Pure” architectural styles

- Pure architectural styles are rarely used in practice
- Systems in practice:
  - Regularly deviate from pure styles
  - Typically feature many architectural styles
- Architects must understand “pure” styles to understand strength and weaknesses of it as well as consequences of deviating from the style

[Topology from Tailor et al.]



Language-based

- Influenced by the languages that implement them
- Lower-level, very flexible
- Often combined with other styles for scalability

Examples:

Main & subroutine

Object-oriented

Data Abstraction and Object-Oriented Organization

Components:

- objects / instance of abstract data type

Connectors:

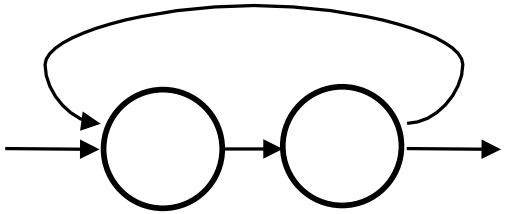
- Procedure/method calls

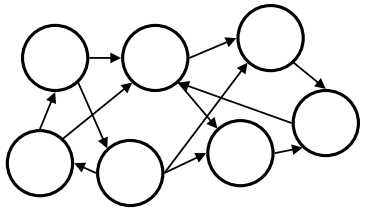
Constraints:

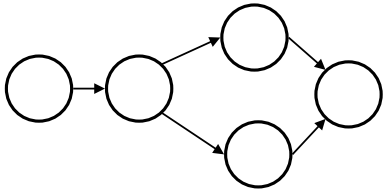
- object responsible for preserving integrity of its representation
- representation hidden from other objects

[Czarnecki]

Dataflow







- A data flow system is one in which
  - availability of data controls computation
  - design structure is determined by order motion of data between components
  - pattern of data flow is explicit

- Variations
  - push vs. pull
  - degree of concurrency

Examples:

Batch-sequential

Pipe-and-filter

Pipe and Filter

Components

- filters

Connectors

- pipes

Constraints

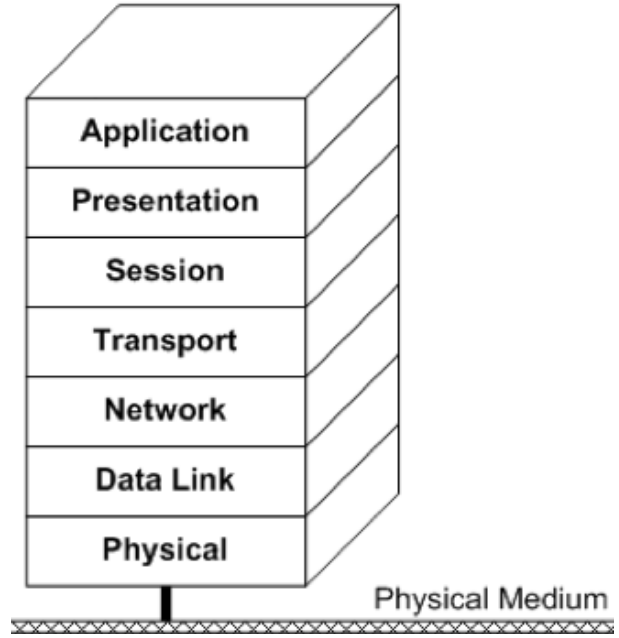
- Filters must be independent entities and should not share state with other filters
- Filters do not know about other filters

Best known use

- Unix shell (ls –l invoices | grep –e August | sort)

[Czarnecki]

Layered Systems



- Layered systems are hierarchically organized providing services to upper layers and acting as clients for lower layers

- Lower levels provide more general functionality to more specific upper layers

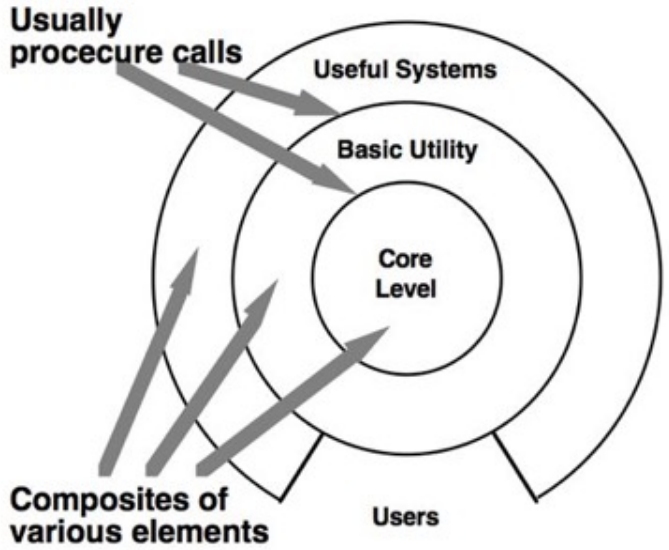
Examples:

Virtual machine

n-tier systems

Client-server

Layered Systems



[Garlan and Shaw. An Introduction to Software Architecture.]

