Design Patterns

Daniel Hinojosa


What are Design Patterns?

Design Patterns

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

— Christopher Alexander

Christopher Alexander was not a software designer

- Alexander was talking about patterns in buildings and towns
- What he says is true about object-oriented design patterns.
- Our solutions are expressed in terms of objects and interfaces instead of walls and doors,
- Both kinds of patterns are a solution to a problem in a context.

Source: Design Patterns: Elements of Reusable Object-Oriented Software

Elements of a Pattern

1. The Pattern Name

- The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
- Naming a pattern immediately increases our design vocabulary. It lets us design at a higher level of abstraction. Having a vocabulary for patterns lets us talk about them with our colleagues, in our documentation, and even to ourselves.
- It makes it easier to think about designs and to communicate them and their trade-offs to others.
- Finding good names has been one of the hardest parts of developing our catalog.

Source: Design Patterns: Elements of Reusable Object-Oriented Software

2. The Problem

- The **problem** describes when to apply the pattern.
- It explains the problem and its context.
- It might describe specific design problems such as how to represent algorithms as objects.
- It might describe class or object structures that are symptomatic of an inflexible design.
- Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.

Source: Design Patterns: Elements of Reusable Object-Oriented Software

3. The Solution

- The **solution** describes the elements that make up:
  - The design
  - Their relationships
  - Responsibilities
  - Collaborations.
- The solution doesn't describe a particular concrete design or implementation,
- A pattern is a template
- Can be applied in many different situations.
- Provides an abstract description of a design problem

Source: Design Patterns: Elements of Reusable Object-Oriented Software

4. The Consequences

- The **consequences** are the results and trade-offs of applying the pattern.
- Though consequences are often unvoiced when we describe design decisions
- They are critical
  - For evaluating design alternatives
  - For understanding the costs and benefits of applying the pattern

Source: Design Patterns: Elements of Reusable Object-Oriented Software


Quick Intro to UML

UML Defined

- UML
  - Unified Modeling Language
  - Communicate designs unambiguously
  - Convey the essence of a design
  - Capture and map functional requirements to their software solutions

Source: Learning UML 2.0 by Russ Miles; Kim Hamilton Published by O'Reilly Media, Inc., 2006

UML can be used however you like:

UML as a sketch

- Make brief sketches to convey key points
- Throwaway sketches—they could be written on a whiteboard

UML as a blueprint

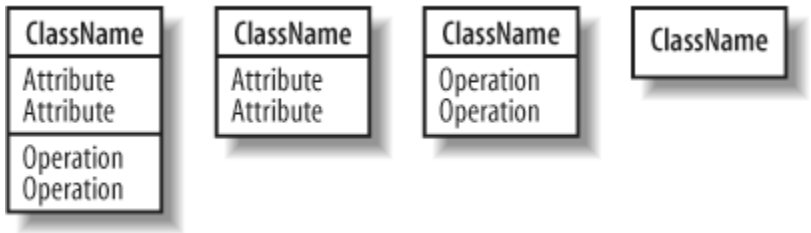
- Provide a detailed specification of a system with UML diagrams
- Would not be disposable but would be generated with a UML tool
- Generally associated with software systems and usually involves keeping models synchronized with the code

