

# BEHAVIORAL PATTERNS

## Software Modeling

Author: Eng. Carlos Andrés Sierra, M.Sc.  
[carlos.andres.sierra.v@gmail.com](mailto:carlos.andres.sierra.v@gmail.com)

Computer Engineer  
Lecturer  
Universidad Distrital Francisco José de Caldas

2024-I



# Outline

## 1 Introduction

## 2 Patterns

- Iterator
- Memento
- Strategy\*
- Template
- Chain of Responsibility\*
- State
- Mediator
- Command\*
- Observer\*

## 3 Conclusions



# Outline

## 1 Introduction

## 2 Patterns

- Iterator
- Memento
- Strategy\*
- Template
- Chain of Responsibility\*
- State
- Mediator
- Command\*
- Observer\*

## 3 Conclusions



# Basic Concepts of Behavioral Patterns

- **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.



# Basic Concepts of Behavioral Patterns

- **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**



# Outline

## 1 Introduction

## 2 Patterns

- Iterator
- Memento
- Strategy\*
- Template
- Chain of Responsibility\*
- State
- Mediator
- Command\*
- Observer\*

## 3 Conclusions



# Outline

## 1 Introduction

## 2 Patterns

- Iterator
- Memento
- Strategy\*
- Template
- Chain of Responsibility\*
- State
- Mediator
- Command\*
- Observer\*

## 3 Conclusions



# Iterator Concepts

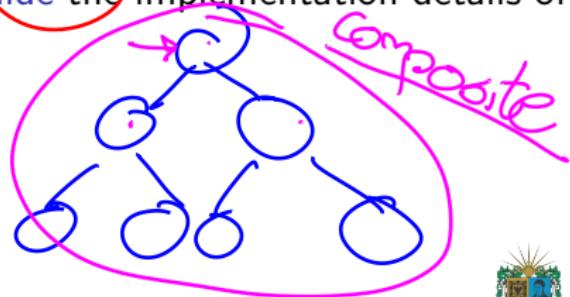
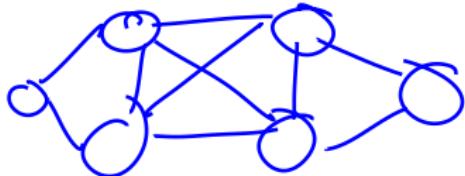
- The **Iterator** pattern is a **behavioral** pattern that allows sequential access to the elements of an **aggregate** object without exposing its underlying representation.
- The **Iterator** pattern is used when you want to provide a standard way to iterate over a collection and **hide the implementation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