Components:

- layer

Connectors:

- protocols

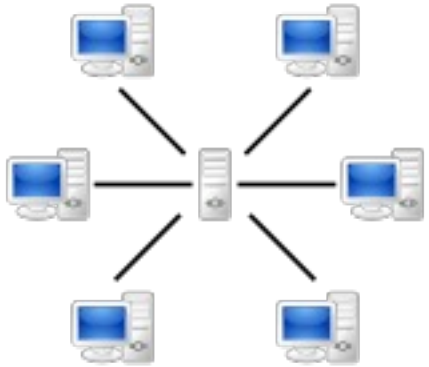
Constraints:

- In a strict layered system, programs at a given level may only access adjacent layers

Best known use:

- Layered communication protocols, e.g. TCP/IP

Client-Server (layered)



[picture from wikipedia]

Components & Connectors

- clients and servers
- remote procedure call, network protocols

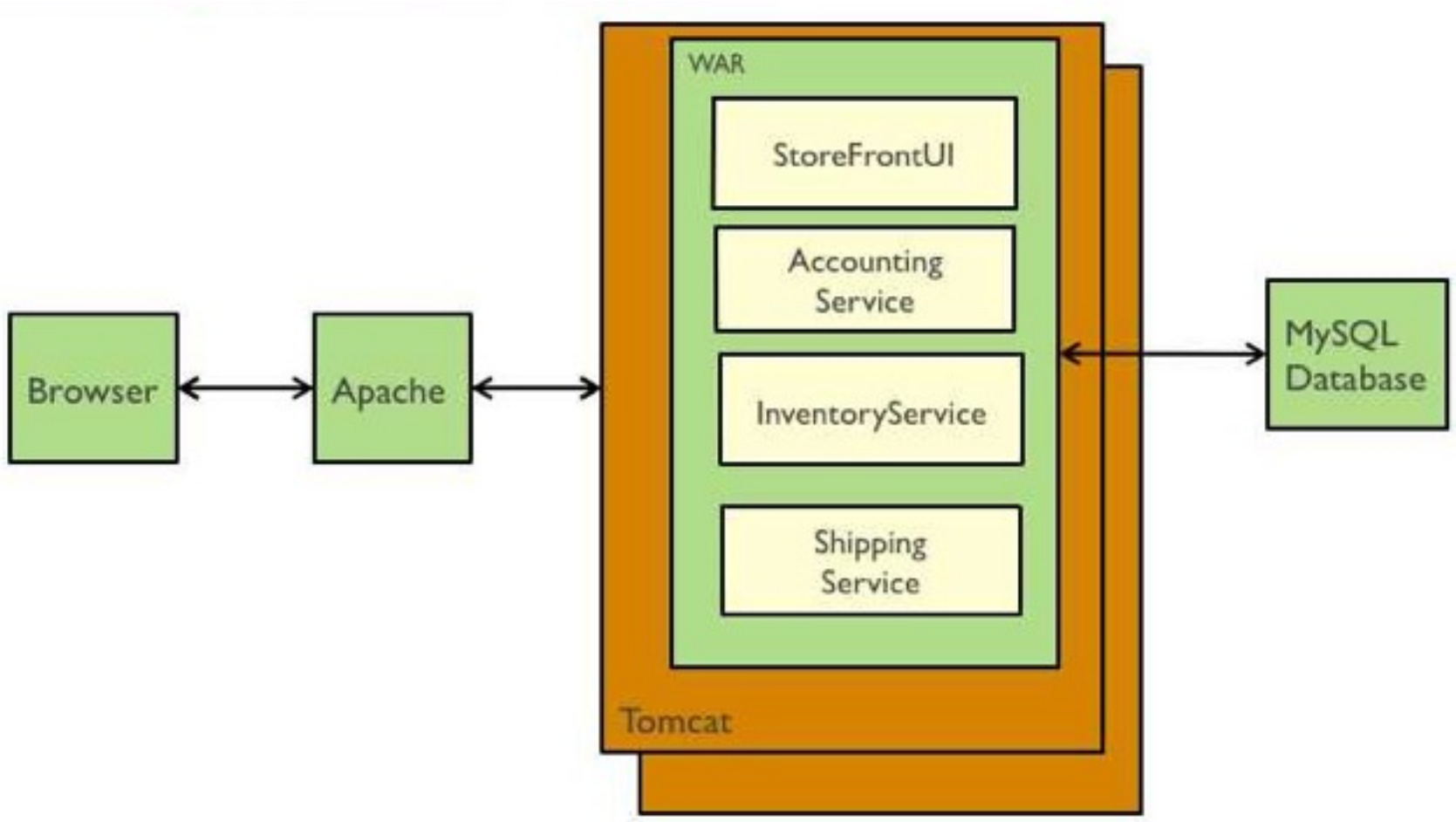
Topology & Constraints

- two-level, with multiple clients making requests to server
- client-to-client communication prohibited

