# Design Patterns

## Software Engineering II

Author: Eng. Liliana Marcela Olarte, M.Sc.
lmolartem@unal.edu.co

Author: Eng. Carlos Andrés Sierra, M.Sc.
casierrav@unal.edu.co

Lecturers
Computer Engineering Program
School of Engineering
Universidad Nacional de Colombia

2025-II

# Outline



UNIVERSIDAD NACIONAL DE COLOMBIA

# Outline



UNIVERSIDAD NACIONAL DE COLOMBIA

# Basics of Design Pattern I

- A **design pattern** is a practical proven solution to a recurring design problem in software engineering.
- A **design pattern** is not a finished design, just flexible or reusable parts of a complete solution.
- Created (or discovered) for **expert developers**, and had been matured by non-expert developers, they are like *food recipes*, or *play tik-tik-toe*.
- There are **23 design patterns**, classified in *three categories*: creational, structural, & behavioral patterns.
- Reported by the **Gang of Four** (**GoF**) in 1994, in the book Design Patterns: Elements of Reusable Object-Oriented Software.

# Basics of Design Pattern I

- A **design pattern** is a practical proven solution to a recurring design problem in software engineering.
- A **design pattern** is not a finished design, just flexible or reusable parts of a complete solution.
- Created (or discovered) for **expert developers** and had been matured by non-expert developers, they are like *food recipes*, or *play tik-tik-toe*.
- There are **23 design patterns**, classified in *three categories*: creational, structural, & behavioral patterns.
- Reported by the **Gang of Four** (**GoF**) in 1994, in the book Design Patterns: Elements of Reusable Object-Oriented Software.

# Basics of Design Pattern I

- A **design pattern** is a practical proven solution to a recurring design problem in software engineering.
- A **design pattern** is not a finished design, just flexible or reusable parts of a complete solution.
- Created (or discovered) for **expert developers**, and had been matured by non-expert developers, they are like *food recipes*, or *play tik-tik-toe*.
- There are **23 design patterns**, classified in *three categories*: creational, structural, & behavioral patterns.
- Reported by the **Gang of Four** (**GoF**) in 1994, in the book Design Patterns: Elements of Reusable Object-Oriented Software.

# Outline



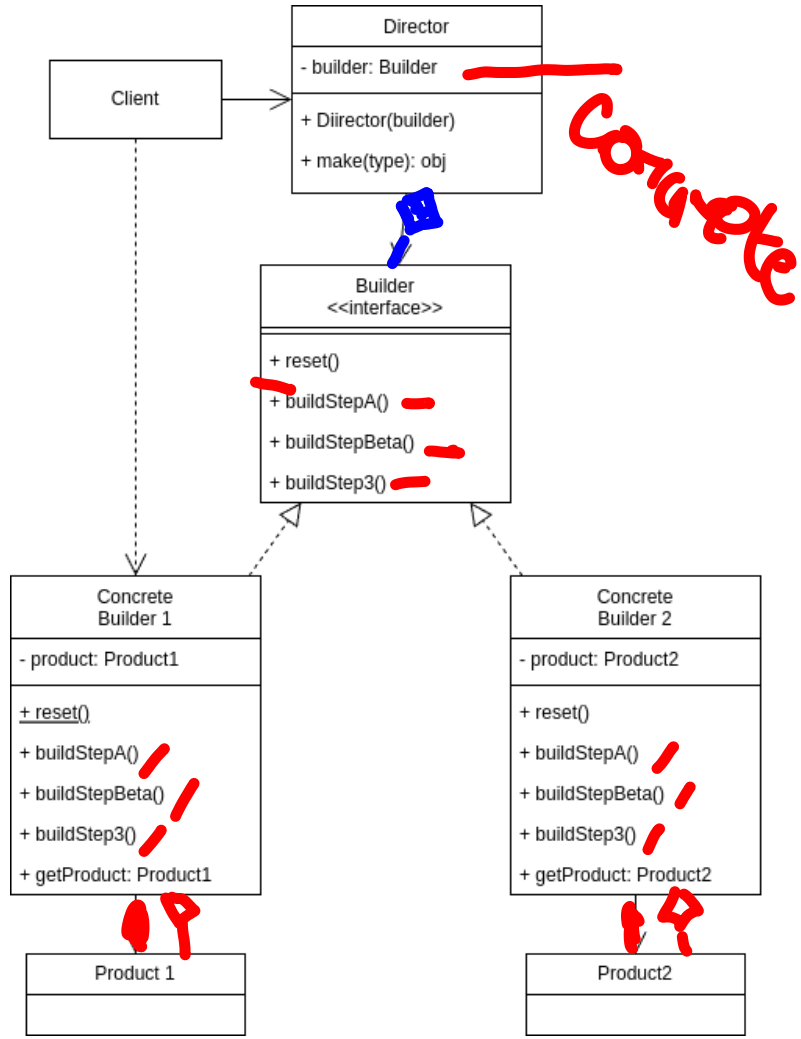
UNIVERSIDAD NACIONAL DE COLOMBIA

# Basic Concepts

- **Intent**: Separate the construction of a complex object from its representation so that the **same construction process** can create **different representations**.
- **Motivation**:
  - **Problem**: An application needs to **create instances** of a class, but the class is abstract and has many **possible implementations**.
  - **Solution**: Provide different object creation mechanisms, which **allow** the client to **create the object** without knowing the actual implementation. This increase flexibility and reuse of code.

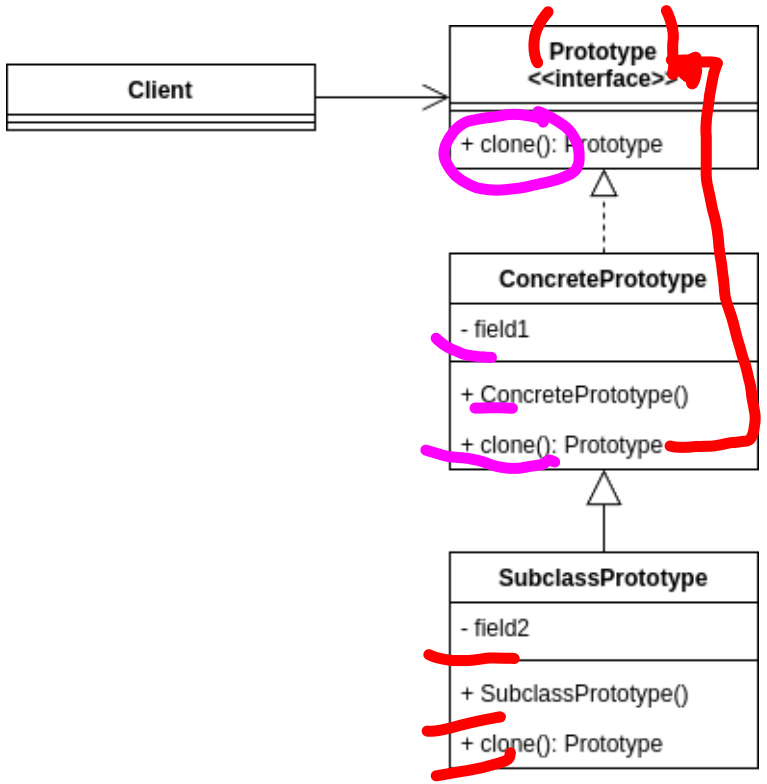
# Builder Pattern — Concepts

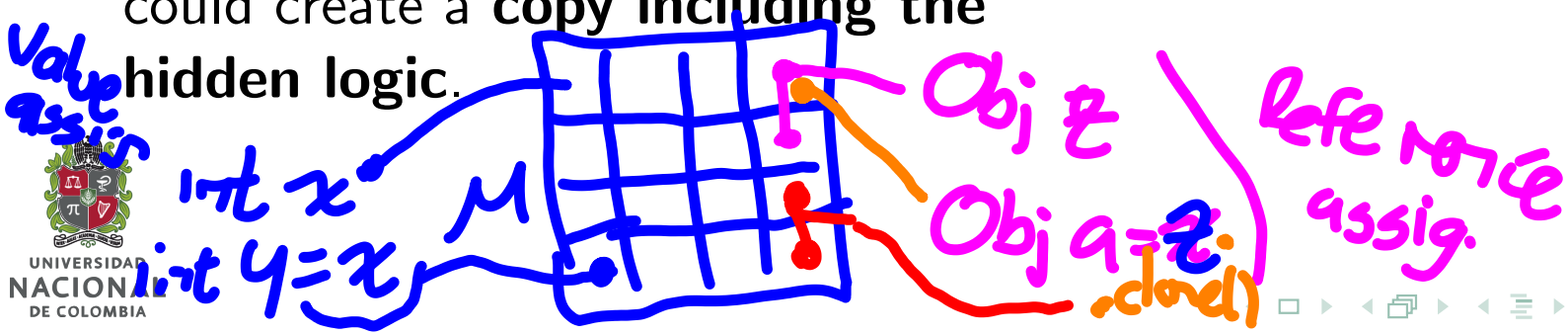
- It is a pattern that lets **construct** a complex object **step by step**. The idea is to create **different representations** of an object using the same construction code.
- A **problem** is work with a class that has **many attributes** and it is difficult to create an instance of it. It gets worse when there are many possible representations of the object because some are optional.
- The **solution** is to encapsulate the object construction and use separate **methods** to add or build the object attributes.