## Iterator 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.



# Iterator Classes Structure

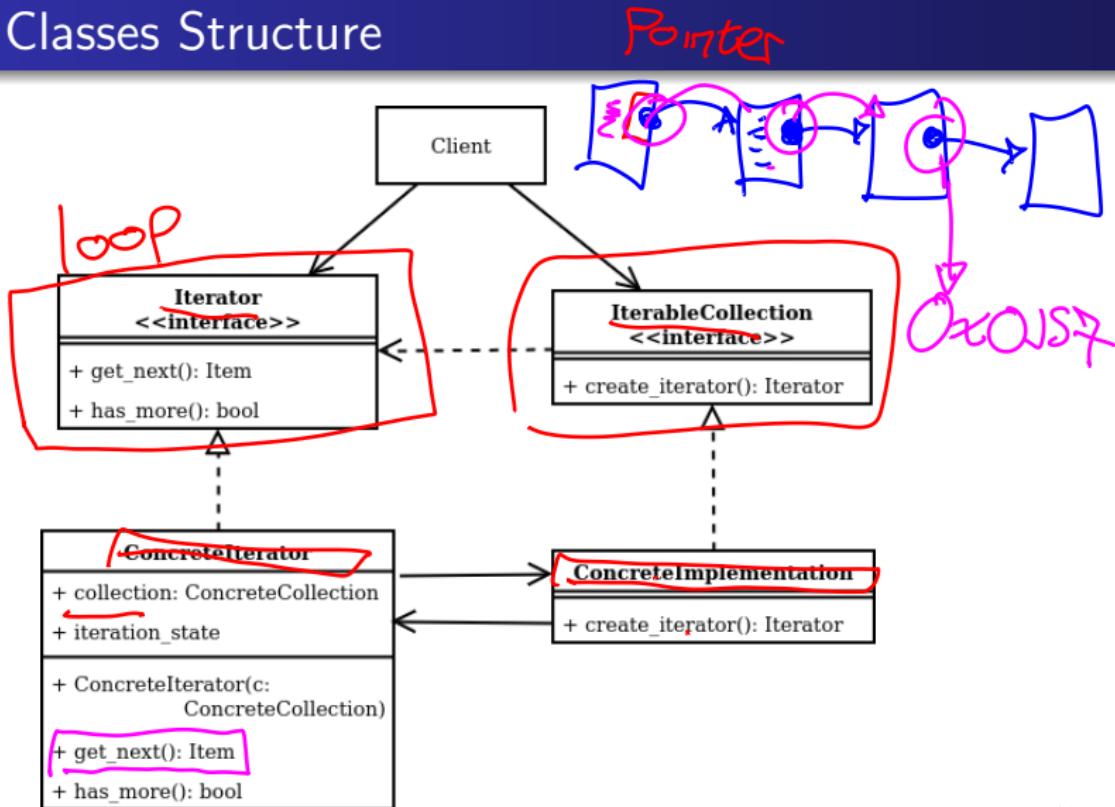
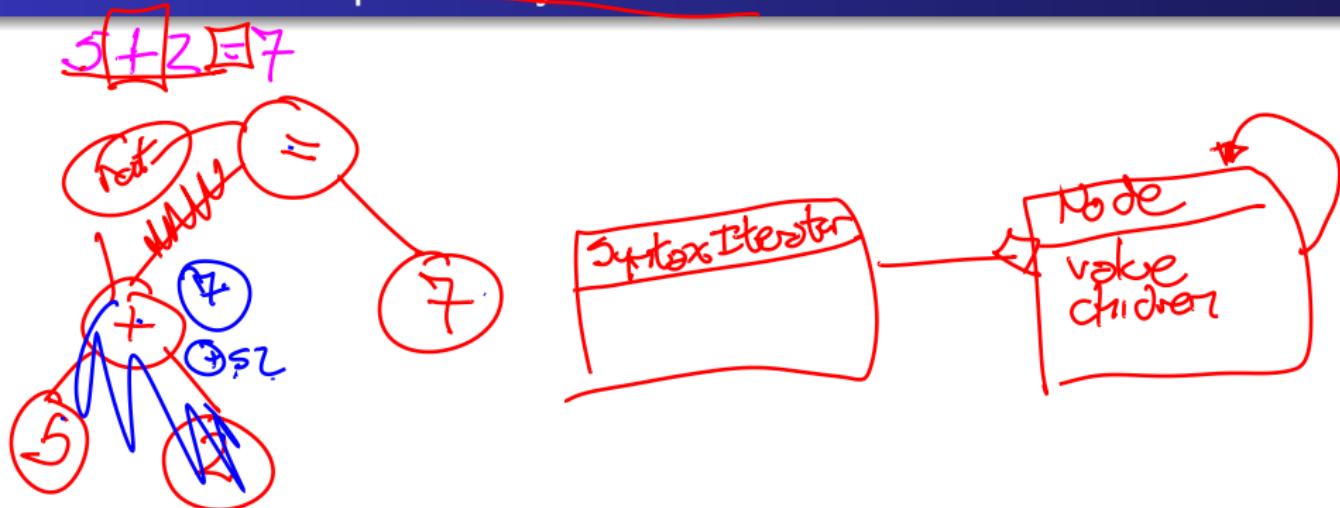


Figure: Iterator Pattern Class Diagram



# Iterator Example: A Syntax Tree



# Outline

## 1 Introduction

## 2 Patterns

- Iterator
- **Memento**
- Strategy\*
- Template
- Chain of Responsability\*
- State
- Mediator
- Command\*
- Observer\*

## 3 Conclusions



# Memento 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.



# Memento 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). (undo)
- The **Memento** pattern is used when you want to provide a rollback mechanism in case of errors or exceptions.