Characteristics

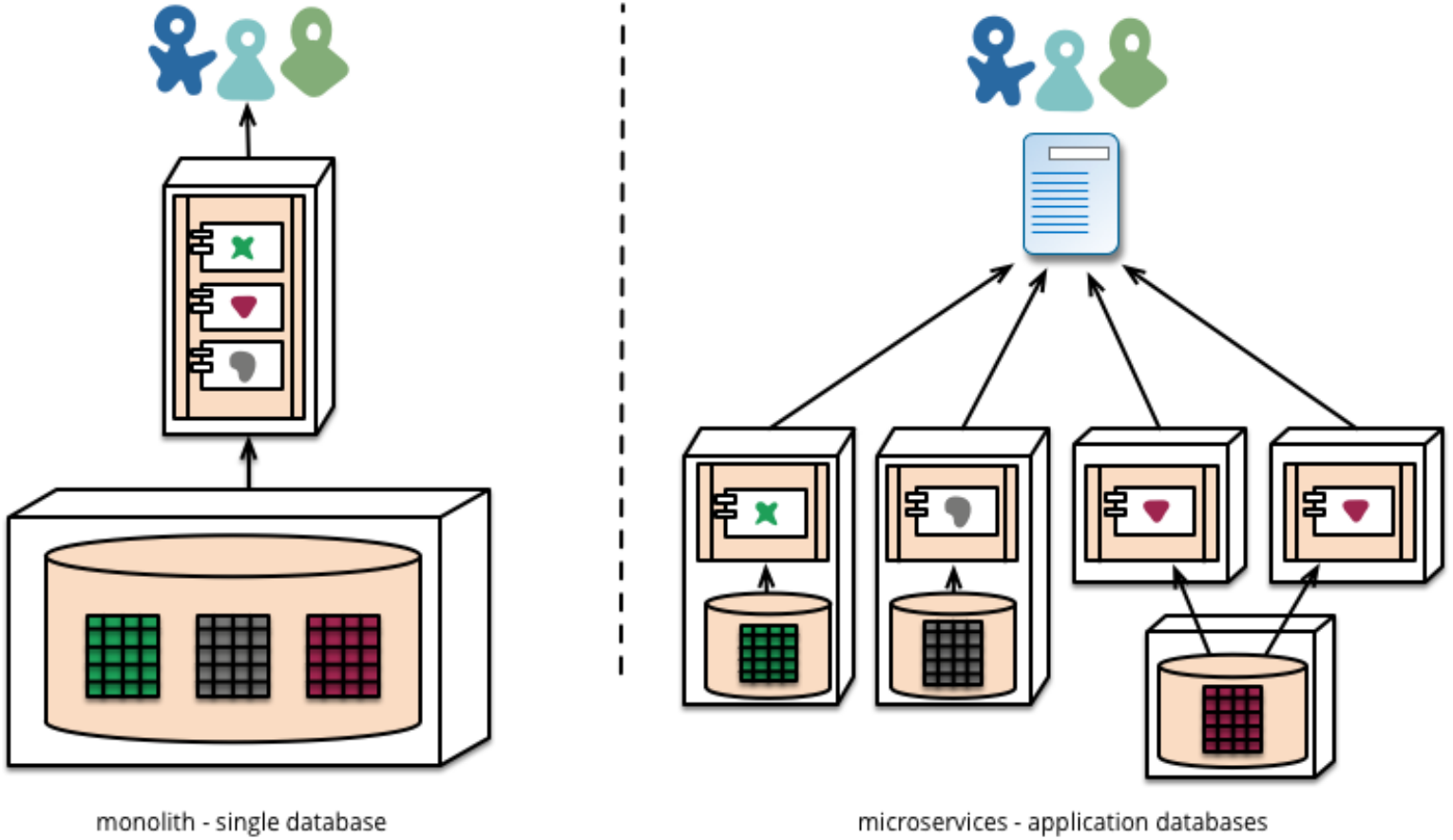
- ✚ Distribution with centralization of computation and data on server; scalability (single server can serve many clients)
- – Responsiveness (if network is slow), robustness (if server goes down)

Traditional Web Application Architecture (n-tier)