# Prototype Pattern — Concepts

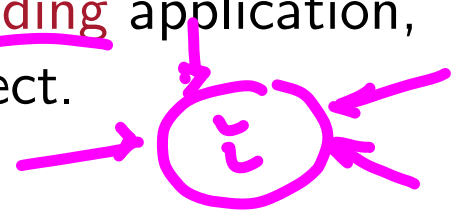
- It is based on **copy** of an **existing object**. It is used when the type of objects to create is determined by a prototypical instance, which is **cloned** to produce new objects.
- Remember, **clone** is not just copy an object, it is **create** a new **object** with the **same** attributes and values of the original object.
- It solves the **problem** of copy the private attributes of an object. So, you could create a **copy including the hidden logic**.



# Singleton Pattern — Concepts

- In an attemp to **reduce memory consumption**, this pattern ensure that a class has only **one instance** and provide a global point of access to it.
- It is used when you need to **control** the **number of instances** of a class, so just one class instance is allowed across all the application. Also it allows to apply the concept of **lazy creation**.
- It is pretty simple: just create a class with a method that **creates** a new **instance** of the class if one does not exist. If an instance already exists, it returns a reference to that object.
- It violates the *Single Responsability Principle* and the *Open/Closed Principle*. Also, internal instance and get method are *static*.
- **Not** a very **good idea** if you are using a multi-trending application, could be issues trying to access a shared single object.



# Singleton Pattern — Classes Structure

Think in a circle room with several doors but *just one doorman*.

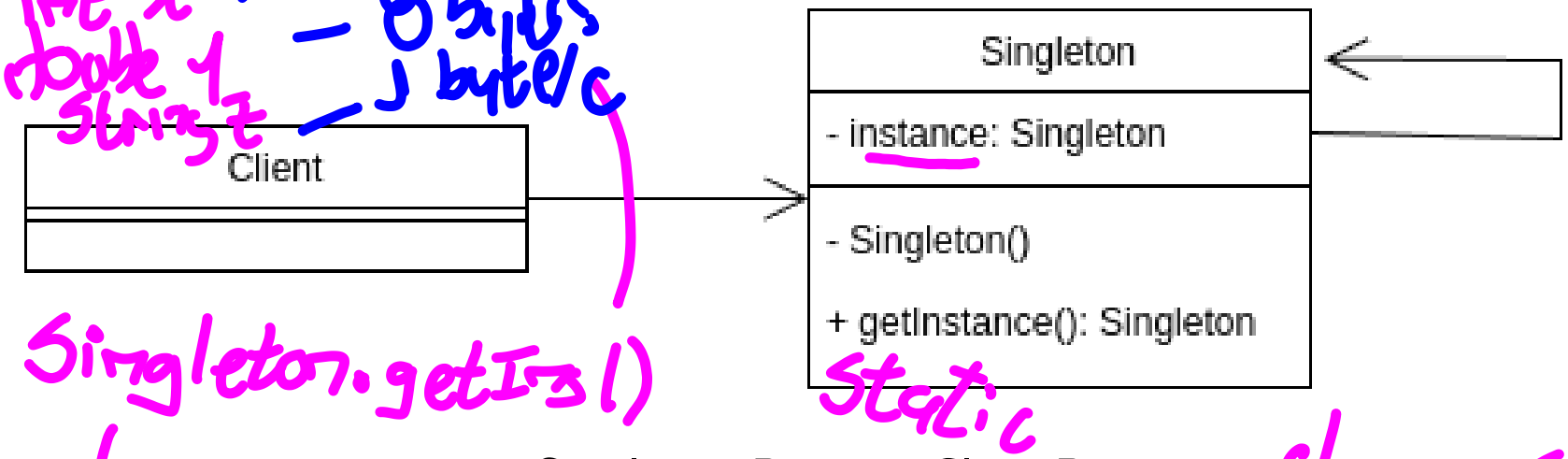



Figure: Singleton Pattern Class Diagram

Demo time!

# Factory Pattern — Concepts

- It is pattern based on a **superclass** and the subclasses could alter the type of objects to be created.
- One of the **most common** design pattern used, is simple, powerful and flexible. It is used with many other **design patterns**.
- It lets make **simple** a complex code. If you have **groups** of **objects** that are created in a similar way, the **factory method** is the best choice.
- The **client** just needs to interact with the **factory**, and the **factory** will create the object. The **client** does **not** need to **know** the actual implementation of the object (or the *subclasses*).

# Factory Pattern — Classes Structure

It is like to watch Charlie and the Chocolate Factory.

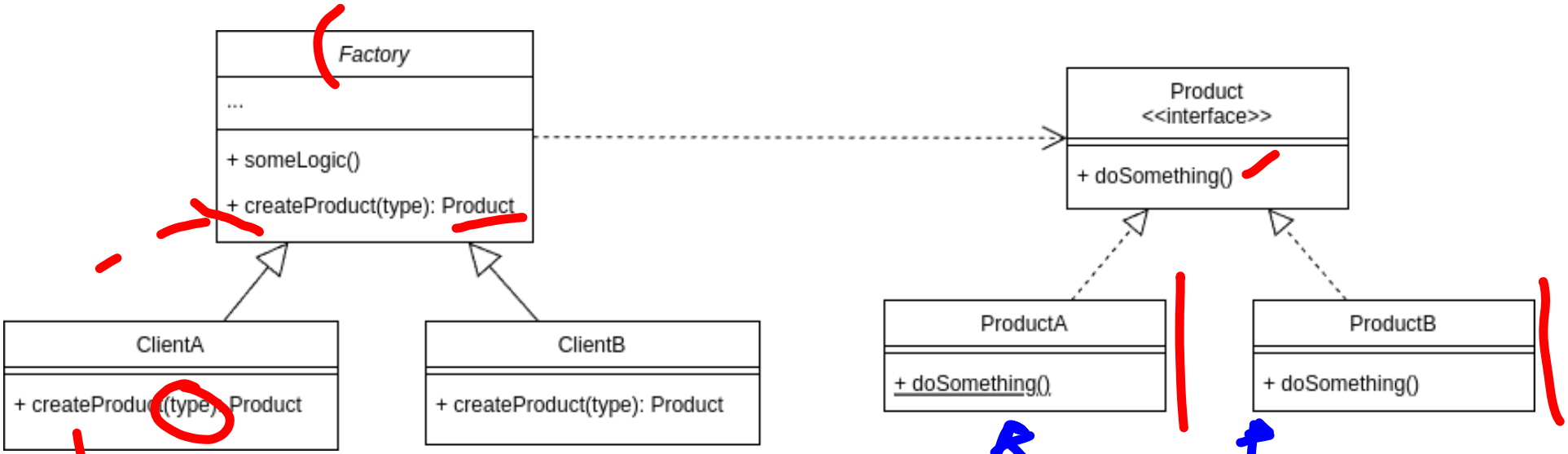



Figure: Factory Pattern Class Diagram

if (type="a")
else if (type="b")

Demo time!

# Abstract Factory Pattern — Concepts

- This is a pattern that lets you **produce families** of related objects without specifying their concrete classes.
- It is a **super factory** that creates other factories. It is used when you have a **super class** that can create **subclasses** and the **subclasses** can create **objects**.
- Also this pattern allows to keep the **client** code decoupled from the **actual objects** in the system. Keep old code when you need to add new representations.
- It is used when you have **many objects** that can be grouped in **families**.

UNIVERSIDAD NACIONAL DE COLOMBIA

# Conclusions

- There are a few ways to **create objects** inside an application in a pretty eficient way.You just need to think about it and choose the best one for your application.
- You **could combine** these patterns to create a more complex and flexible application. However, you need to be careful with the complexity of the application.
- The **Builder** pattern is used to create a complex object step by step.
  The **Prototype** pattern is used to create a new object by copying an existing object.
  The **Singleton** pattern is used to create just one instance of a class.
  The **Factory** pattern is used to create objects in a simple way.
  The **Abstract Factory** pattern is used to create families of objects.