# Memento 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.



# Memento Classes Structure

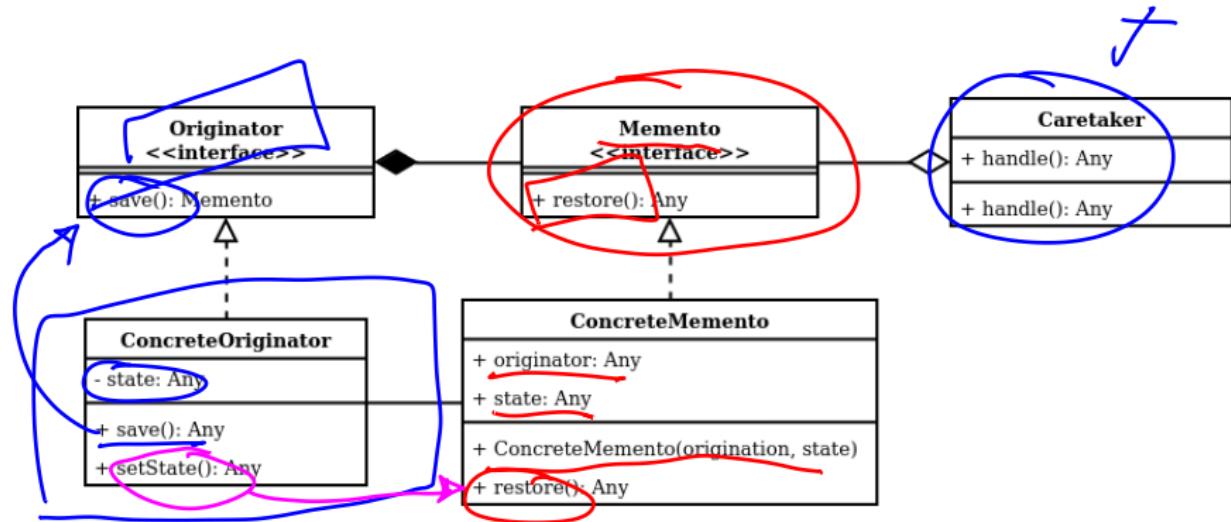
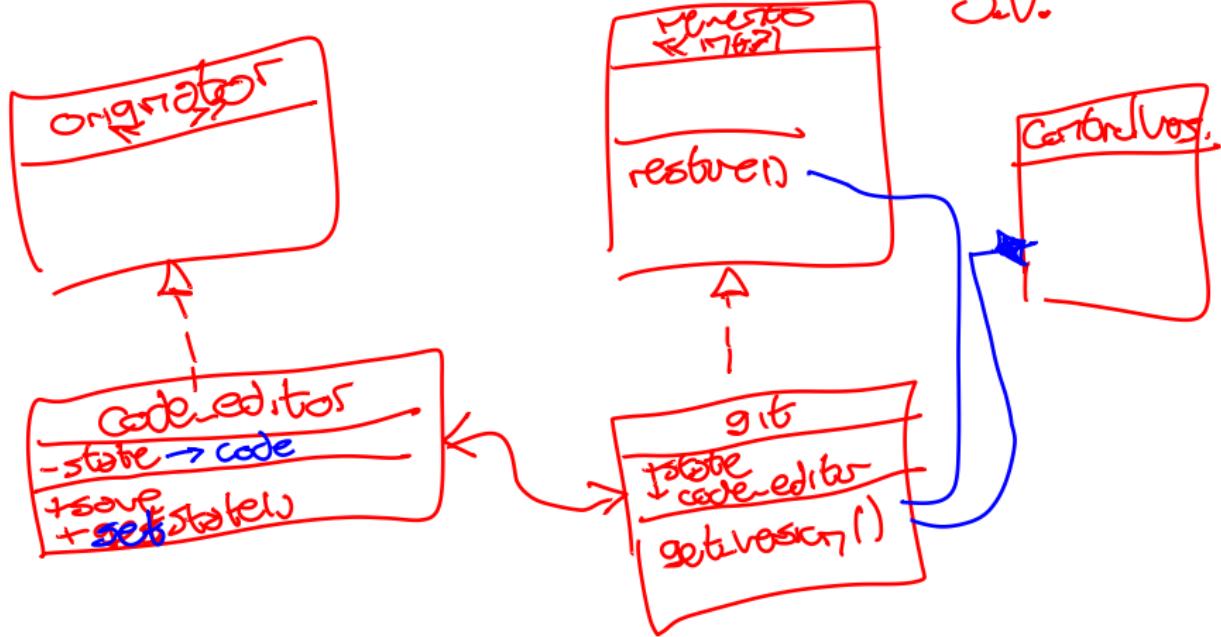


Figure: Memento Pattern Class Diagram



# Memento Example: Versioning Your Code

Git  
SVN



# Outline

## 1 Introduction

## 2 Patterns

- Iterator
- Memento
- **Strategy\***
- Template
- Chain of Responsibility\*
- State
- Mediator
- Command\*
- Observer\*