UML as a programming language

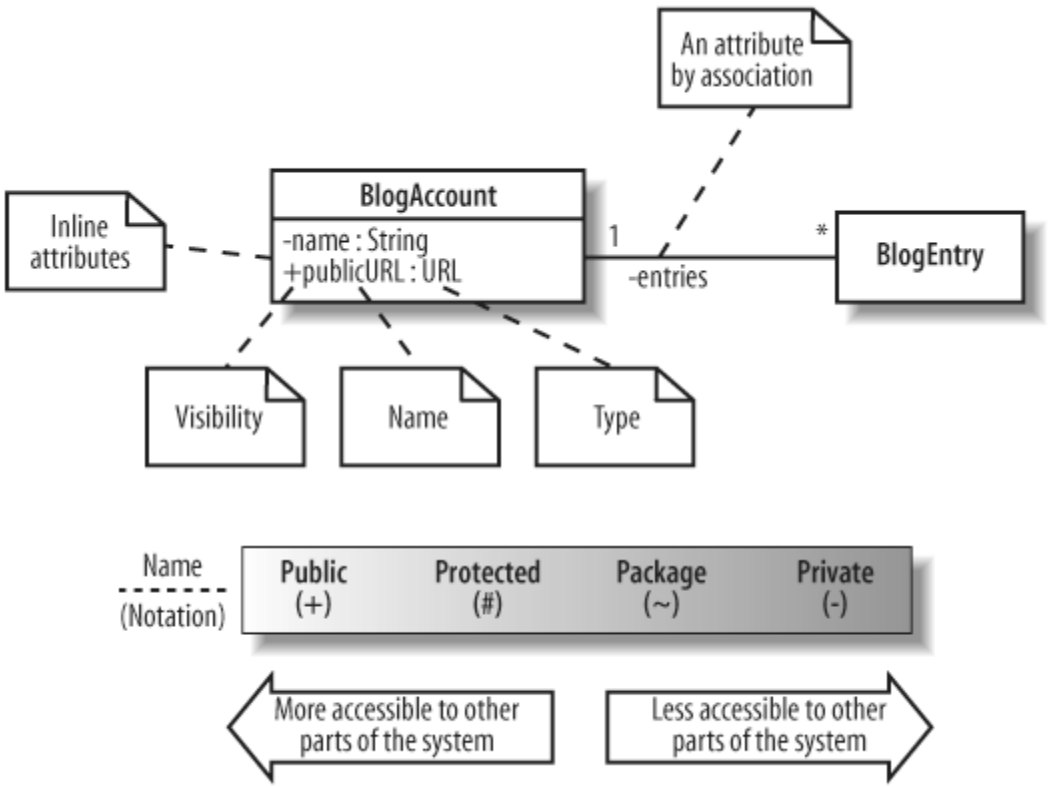
- UML model to executable code
- Meaning that every aspect of the system is modeled
- You can keep your model indefinitely and use transformations and code generation to deploy to different environments

Source: Learning UML 2.0 by Russ Miles; Kim Hamilton Published by O'Reilly Media, Inc., 2006

Classes

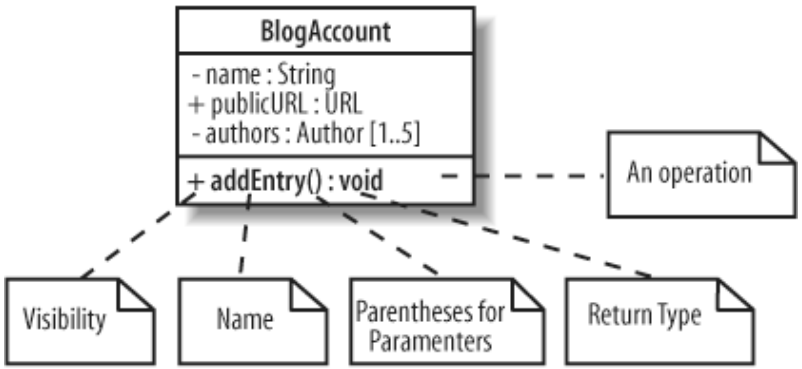

ClassName
Attribute
Attribute
Operation
Operation
ClassName
Attribute
Attribute
ClassName
Operation
Operation
ClassName

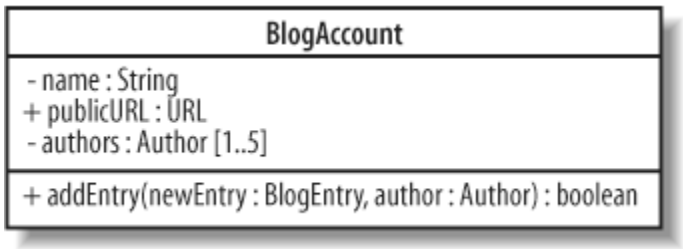
Attributes and Relations in UML



Source: Learning UML 2.0 by Russ Miles; Kim Hamilton Published by O'Reilly Media, Inc., 2006

Class Methods

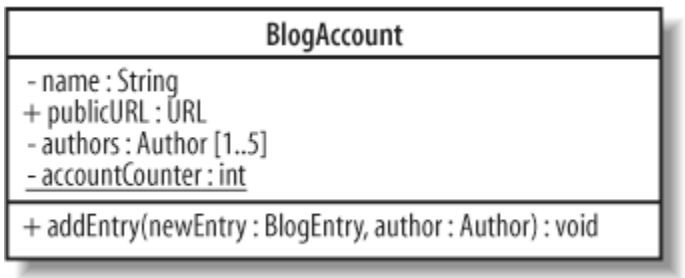




Source: Learning UML 2.0 by Russ Miles; Kim Hamilton Published by O'Reilly Media, Inc., 2006