UNIVERSIDAD NACIONAL DE COLOMBIA

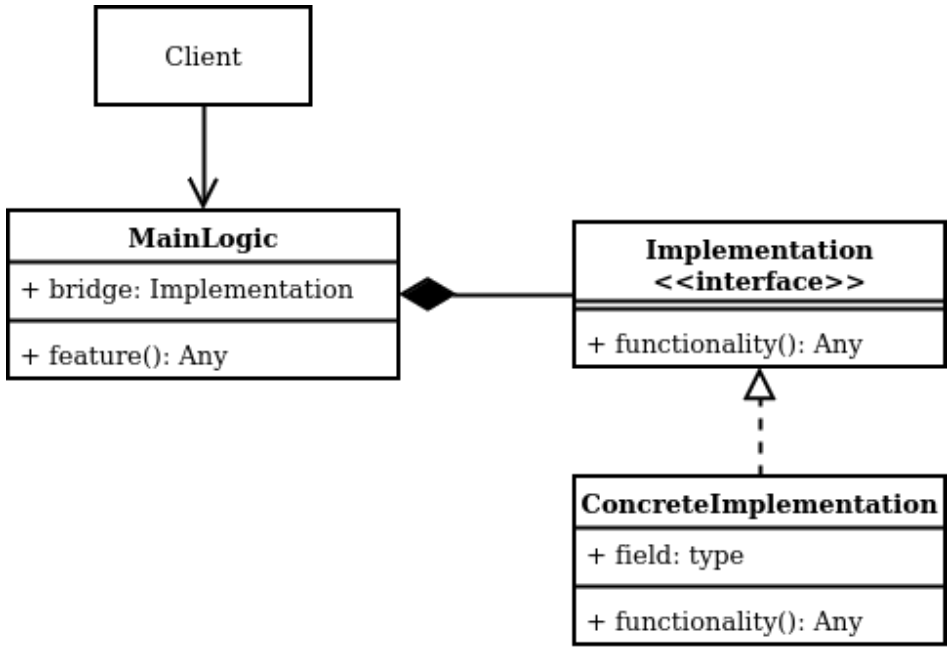
# Outline

# Basic Concepts

- **Intent**: Describe how objects are connected to each other. These patterns are related to the design principles of descomposition and generalization.
- **Motivation**:
  - **Problem**: A system is composed of multiple classes that interact with each other. The system becomes complex due to the relationships between these classes.
  - **Solution**: Structural class patterns use inheritance to compose interfaces or implementations.

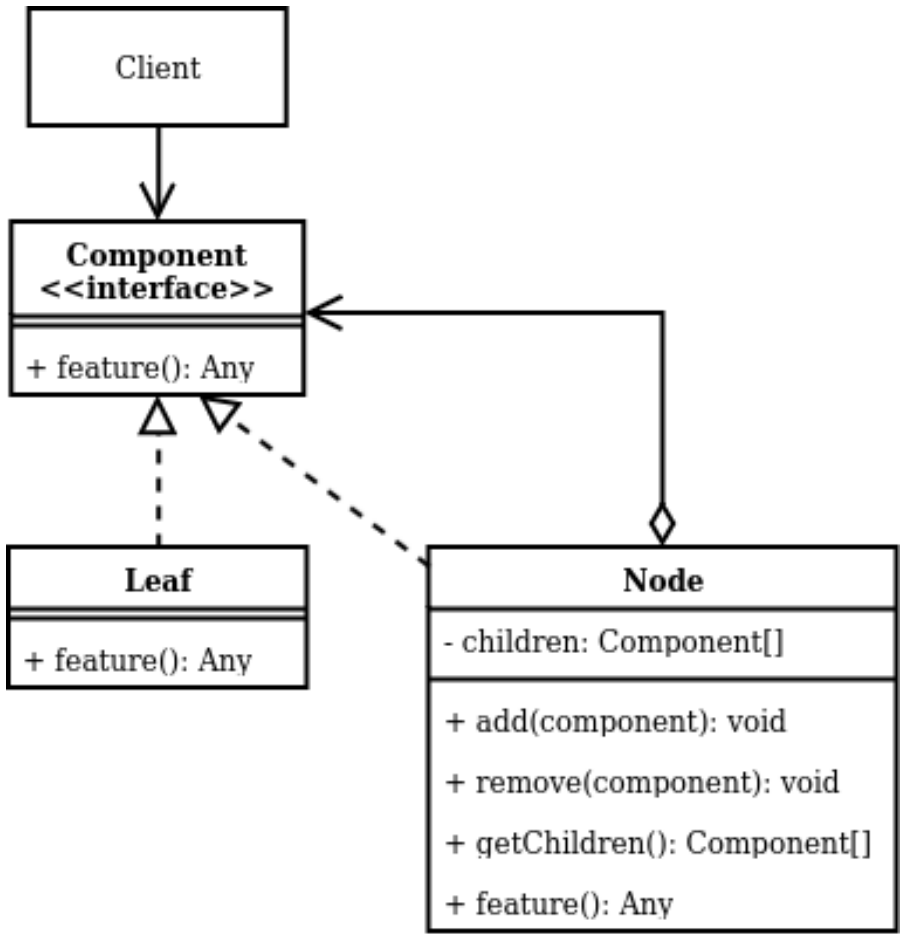
# Bridge Pattern — Concepts

- When there is a **very large class**, this pattern lets **split** it into two separate hierarchies based on *abstraction* and *implementation*.
- Also, it helps when you want to combine two different but **related classes**, and you want to keep them independent.
- This **pattern** solves this problem avoiding **heritance** and trying to switch to **object composition**.



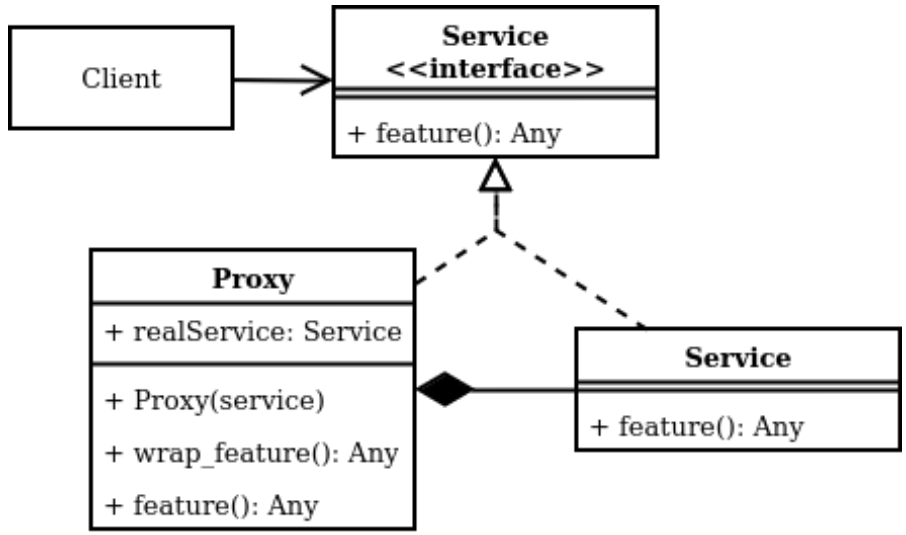
# Composite Pattern — Concepts

- Compose objects into **tree structures** to represent part-whole hierarchies.
- **Composite** lets clients treat individual **objects** and **compositions** of objects uniformly, based on tree nodes concepts.
- It **makes sense** to use this pattern if the problem could be **represented as a tree**.
- In this case, it means to think in polymorphism.