## 3 Conclusions



# Strategy 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.



# Strategy 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.



# Strategy 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**.



# Strategy Classes Structure

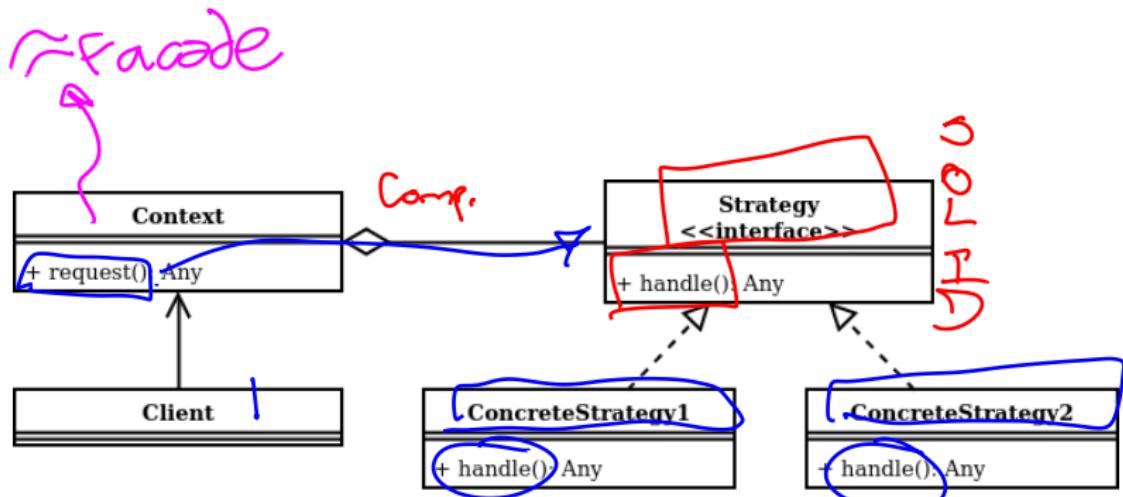
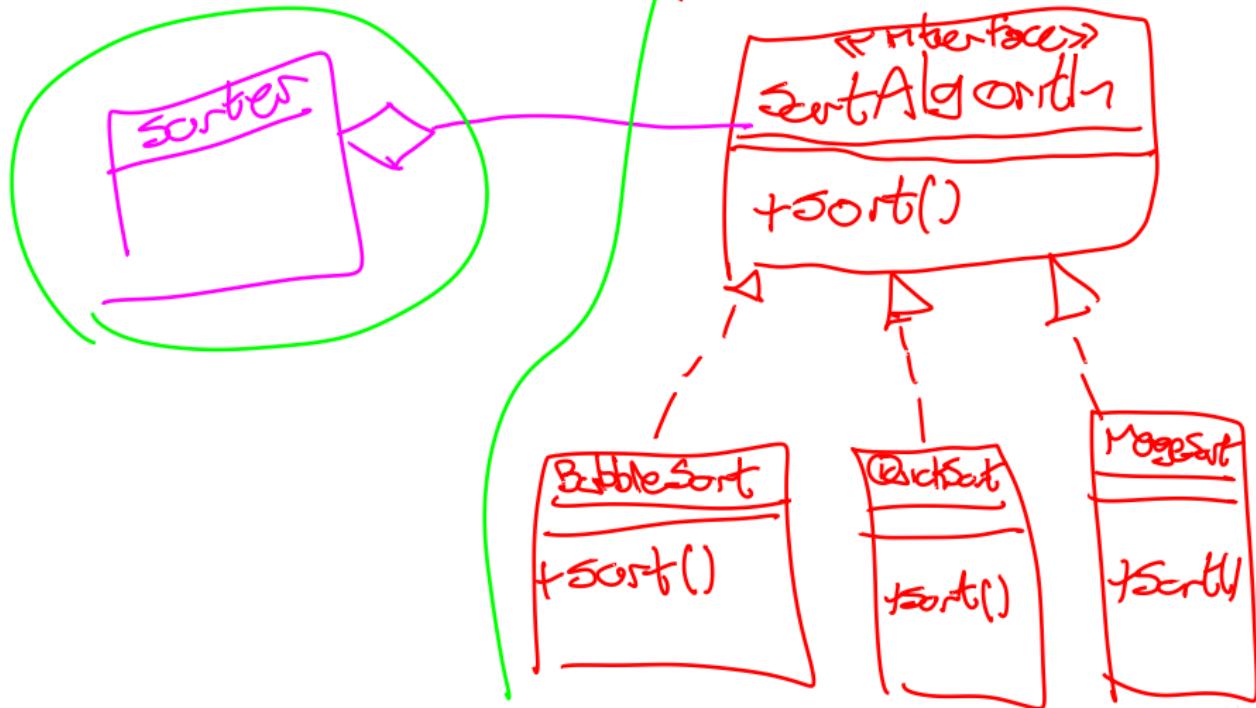


Figure: Strategy Pattern Class Diagram



## Strategy Example: Sort Algorithms



# Outline

## 1 Introduction

## 2 Patterns

- Iterator
- Memento
- Strategy\*
- **Template**
- Chain of Responsibility\*
- State
- Mediator
- Command\*
- Observer\*

## 3 Conclusions



# Template 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.
- The **Template** pattern is used when you have several classes that contain the same set of methods, but you want to avoid code duplication.



# Template 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.
- The **Template** pattern is used when you have several classes that contain the same set of methods, but you want to avoid code duplication.



# Template 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.
- The **Template** pattern is used when you have several classes that contain the same set of methods, but you want to avoid code duplication.