Chris Richardson, Monolithic Architecture Pattern

Monolith → Microservices



monolith - single database

microservices - application databases

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

Microservices

Components

- services: organized around (business) capabilities

Connectors

- Network protocols, e.g. HTTP/REST

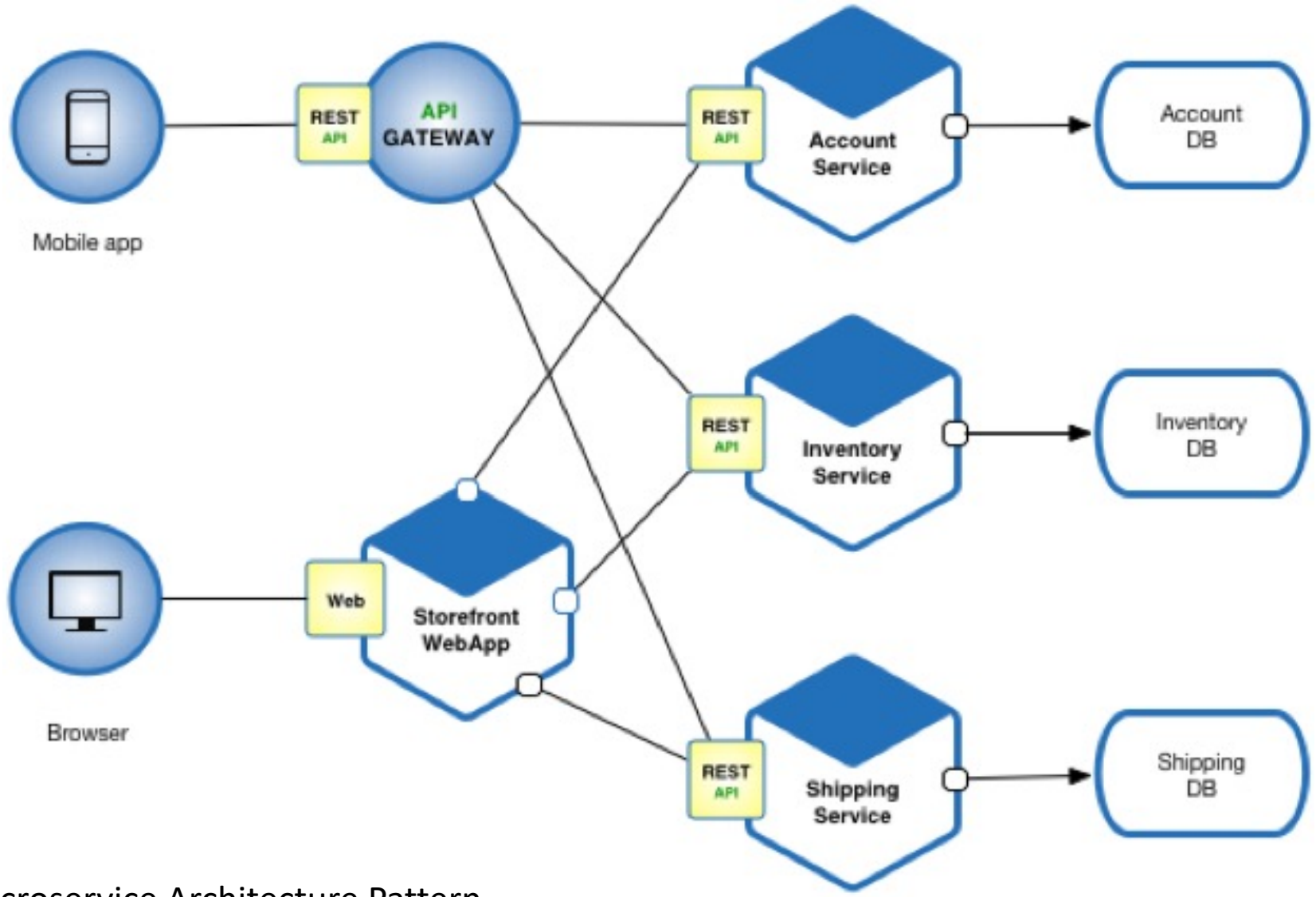
Benefits

- Independently deployable, easier to scale (parts), improved fault isolation

Drawbacks

- Additional complexity of deployment/operation, testing service interaction is more difficult, increased memory consumption

Microservice – Example



Chris Richardson, Microservice Architecture Pattern

Example: Measuring & Visualizing Heart Rate [for reflection, not in class]

Heart Rate Sensor

ECG wave

HR visualization over time

Some of the steps involved:

Sensor -> digitization -> filter noise -> ECG data -> calculate measures (HR,…) -> active period detection -> visualization

https://buy.garmin.com/en-US/US/p/552982

Exercise/Discussion: Measuring & Visualizing Heart Rate [for reflection, not in class]

Heart Rate Sensor

ECG wave

HR visualization over time

Imagine you were to design/sketch an architecture for the system.

Which style would you choose and why?

What are the main components and connectors you'd have?

Steps involved:

Sensor -> digitization -> filter noise -> ECG data -> calculate measures (HR,…) -> active period detection -> visualization