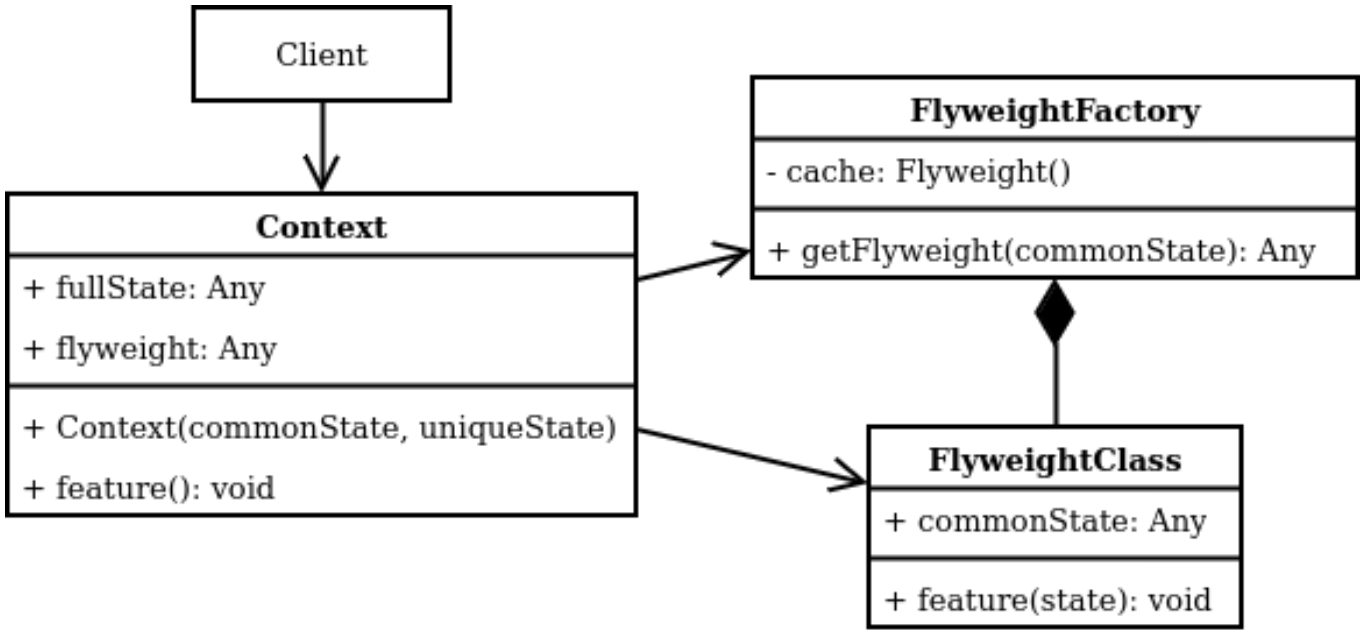
# Proxy Pattern — Concepts

- This pattern lets to provide a **substitute** for an object. In this way, access could be controlled.
- It is useful when you want to **add a level of indirection** to control access to an object. It is like a middle layer without affecting previous logic.
- It is useful when you want to **reduce memory** used in a service, similar to think in *cache memory*. Also, it lets **add additional logic** (like *logging* or *security*) to an existing logic **without** change original class.



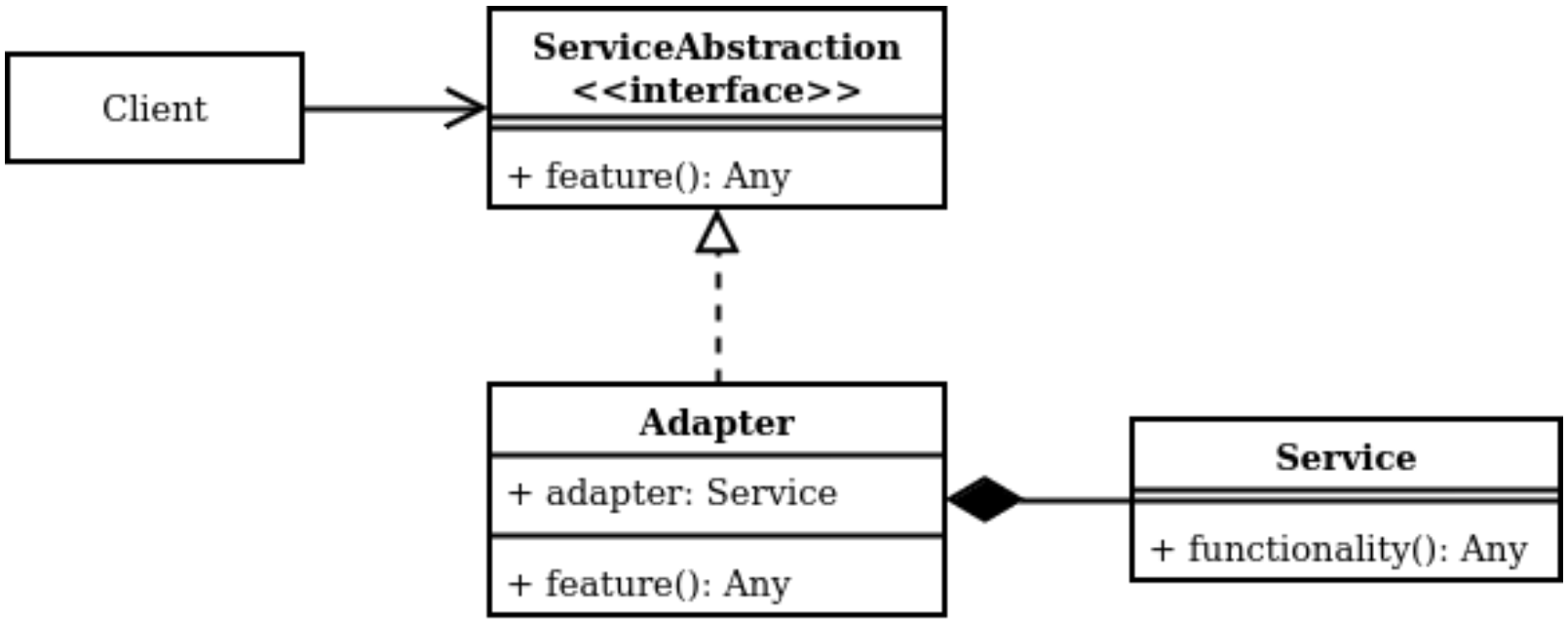
# Flyweight Pattern — Concepts

- This pattern lets you use **sharing** to support large numbers of fine-grained objects efficiently.
- The idea to **reuse** objects parts with **immutable** state. This lets share common parts and reduce memory usage.
- It **makes sense** to use it in **problems** with **high memory** comsumption, but with repeated objects, i.e. some *simulations*.



# Adapter Pattern — Concepts

- This pattern is pretty simple, it just attempts to **convert** the **interface** of a class into **another interface** clients expects.
- It is useful when you want to **reuse** an existing class, but its **not compatible** with the rest of your code, or at least where you need it.
- In increases compatibility, and lets define an architecture based on **interfaces** and not on concrete classes.



# Decorator Pattern — Concepts

- This pattern lets you **attach** additional **functionalities** to an object dynamically.
- It is useful when you want to **add** new **functionalities** to an object **without affecting** its **original logic**. Indeed, you could add same additional functionalities to different objects.
- Here the concept of **wrap** an object with another object is important. One object could have some **behaviors** from another object without heritance. It is because in this case the relation is based on object-oriented aggregation

# Decorator Pattern — Classes Structure

It is like Dr. Strange and his **Cloak of Levitation**.

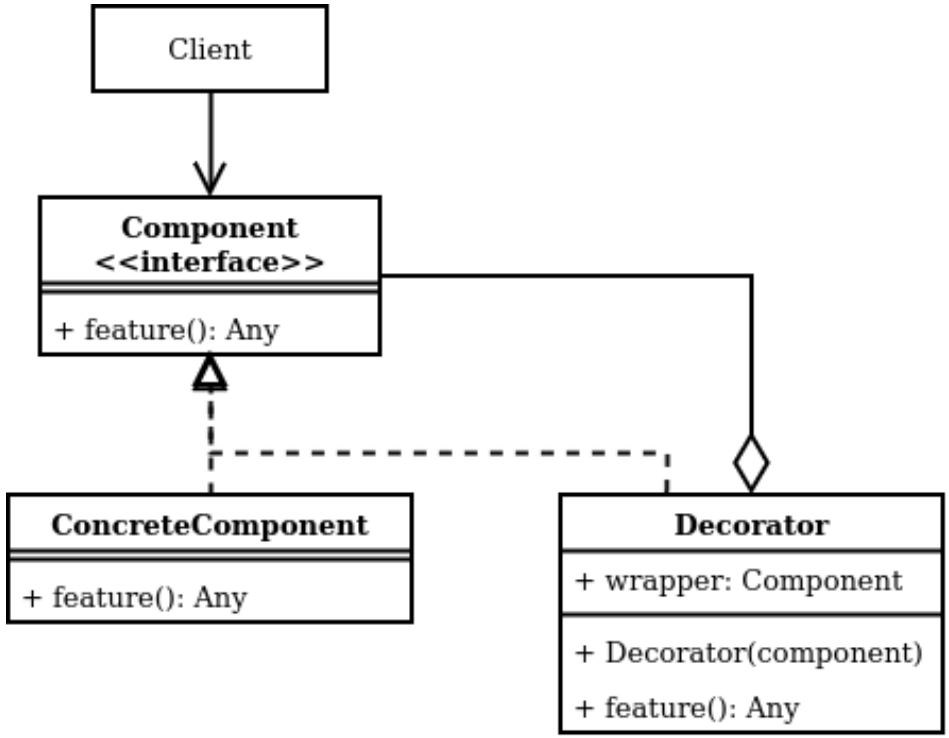



Figure: Decorator Pattern Class Diagram

Demo time!