# Template Classes Structure

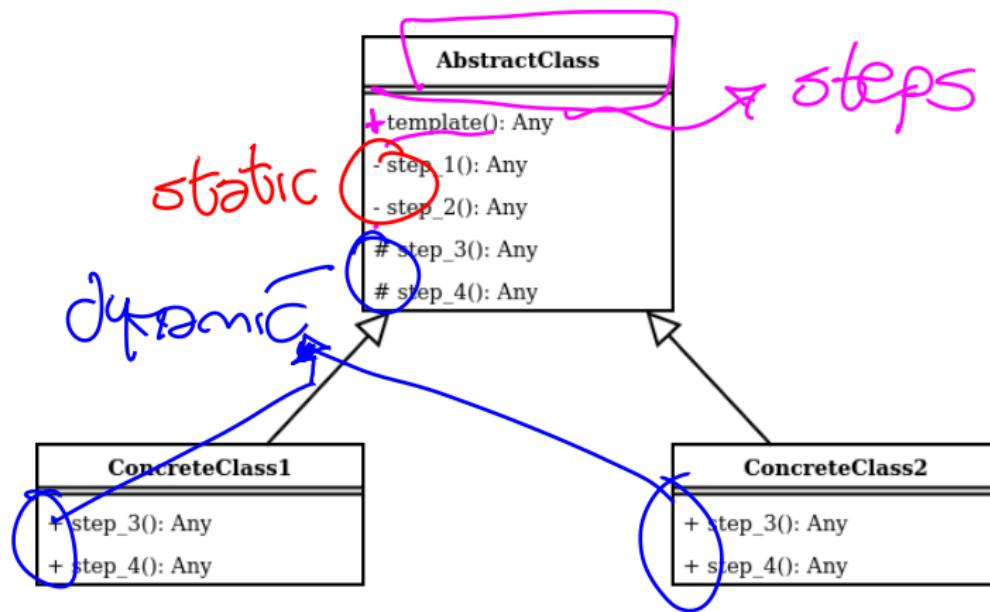
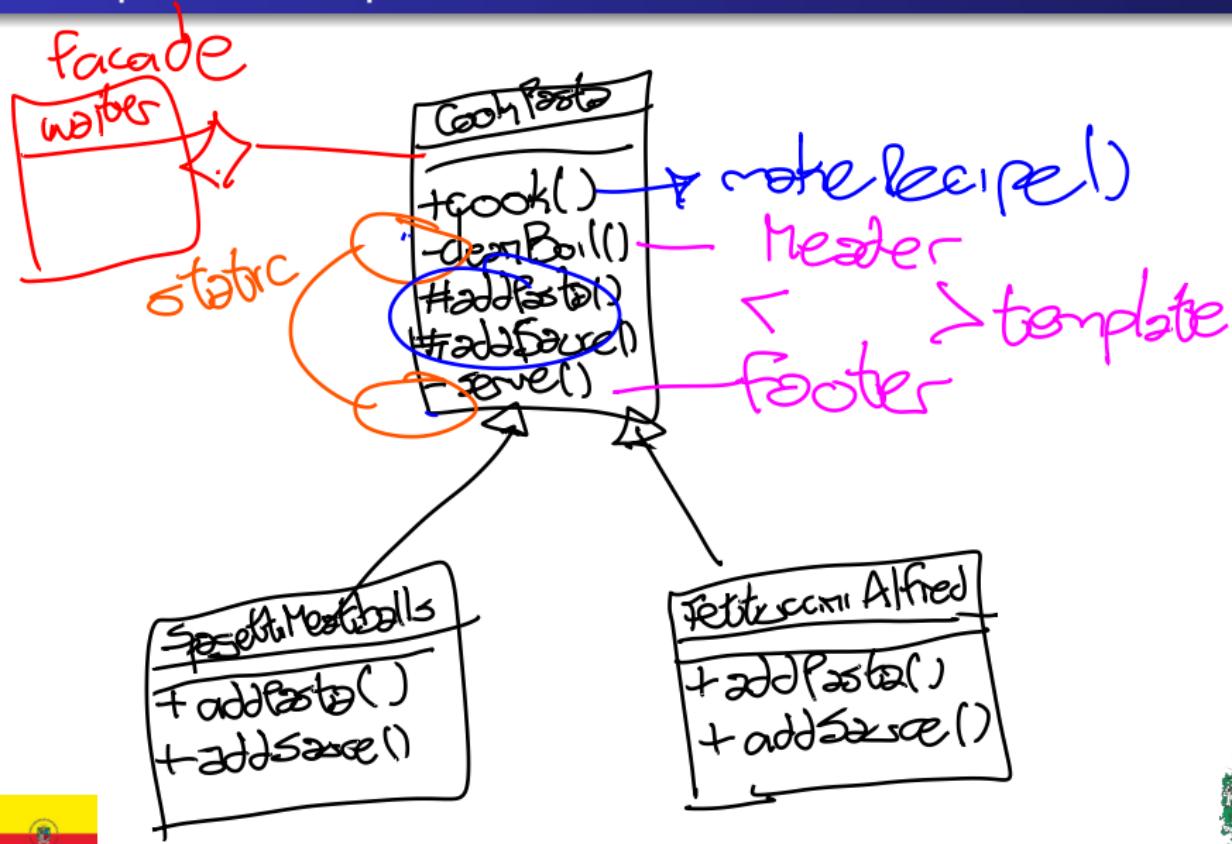


Figure: Template Pattern Class Diagram



# Template Example: Let's Cook Pasta!



# Outline

## 1 Introduction

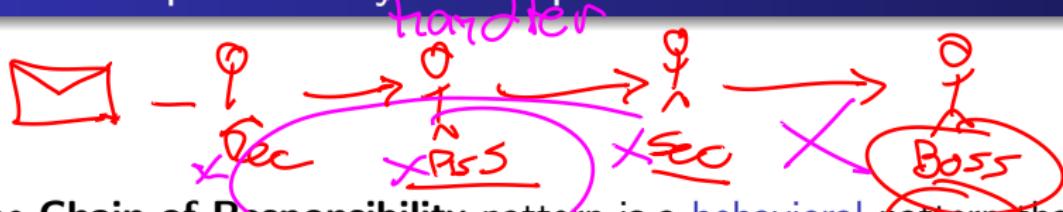
## 2 Patterns

- Iterator
- Memento
- Strategy\*
- Template
- **Chain of Responsibility\***
- State
- Mediator
- Command\*
- Observer\*