Static Fields

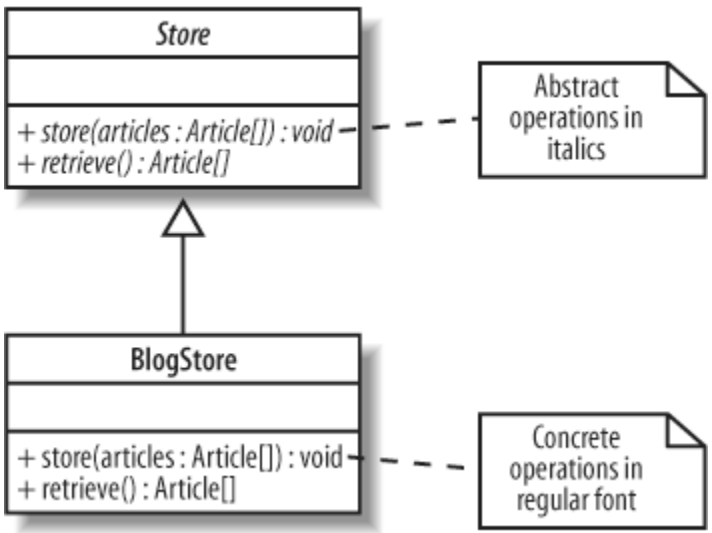
`static` is conveyed with an underline



Source: Learning UML 2.0 by Russ Miles; Kim Hamilton Published by O'Reilly Media, Inc., 2006

Abstract

`abstract` is conveyed in italics



Source: Learning UML 2.0 by Russ Miles; Kim Hamilton Published by O'Reilly Media, Inc., 2006

Interfaces

`interface` is conveyed either with the interface stereotype or the ball notation

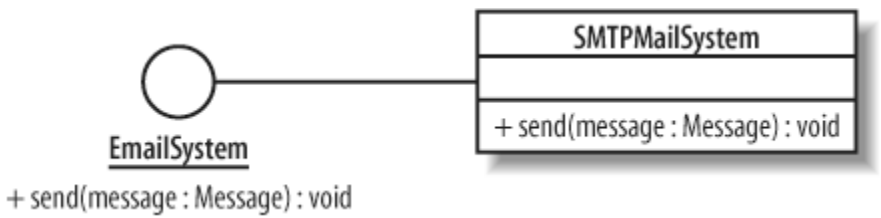


Figure 1. Ball Notation

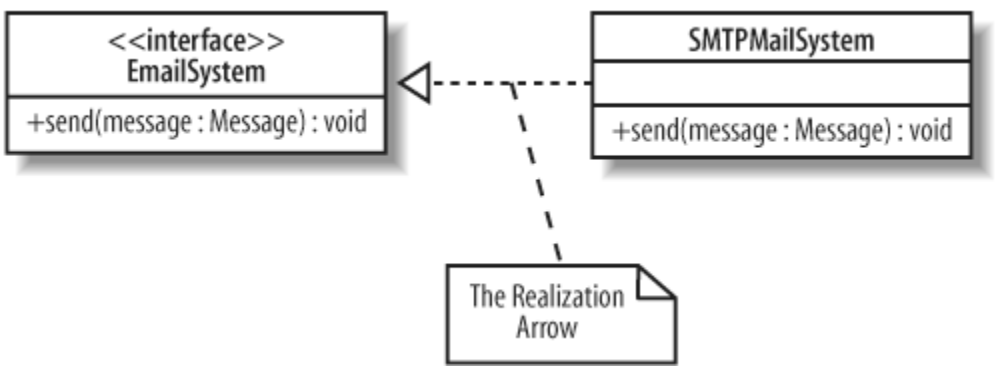
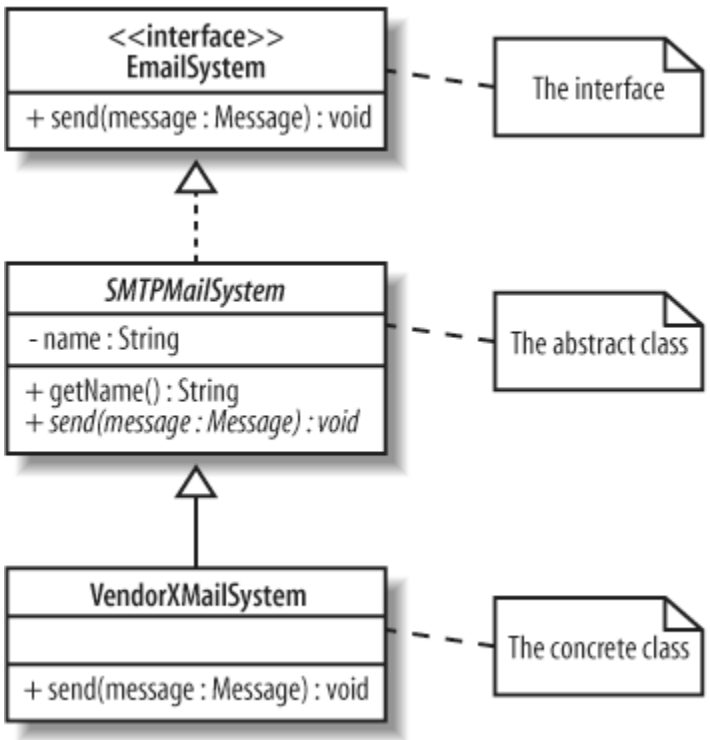