# Facade Pattern — Concepts

- This pattern provides a **unified interface** to a set of classes that could be group into a subsystem.
- It is useful when you want to **define a high-level interface** that makes the subsystem **easier to use**. It means, hide any complex logicand let the client use a **simple interface**.
- It is normal when you want to **reduce** the **dependencies**. The client just interacts with the facade, and the **facade interacts with the subsystem**.
- You could add complexity at the subsystem and **client will not be affected**, it increases flexibility. At most, there will be more new functionalities to be exposed to the client.

# Facade Pattern — Classes Structure

You are the only one who knows how to find something in your bedroom.

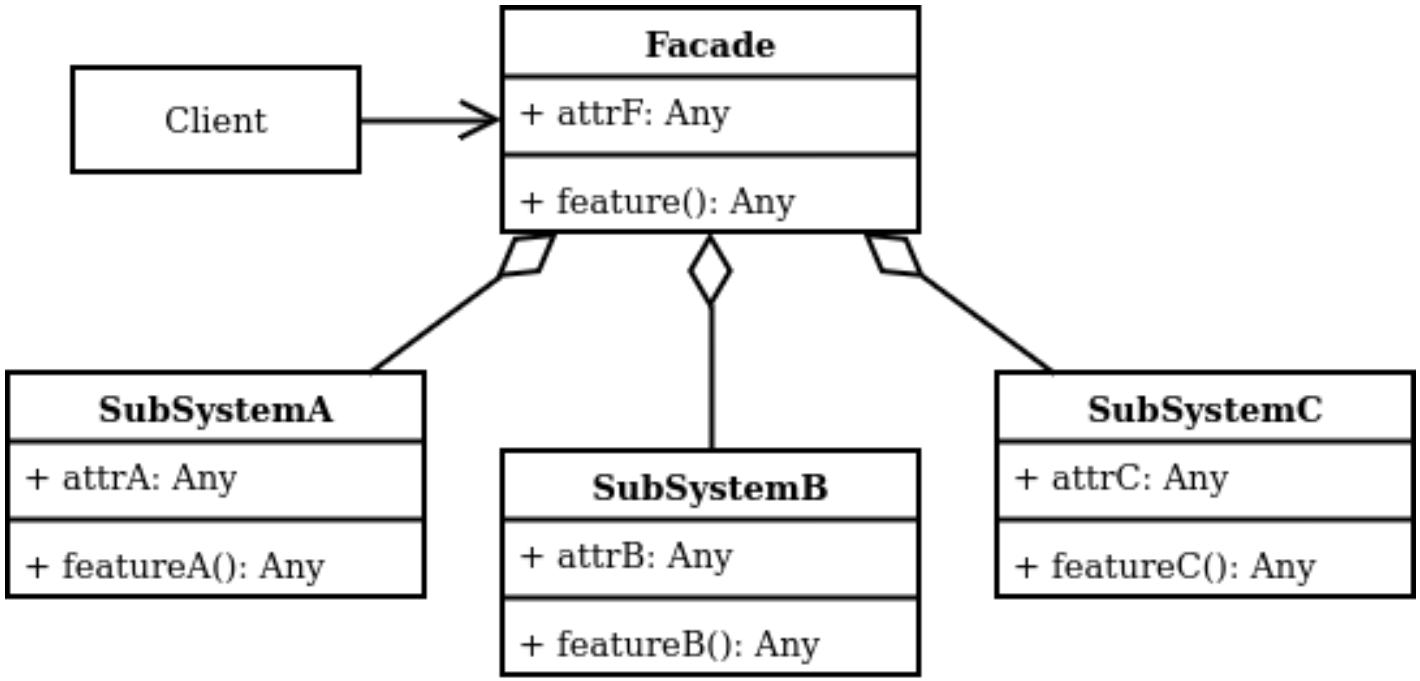



Figure: Facade Pattern Class Diagram

Demo time!

# Conclusions

- **Structural patterns** are useful to describe how objects are connected to each other.
- They are related to the **design principles** of decomposition and generalization.
- You could fix a lot of problems with these **patterns** as a nice solution. However, be careful with the complexity of the solution.
- The idea is to make the **system** flexible, reusable, and easy to maintain.

# Outline



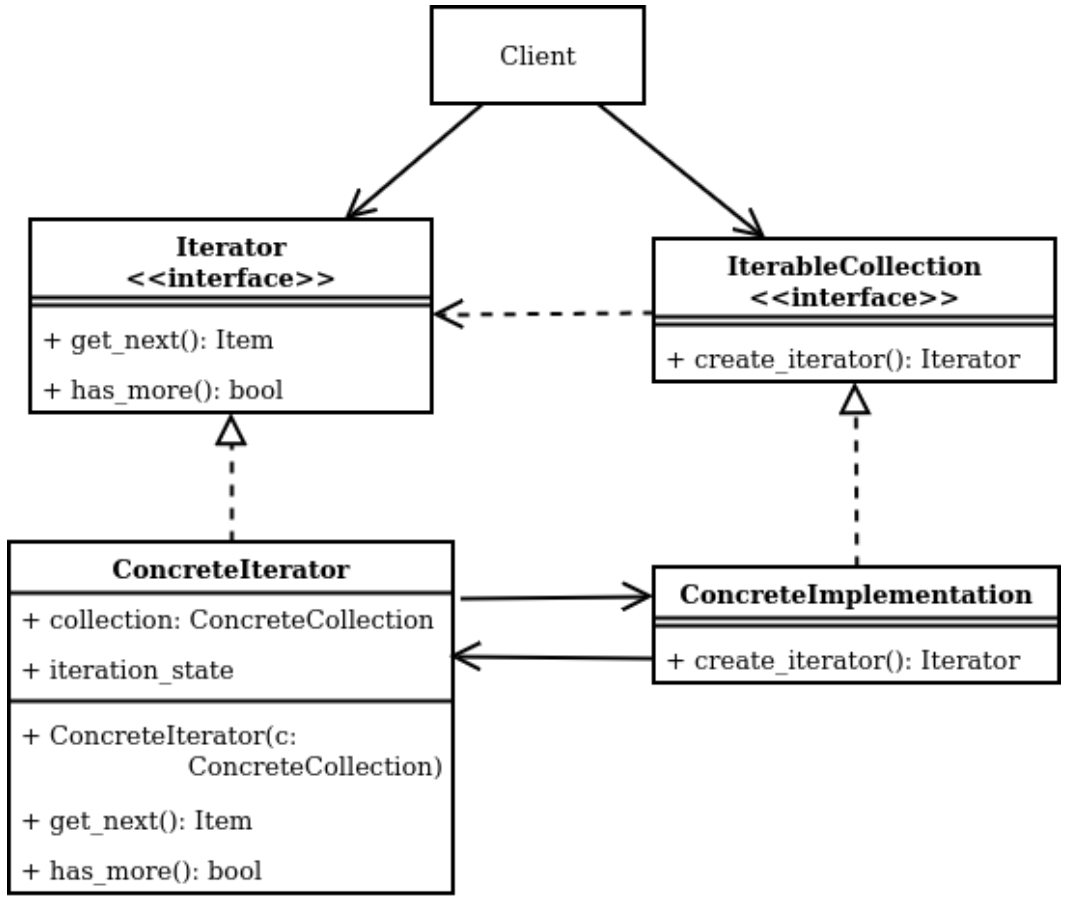
UNIVERSIDAD NACIONAL DE COLOMBIA

# Basic Concepts

- **Intent**: Focus on how classes distribute responsibilities among them, and at the same time each class just does a single cohesive function. It is like a *F1 Pits Team*, each one has a **single responsability**, but all together creates a complete team workflow.
- **Motivation**:
  - **Problem**: A system should be configured with multiple algorithms, and a system should be independent of how its operations are performed.
  - **Solution**: Define each algorithm, encapsulate each one, and make them work together.