## 3 Conclusions



# Chain of Responsibility 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 Responsibility 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 Responsibility 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 Classes Structure

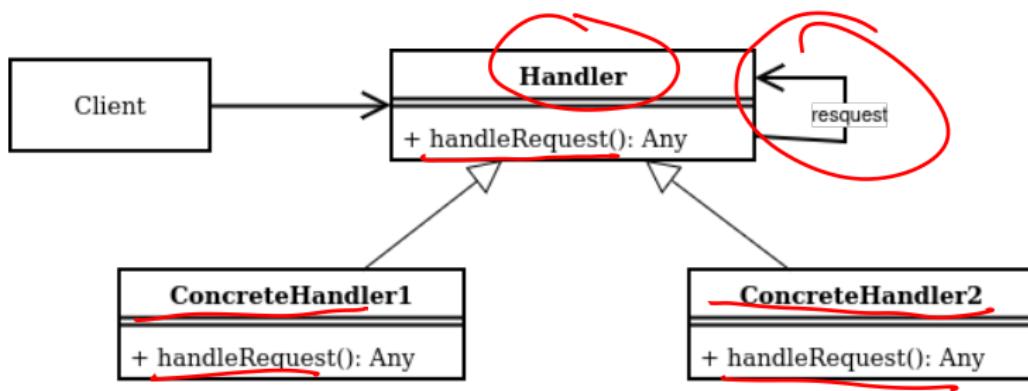
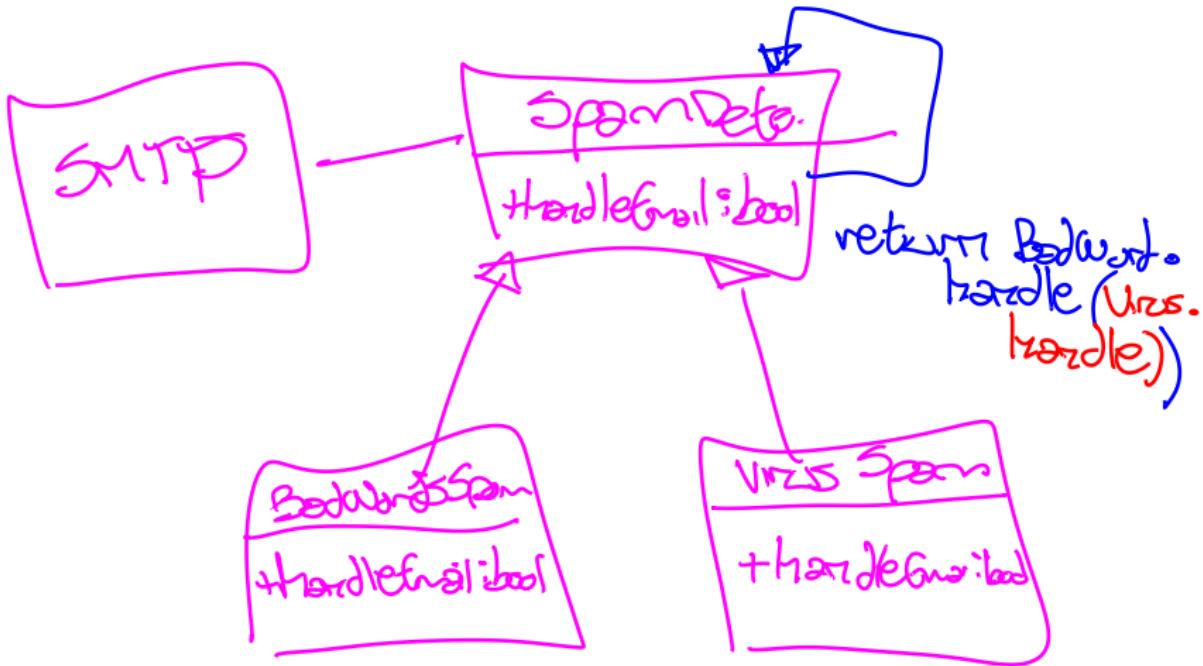


Figure: Chain of Responsibility Pattern Class Diagram



# Chain of Responsability Example: Filter an Email



# Outline

## 1 Introduction

## 2 Patterns

- Iterator
- Memento
- Strategy\*
- Template
- Chain of Responsibility\*
- State
- Mediator
- Command\*
- Observer\*