Figure 2. Stereotype Notation

Source: Learning UML 2.0 by Russ Miles; Kim Hamilton Published by O'Reilly Media, Inc., 2006

Concrete to Abstract to Interface

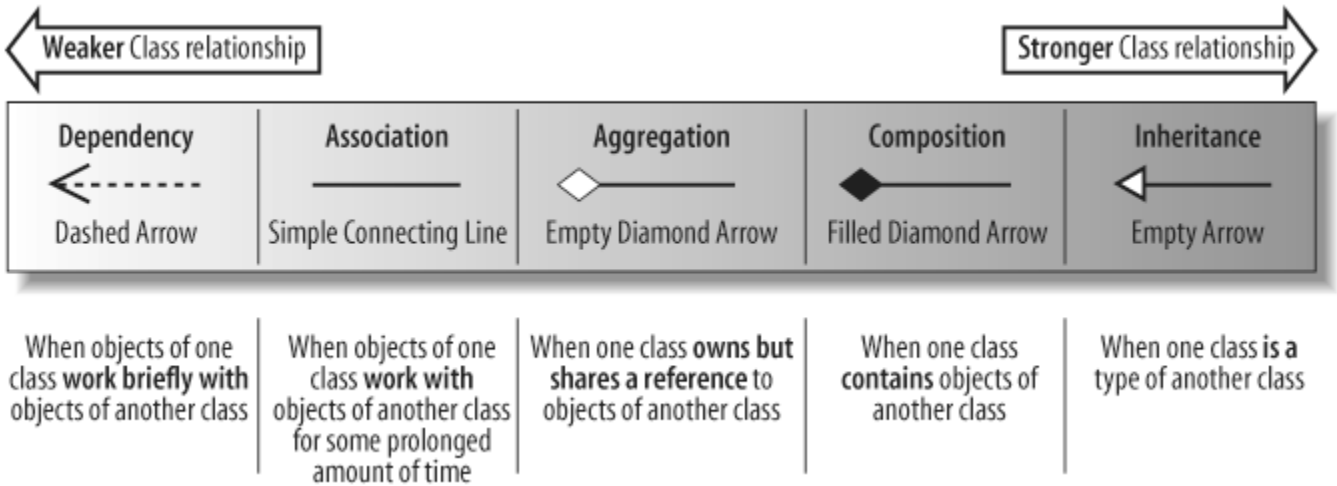
Here is a complete inheritance chain using concrete, `abstract`, and `interface`



Source: Learning UML 2.0 by Russ Miles; Kim Hamilton Published by O'Reilly Media, Inc., 2006

Dependencies

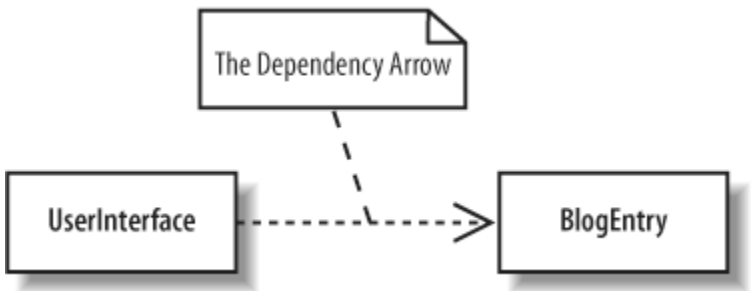
Diagram of all the different kinds of dependencies