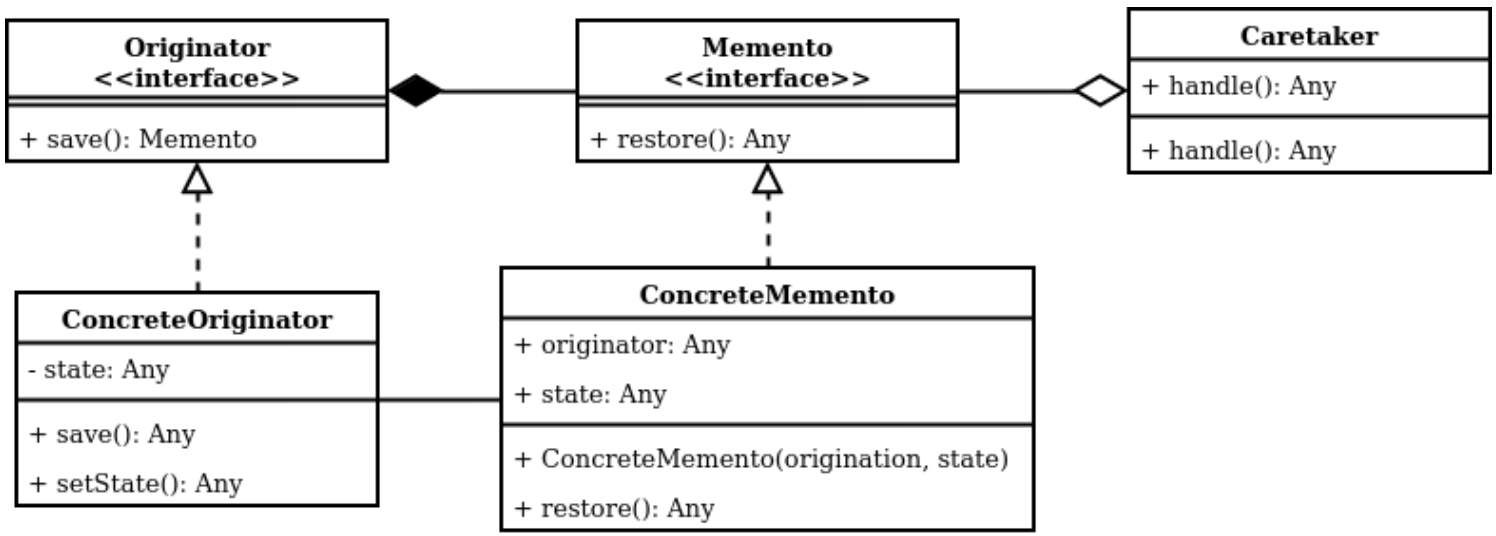
# Iterator Pattern — Concepts

- The **Iterator** pattern is a behavioral pattern that allows sequential access to the elements of an aggregate object without exposing its underlying representation.
- The **Iterator** pattern is used when you want to provide a standard way to iterate over a collection and hide the implementation details of *how* the collection is traversed.



# Memento Pattern — Concepts

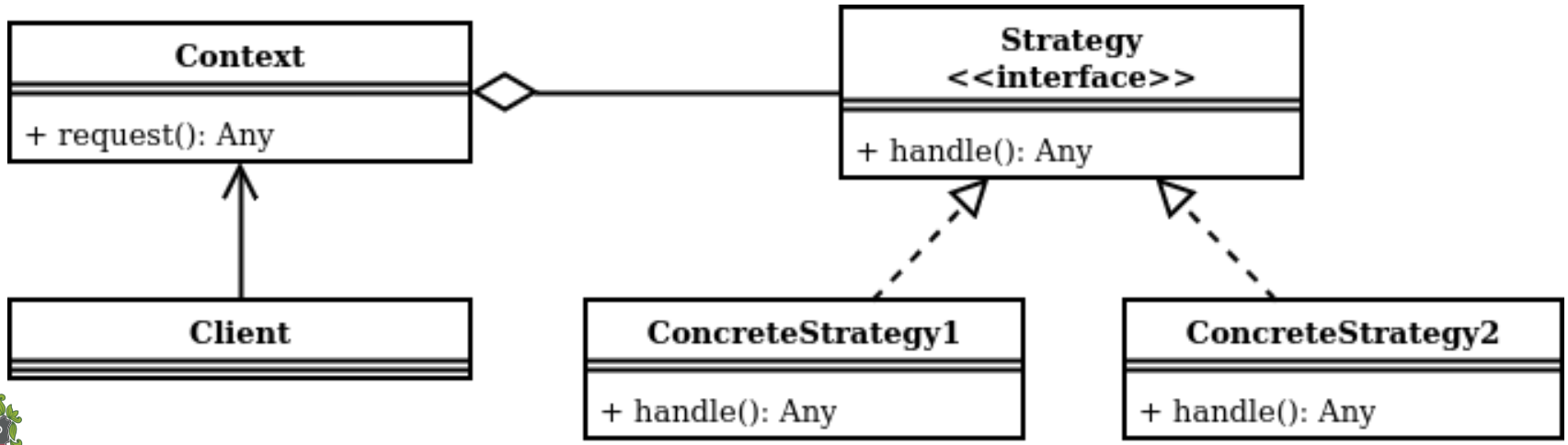
- The **Memento** pattern is a behavioral pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.
- The **Memento** pattern is used when you want to provide the ability to restore an object to its previous state (`undo`).
- The **Memento** pattern is used when you want to provide a rollback mechanism in case of errors or exceptions.




UNIVERSIDAD NACIONAL DE COLOMBIA

# Strategy Pattern — Concepts

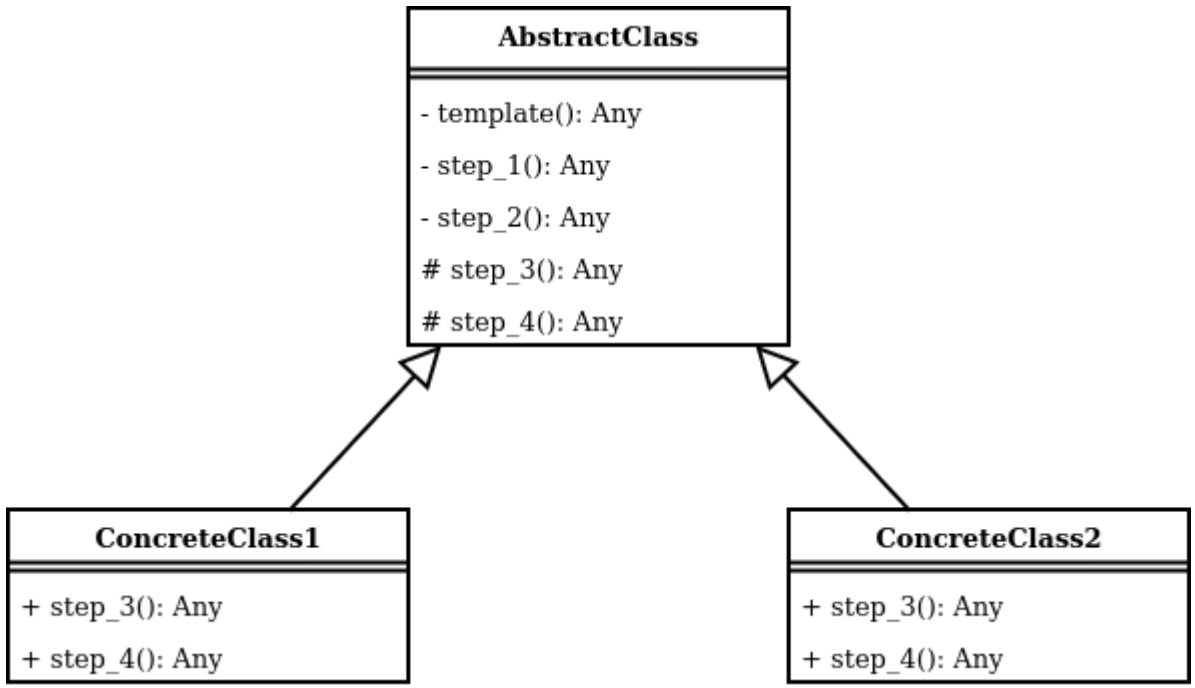
- The **Strategy** pattern is a behavioral pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.
- The **Strategy** pattern is used when you want to define a class that will have one behavior that is similar to other behaviors in a list.
- The **Strategy** pattern is used when you need to use **one** of several behaviors dynamically.