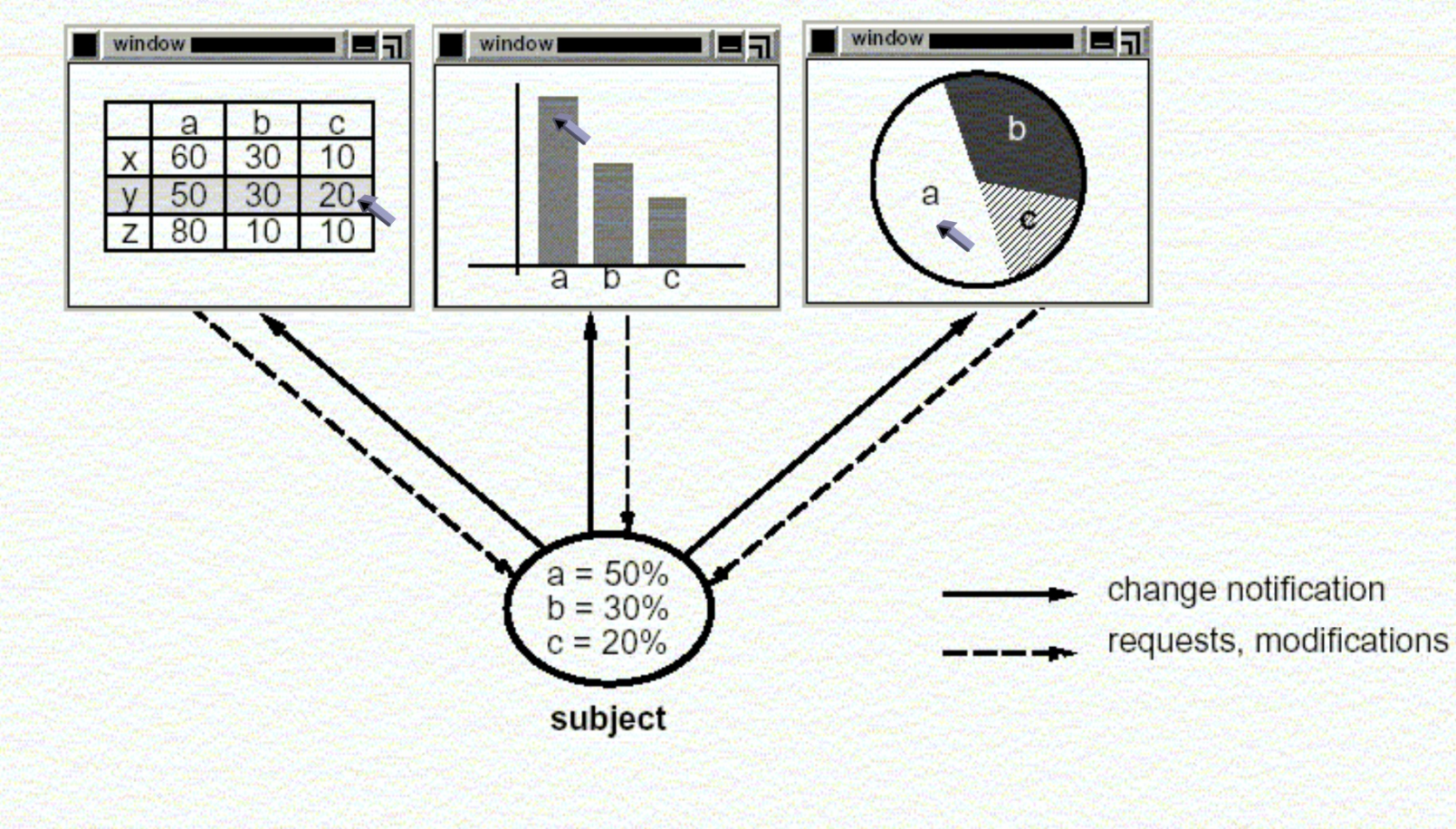
MV*
(Implicit Invocation Example)

Learning Objectives

Be able to:

- Explain MV* and discuss their differences, benefits and limitations

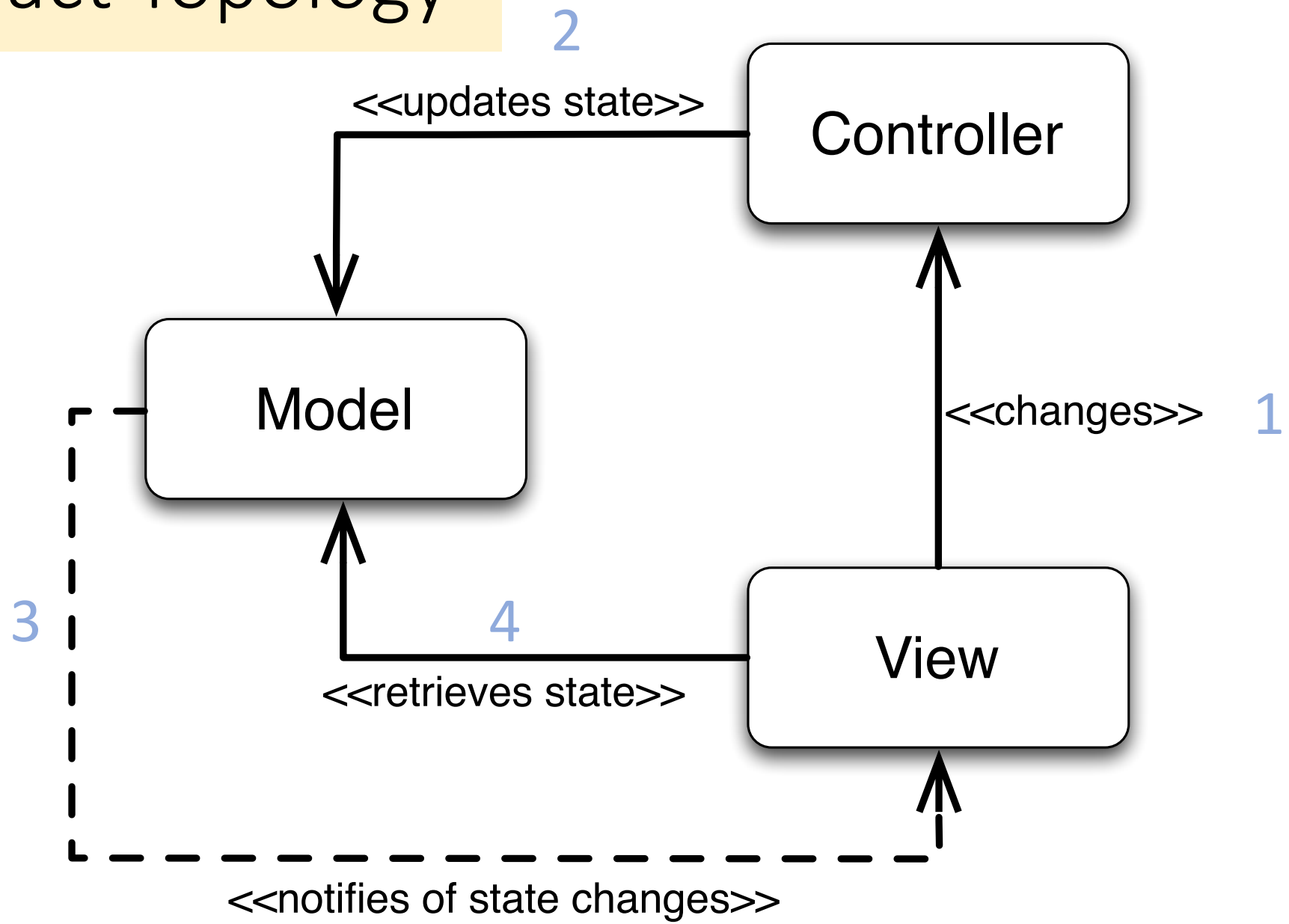
Model-View-Controller (implicit invocation)