## 3 Conclusions



# State 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.



# State 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.



# State 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.



# State Classes Structure

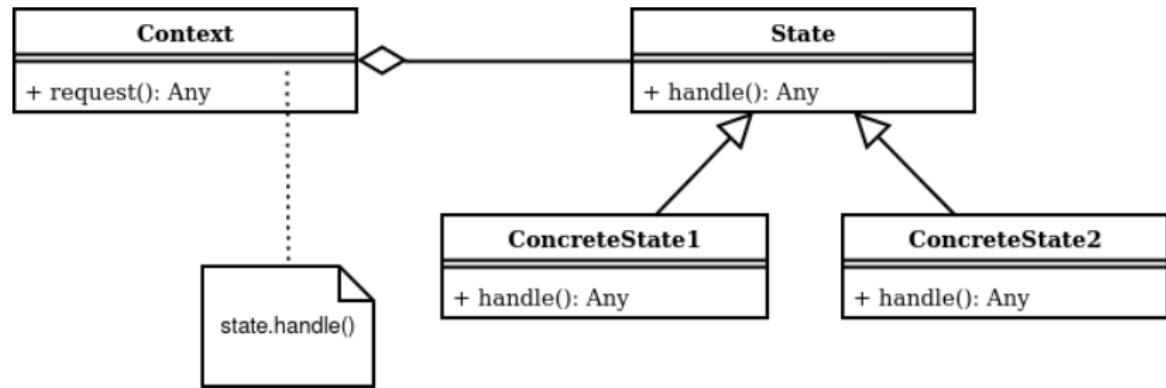


Figure: State Pattern Class Diagram



# State Example: Vending Machine



# Outline

## 1 Introduction

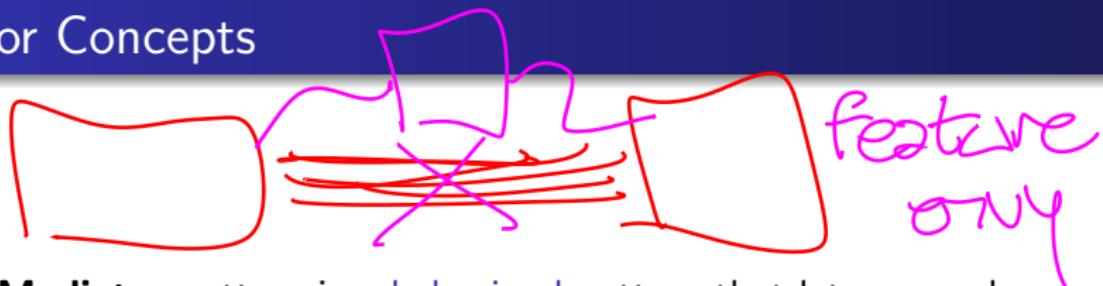
## 2 Patterns

- Iterator
- Memento
- Strategy\*
- Template
- Chain of Responsibility\*
- State
- **Mediator**
- Command\*
- Observer\*

## 3 Conclusions



# Mediator 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.



# Mediator 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.



# Mediator 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.



# Mediator Classes Structure

Facade

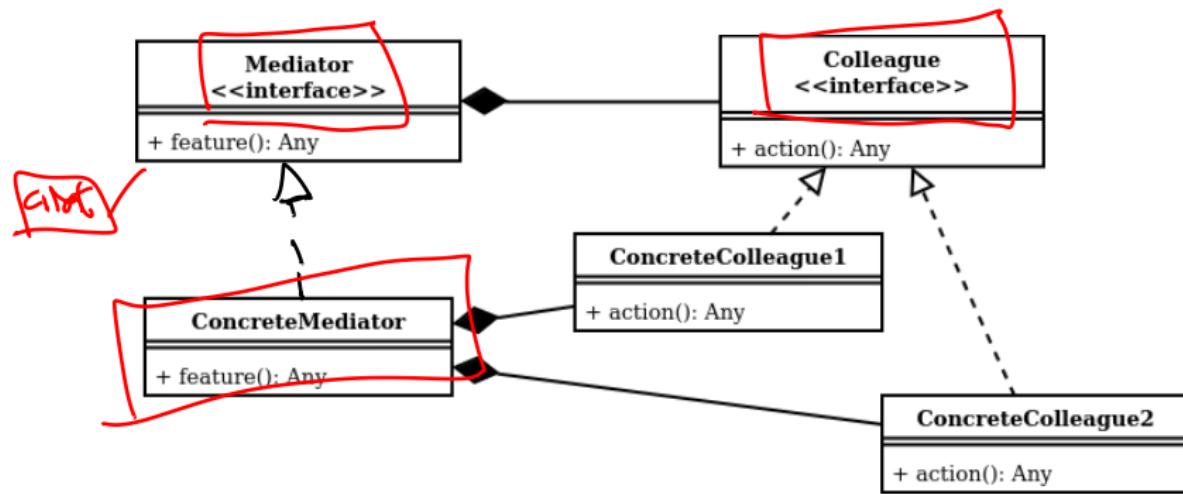


Figure: Mediator Pattern Class Diagram



# Mediator Example: Smart Home



# Outline

## 1 Introduction

## 2 Patterns

- Iterator
- Memento
- Strategy\*
- Template
- Chain of Responsibility\*
- State
- Mediator
- **Command\***
- Observer\*

## 3 Conclusions



# Command Concepts

- The **Command** pattern is a **behavioral** pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass **requests** as a method argument, delay or queue a request's execution, and support undoable operations.
- The **Command** pattern is used when you want to parameterize objects with commands.
- The **Command** pattern is used when you want to queue operations, schedule their execution, or execute them remotely.