Source: Learning UML 2.0 by Russ Miles; Kim Hamilton Published by O'Reilly Media, Inc., 2006

Use Dependency

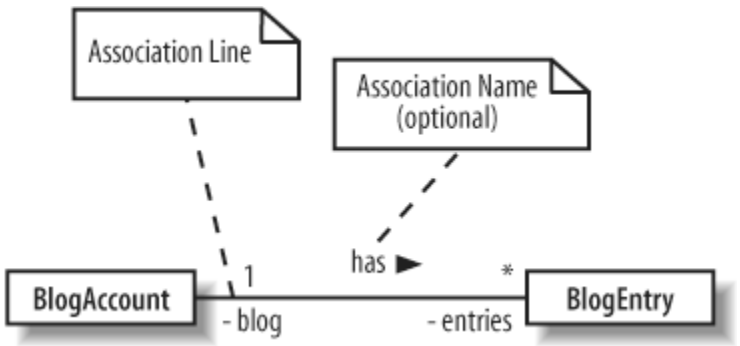
A standard dependency declares that a class needs to know about another class to use objects of that class



Source: Learning UML 2.0 by Russ Miles; Kim Hamilton Published by O'Reilly Media, Inc., 2006

Association Dependency

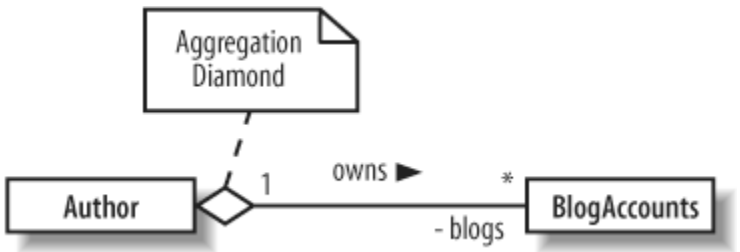
Association - A class will actually contain a reference to an object, or objects, of the other class in the form of an attribute



Source: Learning UML 2.0 by Russ Miles; Kim Hamilton Published by O'Reilly Media, Inc., 2006

Aggregation Dependency

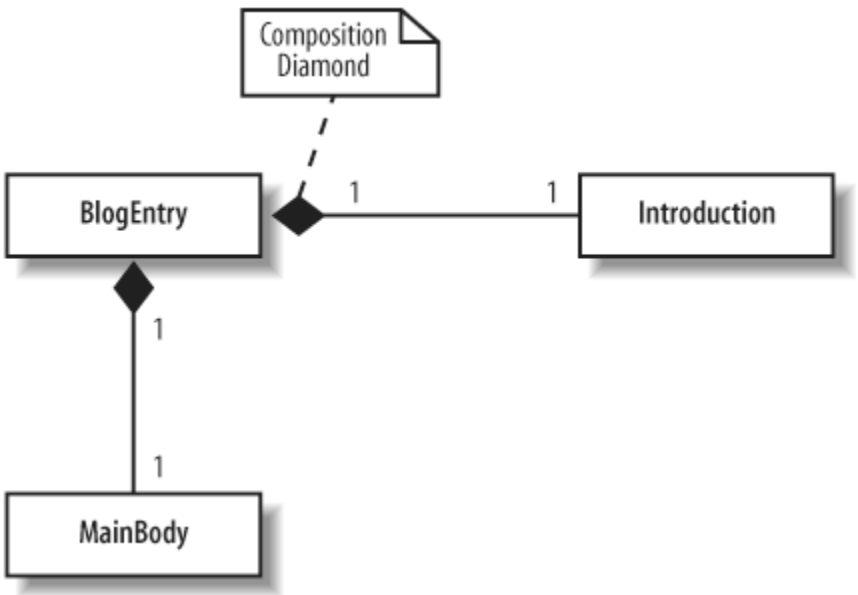
A stronger version of association and is used to indicate that a class actually owns but may share objects of another class.