UNIVERSIDAD NACIONAL DE COLOMBIA

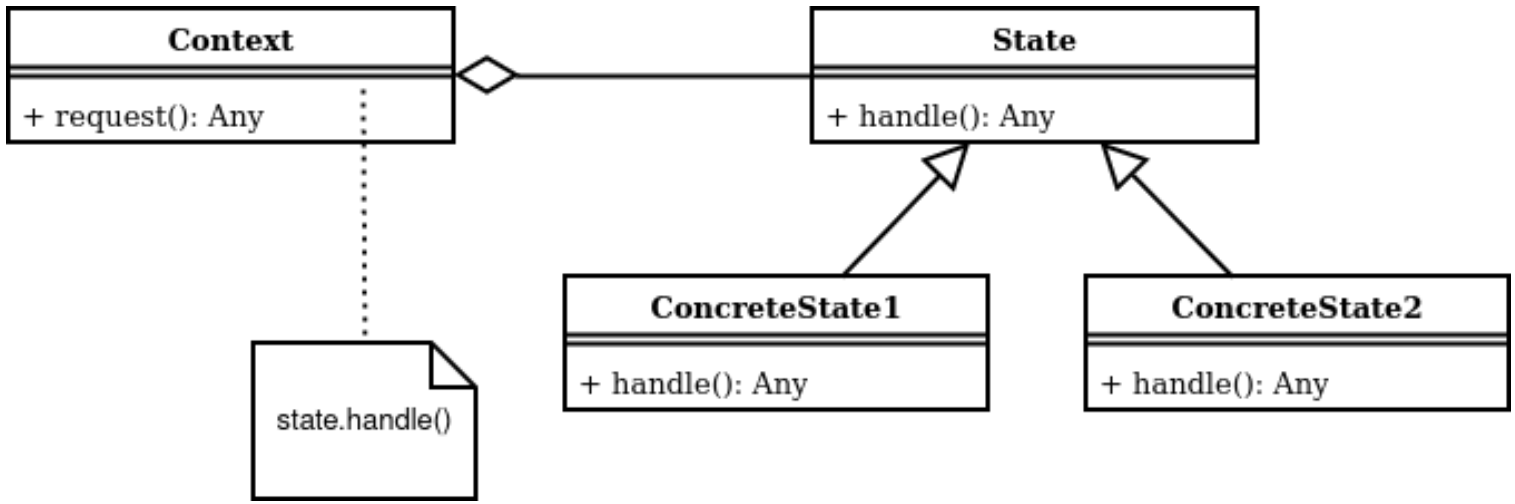
# Template Pattern — Concepts

- The **Template** pattern is a behavioral pattern that defines the program skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.
- The **Template** pattern is used when you want to let clients extend only specific steps of an algorithm, but **not the whole algorithm** or its structure.



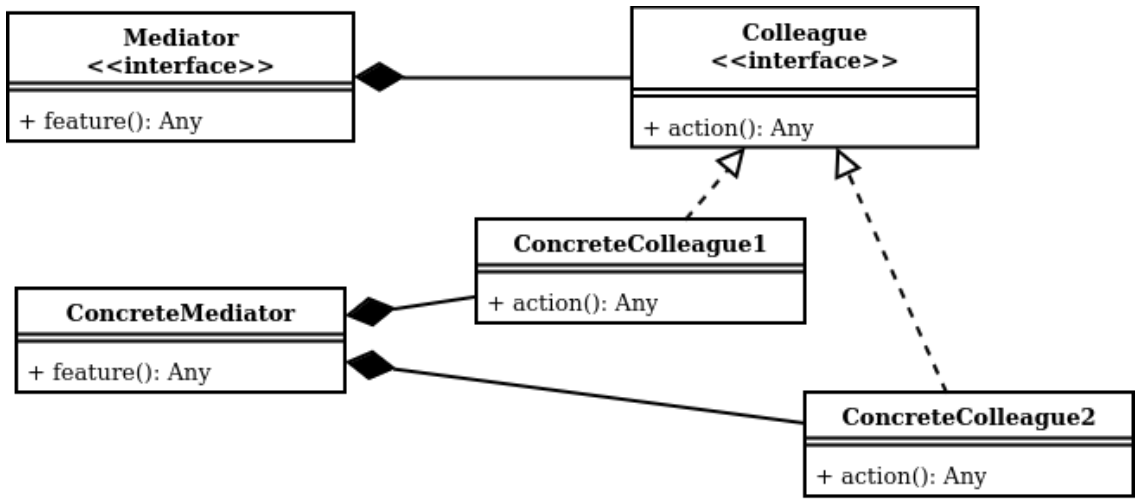
# State Pattern — Concepts

- The **State** pattern is a behavioral pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.
- The **State** pattern is used when you want to have an object that behaves as if it were an instance of a different class when its internal state changes.
- The **State** pattern is used when you want to avoid a large number of conditional statements in your code.



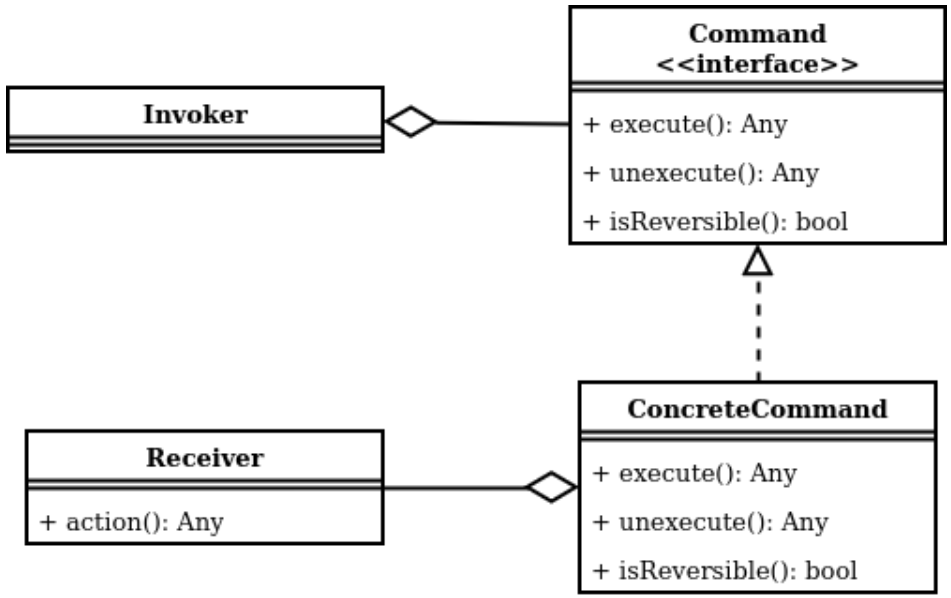
# Mediator Pattern — Concepts

- The **Mediator** pattern is a behavioral pattern that lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.
- The **Mediator** pattern is used when you want to reduce the number of dependencies between your classes.
- The **Mediator** pattern is used when you want to simplify the communication between objects in a system.



# Command Pattern — Concepts

- This pattern is pretty simple, it just attempts to **convert** the **interface** of a class into **another interface** clients expects.
- It is useful when you want to **reuse** an existing class, but its **not compatible** with the rest of your code, or at least where you need it.
- In increases compatibility, and lets define an architecture based on **interfaces** and not on concrete classes.




UNIVERSIDAD NACIONAL DE COLOMBIA

# Chain of Responsability Pattern — Concepts

- The **Chain of Responsibility** pattern is a behavioral pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to **process the request** or **to pass it along the chain**.
- The **Chain of Responsibility** pattern is used when you want to give more than one object a chance to handle a **request**.
- The **Chain of Responsibility** pattern is used when you want to pass a request to one of several objects **without** specifying the receiver explicitly.

# Chain of Responsability Pattern — Classes Structure

A lot of quality reviewers are needed to approve a **high quality product**.

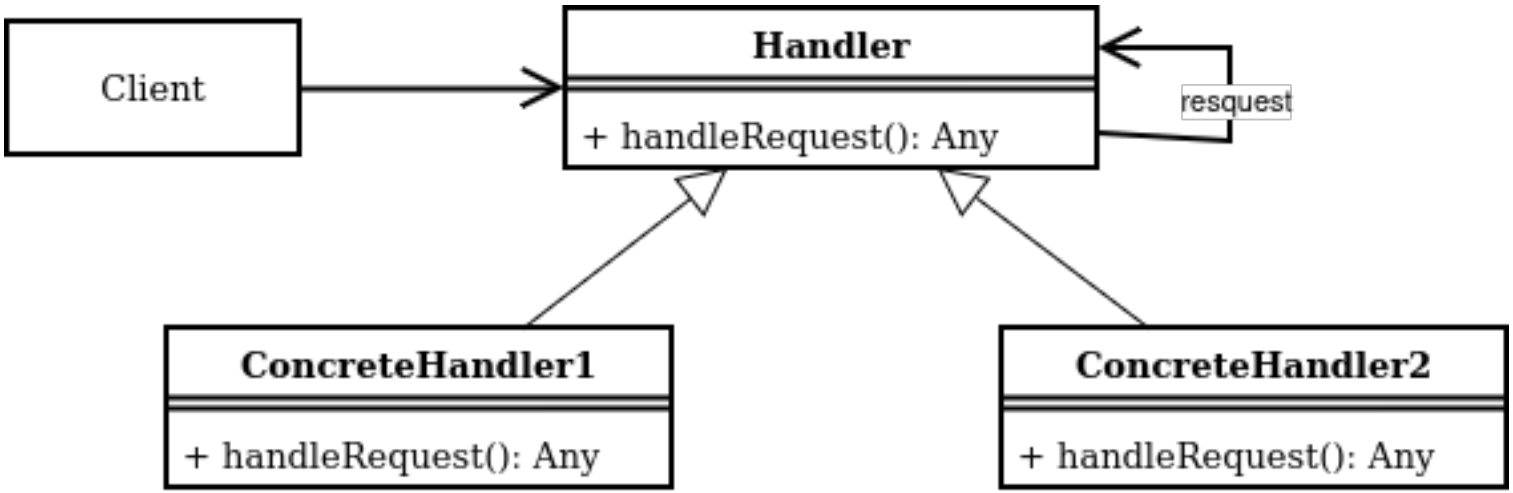



Figure: Chain of Responsability Pattern Class Diagram

Demo time!