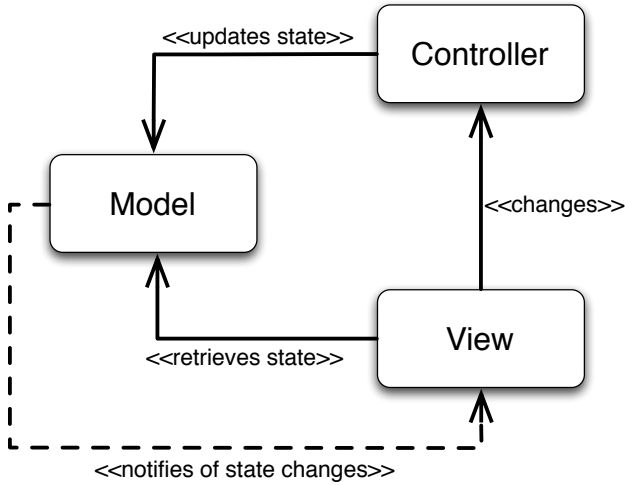
MVC Motivation

- UI changes more frequently than business logic
  - e.g., layout changes (esp. in web applications)
- The same data is often displayed in different ways
  - e.g., table view vs chart view
  - the same business logic can drive both
- Designers and developers are different people
- Testing UI code is difficult and expensive
- **Main Goal:** Decouple models and views
  - Increase **maintainability/testability** of system
  - Permit new views to be developed

Abstract Topology



Model View Controller



- Model
  - contains application data (often persisted to data store)
  - often the Subjects in the Observer design pattern
- View
  - presents model to user
  - allows user to manipulate data (does not store data)
  - configurable to display different data
- Controller
  - glues model and view together; updates model when user manipulates view
  - houses application logic
  - loose coupling between model and others; view tightly coupled/cohesive with controller

Compound Pattern / Style

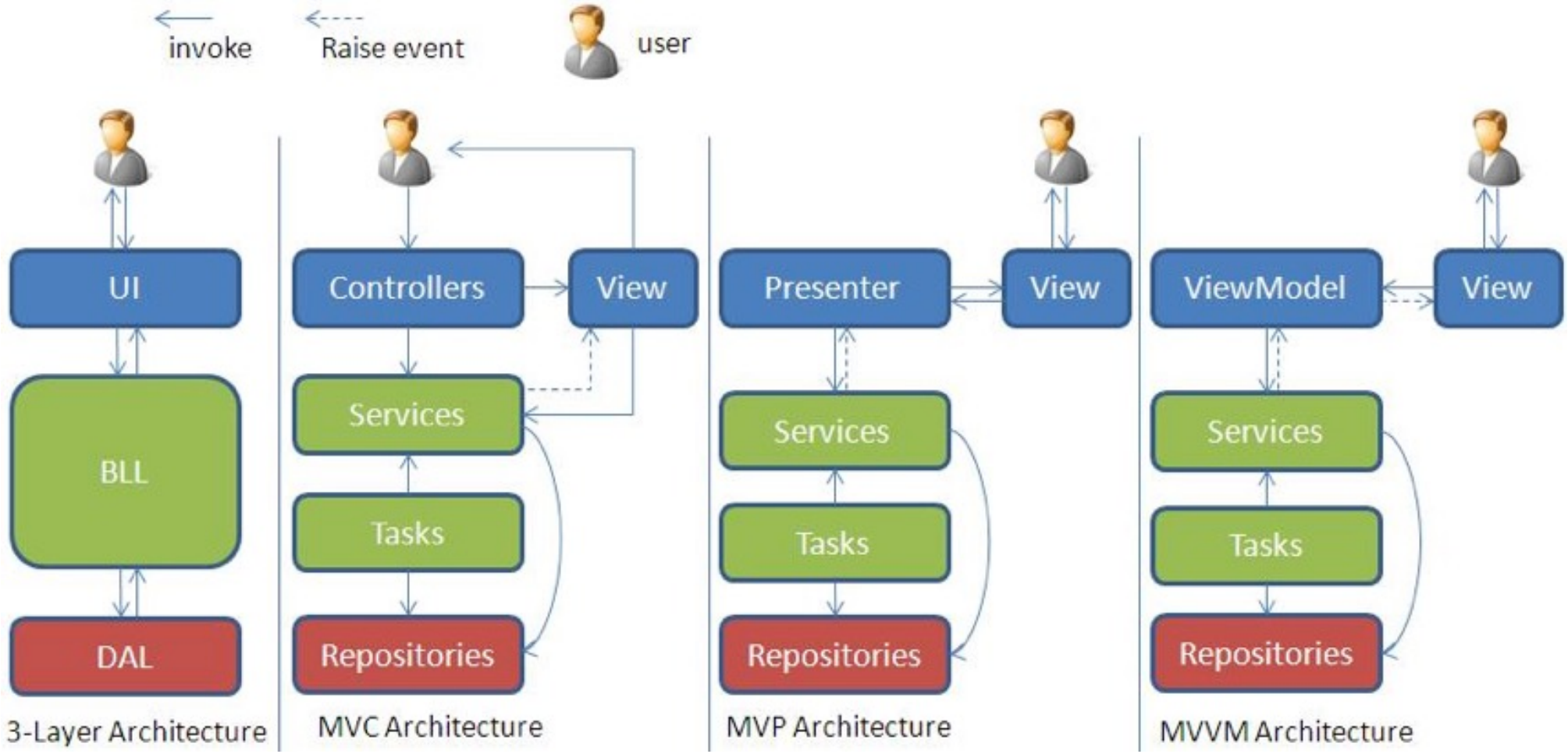
MVC (and other similar patterns) rely upon several more basic design patterns