# Command Concepts

- The **Command** pattern is a behavioral pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass **requests** as a method argument, delay or queue a request's execution, and support undoable operations.
- The **Command** pattern is used when you want to **parameterize objects** with commands.
- The **Command** pattern is used when you want to queue operations, schedule their execution, or execute them remotely.



# Command Concepts

- The **Command** pattern is a behavioral pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass **requests** as a method argument, delay or queue a request's execution, and support undoable operations.
- The **Command** pattern is used when you want to **parameterize objects** with commands.
- The **Command** pattern is used when you want to queue operations, schedule their execution, or execute them remotely.



# Command Classes Structure

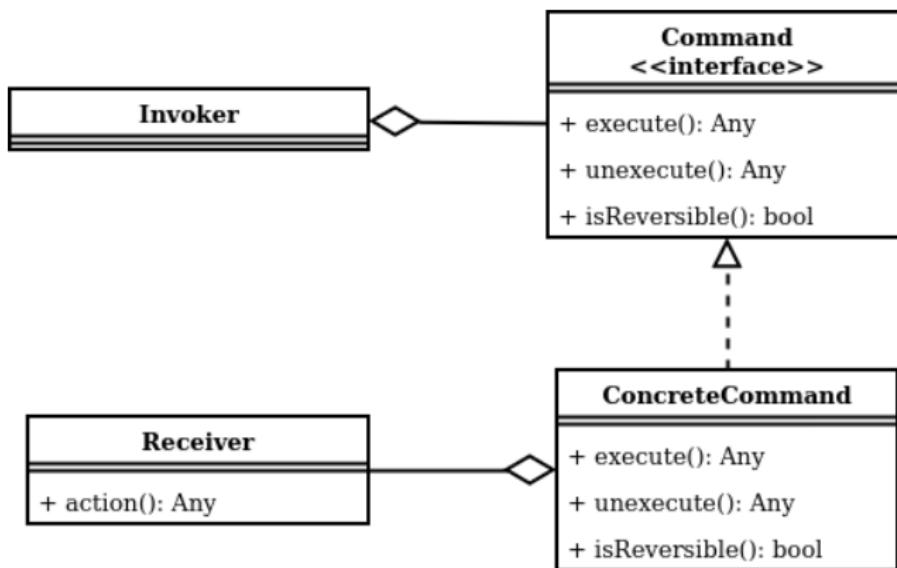


Figure: Command Pattern Class Diagram



# Command Example: Your Own Text Editor



# Outline

## 1 Introduction

## 2 Patterns

- Iterator
- Memento
- Strategy\*
- Template
- Chain of Responsibility\*
- State
- Mediator
- Command\*
- Observer\*

## 3 Conclusions



# Observer 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 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 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 Classes Structure

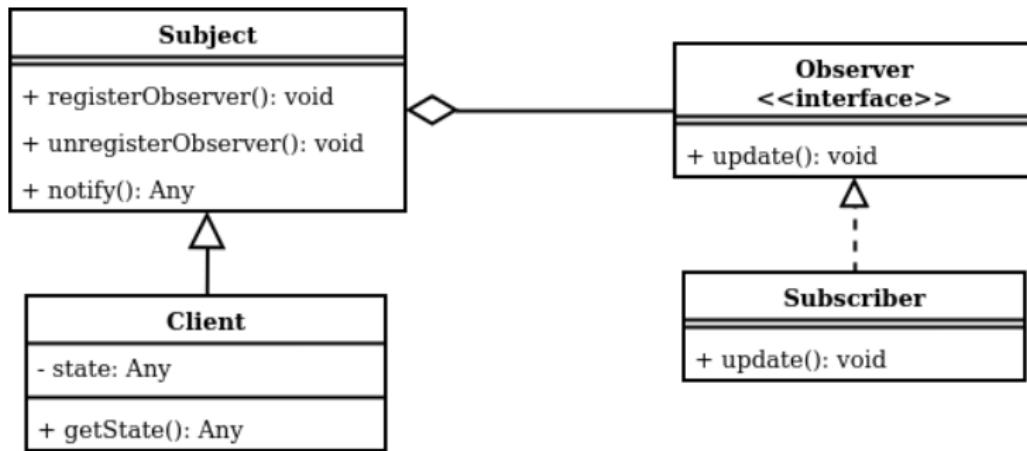


Figure: Observer Class Diagram



# Observer Example: Blogs!



# Outline

## 1 Introduction

## 2 Patterns

- Iterator
- Memento
- Strategy\*
- Template
- Chain of Responsibility\*
- State
- Mediator
- Command\*
- Observer\*

## 3 Conclusions



# 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.



# 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.



# 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

## 1 Introduction

## 2 Patterns

- Iterator
- Memento
- Strategy\*
- Template
- Chain of Responsibility\*
- State
- Mediator
- Command\*
- Observer\*

## 3 Conclusions



# Thanks!

## Questions?



Repo: <https://github.com/EngAndres/ud-public/tree/main/courses/software-modeling>