# Observer Pattern — Concepts

- The **Observer** pattern is a behavioral pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.
- The **Observer** pattern is used when you need many other objects to receive an update when another object changes.
- The **Observer** pattern is used when an object should be able to notify other objects without making assumptions about who these objects are.

# Observer Pattern — Classes Structure

When you have a lot of eyes looking at you, you are an **observer**.

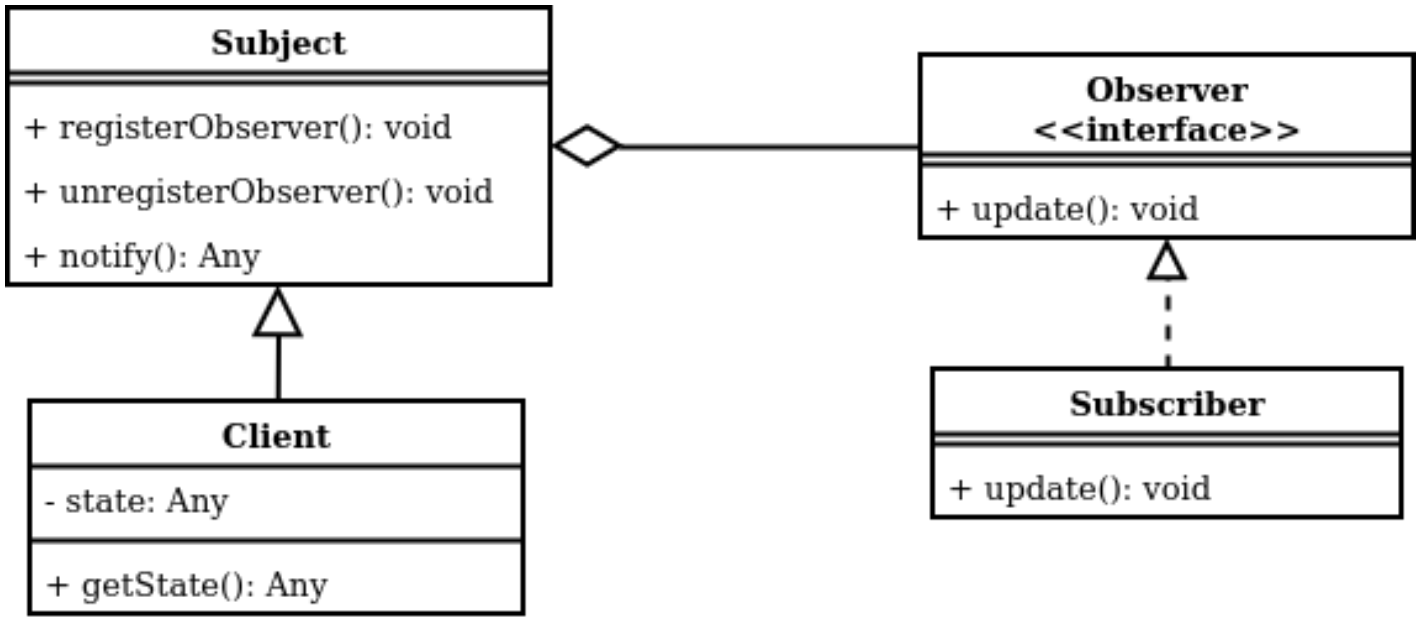



Figure: Observer Pattern Class Diagram

Demo time!

# Conclusions

- **Behavioral Patterns** are a set of *patterns* that focus on how objects distribute responsibilities among them.
- **Behavioral Patterns** are *used* when you want to `provide` a standard way to iterate over a collection, save and restore the previous state of an object, define a family of algorithms, *alter an object's behavior* when its internal state changes, reduce chaotic dependencies between objects, turn a request into a stand-alone object, define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.
- **Behavioral Patterns** are not recommended when you have a system that doesn't change, or you have a system that doesn't have a lot of objects.

# Outline



# Bad Coding

- **Bad Coding** is a software design problem that states that the code is not well written.
- If the *software* has **bad coding**, it is not maintainable and extensible.
- **Spaghetti Code** is a `bad coding` that is difficult to understand and maintain.
- **Bad practices** as copy-paste code, hardcoded values, and magic numbers are *bad coding*.

# Code Quality

- **Code Quality** is a process to validate that the code is well written.
- **Metrics** as code coverage, cyclomatic complexity, and code smells are used to measure the code quality.
- Code Review is a process to validate that the code is well written by another developer.
- Unit Testing is a process to validate that a small fragment of code is working as expected

# Stupid Deployments!

# Anti—Patterns

- **AntiPatterns** are bad practices in software design.
- An **AntiPattern** is a pattern that is *commonly used* but is ineffective and counterproductive.
- **AntiPatterns** are used to *identify* and fix bad practices in software design.
- **Techniques** to avoid `AntiPatterns` are refactoring, code review, and unit testing.

# Identify and Fix Code Smells

- **Identify Code Smells** is a process to find the bad coding in the software.
- **Fix Code Smells** is a process to correct the bad coding in the software.
- To *identify* and *fix* code smells, the software should be refactored.
- **Refactoring** is a process to improve the software without changing the behavior. A good book is *Refactoring: Improving the Design of Existing Code*, by Martin Flower.
- Techniques like code review and unit testing are used to *identify* and *fix* code smells.
- **Linters** and **static analysis tools** are used to identify and fix code smells.

# Examples of Code Smells I

- **Comments** are used to explain the code. It could be a code smell because the code maybe is not self-explanatory. Should have a equilibrium of comments.
- **Long Methods** and **Long Classes** (`Good Classes` or `Black-Hole Classes`) are used to group the code. It could be a code smell because the method or the class maybe is doing too much. Remember: Single Responsability and Separation of Concerns.
- **Magic Numbers** are used to `hardcode` values. It could be a code smell because the value maybe is not modularized. Use constants instead.
- **Duplicated Code** is used to reuse the code, maybe in blocks of code that are similar. It could be a code smell because the code maybe is not modularized. DRY (*don't repeat yourself*) principle.

# Examples of Code Smells II

- **Dead Code** is used to keep the code that is not used. It could be a code smell because the code maybe is not maintainable. Remove the code that is not used.
- **Data Classes** are used to group the data. It could be a code smell because the class contains only data and not real functionality. Use encapsulation instead, and not just getters & setters.
- **Feature Envy** consist in a method that uses more the data of another class than its own data. It could be a code smell because it increases the coupling between the classes. Use encapsulation instead, or a design pattern like Observer.
- **Data Clumps** consist in a group of data that is used together. It could be a code smell because the data maybe is not modularized. Use encapsulation instead, or a design pattern like Composite.

# Examples of Code Smells III

- **Refused Bequest** occurs when a class inherits from another class but does not use the inherited methods. It could be a code smell because the class maybe is not modularized. Use composition instead, or a design pattern like Template.
- **Switch Statements** occurs when a class has a lot of `switch` statements. It could be a code smell because the class maybe is not modularized. Use polymorphism instead, or a design pattern like Strategy.
- **Long Parameter List** consists in a method that has a lot of parameters. It could be a code smell because the method maybe is doing too much or is hard to call. Use parameter objects instead.
- **Divergent Change** occurs when a class is changed for different reasons. It could be a code smell because the class maybe is not modularized. Use composition instead, or a design pattern like Strategy.

# Examples of Code Smells IV

- **Shotgun Surgery** is a common problem in software design. It occurs when a change in a class requires changes in many other classes. It could be a code smell because the class maybe is not modularized. Use composition instead, or a structural design pattern.
- **Innapropiate Intimacy** occurs when a class has a lot of dependencies with other classes. It could be a code smell because the class maybe is not modularized. Use composition instead, or a design pattern as proxy. Remember the Principle of Least Knowledge.
- **Message Chains** violates the Law of Demeter. It occurs when a class calls a method of another class that calls a method of another class, and so on. It could be a code smell because the class maybe is not modularized. Use encapsulation instead, or a design pattern like Observer.

# Examples of Code Smells V

- **Primitive Obsession** consists in the use of primitive types instead of objects. It could be a code smell because the code maybe is not using right obstractions. Use abstract types instead.
- **Speculative Generality** consists in the use of design patterns that are not needed, or to create interfaces thinking maybe those could be useful in the future. It could be a code smell because the code maybe is not modularized. Use design patterns only when needed.

# Outline



# Thanks!

# Questions?