In MVC

- View (context) / Controller (strategy) leverage the strategy pattern
- View often uses a composite pattern (for nested views)
- View (observer) / Model (subject) interact through the observer pattern

Other meta-patterns rely upon similar lower-level design patterns

MVC, MVP and MVVM



Originally from https://merroun.wordpress.com/2012/03/28/mvvm-mvp-and-mvc-software-patterns-againts-3-layered-architecture/

Code Exercise
–
Design Pattern

Code Exercise – Observer Pattern

1. Download the solitaire.zip file from OLAT
2. Unzip file and open folder (File > Open Folder…) in your IDE (IntelliJ or VSCode) and run it:
   - VSCode: Go to the file `**module-info.java**` file in the source 'src' folder and click on the run button (top right: 'Run Java'). You should then choose the target Solitaire (src/ca/mcgill/cs/stg/solitaire/gui/Solitaire) to run the application.
   - IntelliJ: Click on the green run Button and choose the target Solitaire

3. Describe how the Observer Desing Pattern has been implemented in the Solitaire Game. Investigate the program code regarding following points:
   - Which classes are Listeners/Observers, how do they register, and what do they observe (what is the Subject)?
   - How and where do the Listeners/Observers get notified about a state change?
   - How do the Listener/Observer objects react to a state change?
   - Draw a class diagram of the Observer Pattern in the Solitaire Game.

Code Exercise – Visual Studio Code for Java

1. Install and use **Java17** on your machine
2. Download and install maven build tools (VSCode)
3. Install the Extension Pack for Java (VSCode)

Code Exercise – Observer Pattern

Q: Which classes are Listeners, how do they register, and what do they observe (what is the subject)?

A: The classes *CardPileView*, *DeckView*, *DiscardPileView*, *SuitStack* implement the GameModelListener interface, and they add/register themselves as listeners for the GameModel in their constructor:

```
DiscardPileView(GameModel pModel)
{
    aModel = pModel;
    // ...
    aModel.addListener(this);
}
```

The listeners observe the State of the GameModel object.

Code Exercise – Observer Pattern

Q: How and where do the Listeners/Observers get notified about a state change?

A: The GameModel object has a reference to a list of all Listener/Observer objects, and when the notifyListeners method is called it notifies all listeners by calling gameStateChanged():

```
private void notifyListeners()
{
    for( GameModelListener listener : aListeners )
    {
        listener.gameStateChanged();
    }
}
```

Code Exercise – Observer Pattern

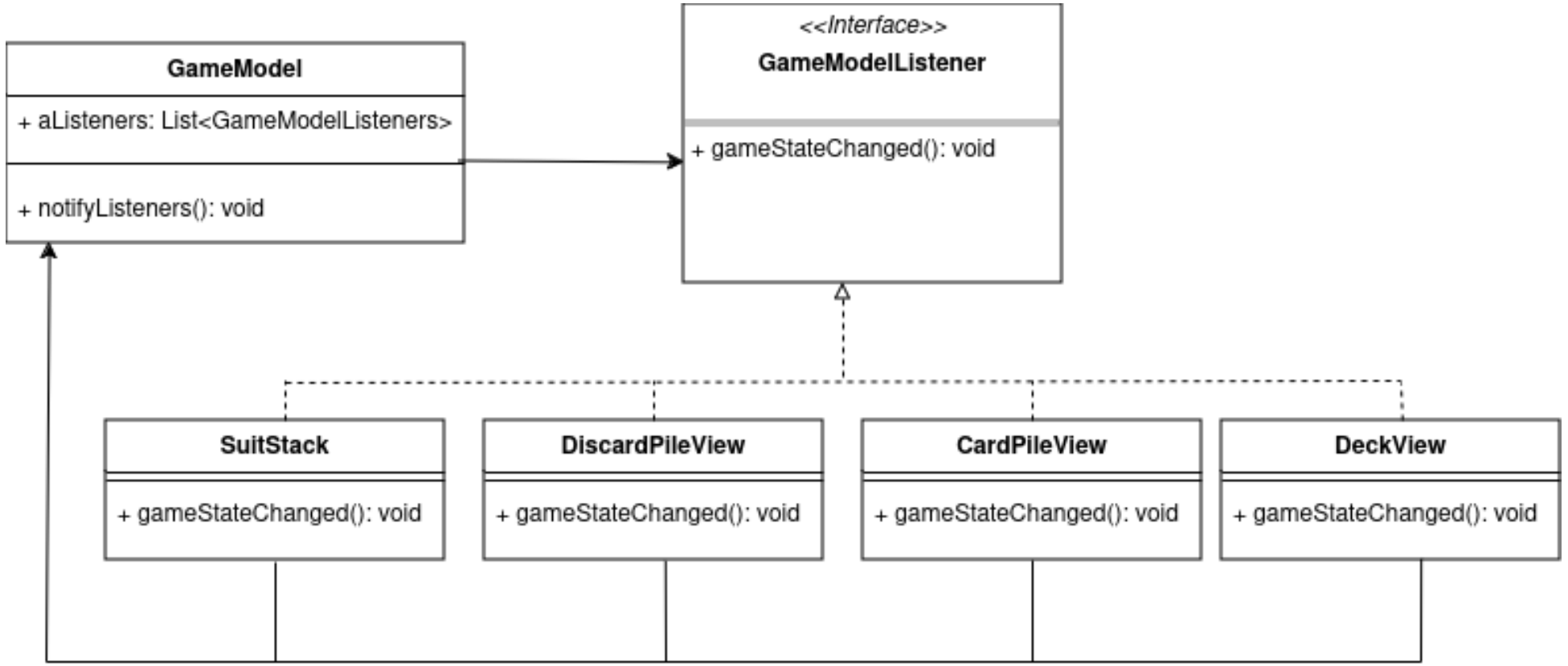
Q: How do the Listener/Observer objects react to a state change?

A: For example, in class DeckView: The view of the deck gets updated based on whether the deck is empty or not.

```
@Override
public void gameStateChanged()
{
    if( aModel.isDeckEmpty() )
    {
        ((Button)getChildren().get(0)).setGraphic(createNewGameImage());
    }
    else
    {
        ((Button)getChildren().get(0)).setGraphic(new ImageView(CardImages.getBack()));
    }
}
```

Code Exercise – Observer Pattern

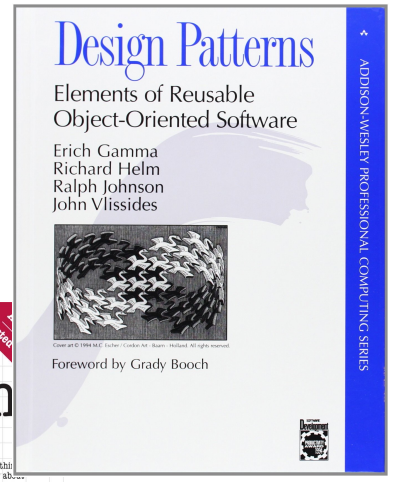
Q: Draw a class diagram of the Observer Pattern in the Solitaire Game



Quiz & More

Additional Resources

- Gamma, Helm, Johnson, Vlissides. *Design Patterns.* Addison-Wesley.
- Freeman et. Al. Head First Design Patterns.



- Wikipedia (don't trust it blindly!)
- https://sourcemaking.com/design_patterns/
- Bob Tarr's course
  - http://userpages.umbc.edu/~tarr/dp/spr03/cs491.html
- Quick design patterns reference cards
  - www.mcdonaldland.info/2007/11/28/40/
- Test (difficult): www.vincehuston.org/dp/patterns_quiz.html

Additional Resources

- See optional readings in OLAT
- David Garlan and Mary Shaw “An Introduction to Software Architecture”
- Richard Taylor et al. “Software Architecture: Foundations, Theory and Practice” (Wiley)
- MV*
  - http://www.teehanlax.com/blog/model-view-viewmodel-for-ios/
  - https://www.objc.io/issues/13-architecture/mvvm/
  - https://www.objc.io/issues/1-view-controllers/lighter-view-controllers/

Quiz

1. In the Model View Controller, all components are directly connected with each other in some way, whereas in Model View Presenter, the Model and View are not directly connected with each other.

   [True/False]

Quiz

2. Imagine that you are tasked with developing software for an automated plant watering system. The system's primary function is to adjust the watering volume based on the growth stage of the plant. For example, a small, young plant requires less water, whereas a larger, more mature plant needs more. After each watering session, the system uses a camera to assess the plant's size and then determines the appropriate amount of water for the next session. Given these requirements, which design pattern would be most appropriate to manage the dynamic watering schedule based on the observed plant growth stages?

    A) Strategy Pattern

    B) State Pattern

Quiz

3. In an online multiplayer game, the behavior of the player can change depending on various power-ups (like invisibility or speed boost) collected during gameplay. These behaviors can be added dynamically and should be removable. Which design pattern of the following best supports this kind of dynamic behavior modification (only choose one)?

   [Decorator / Singleton / State / Strategy]

Quiz

4. You are tasked with designing a system that controls the distribution of tasks among different services in a cloud-based application. Each service should be able to handle tasks independently and scale based on demand. Which architectural style fits this scenario best?

   [MVC / Singleton / Microservices / Observer]