Source: Learning UML 2.0 by Russ Miles; Kim Hamilton Published by O'Reilly Media, Inc., 2006

Composition Dependency

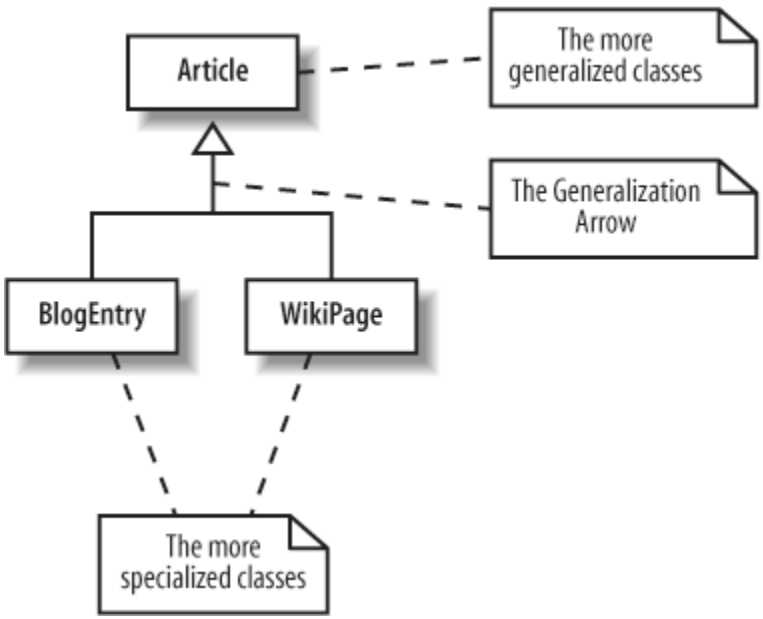
Composition - association of an element or elements that are not exposed



Source: Learning UML 2.0 by Russ Miles; Kim Hamilton Published by O'Reilly Media, Inc., 2006

Generalization

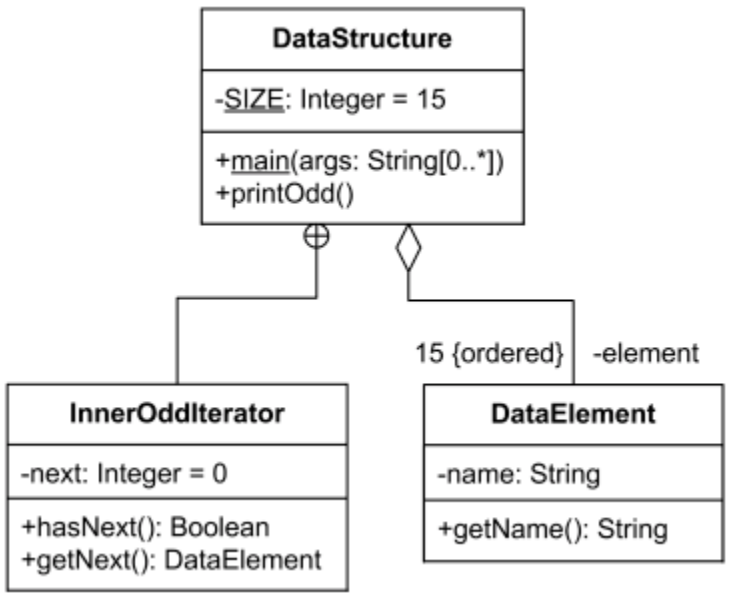
Inheritance - One type has a "is-a" relationship with another



Source: Learning UML 2.0 by Russ Miles; Kim Hamilton Published by O'Reilly Media, Inc., 2006

Inner Class Dependency

The circle with a cross is an anchor, and denotes that one is an inner-class of another class



Source: Learning UML 2.0 by Russ Miles; Kim Hamilton Published by O'Reilly Media, Inc., 2006

Software Development Techniques


Test Driven Development

The Rules

1. Quickly add a test.
2. Run all tests and see the new one fail.
3. Make a little change.
4. Run all tests and see them all succeed.
5. Refactor to remove duplication.

Kent Beck – Test Driven Development By Example 2003

Another Perspective

1. Write a failing test
2. Write code to make it pass
3. Repeat steps 1 and 2
4. Along the way, refactor aggressively
5. When you can't think of any more tests, you must be done

Neal Ford – Evolutionary Architecture and Emergent Design 2009

Purpose

- Cleaner API
- Better Testing Coverage
- Promotes design decisions up front
- Allows you and your team to understand your code
- Model the API the way you want it to look
- Means of communicating an API before implementation
- Avoids Technical Debt
- Can be used with any programming language

The Three TDD Laws

Law 1: Don’t do Production without a test

You may not write production code until you have written a failing unit test.

Clean Code 2008
— Bob Martin

Law 2: Keep Unit Tests Light

You may not write more of a unit test than is sufficient to fail

Clean Code 2008
— Bob Martin

Law 3: Don’t do more production without a test

You may not write more production code than is sufficient to pass the currently failing test.

Clean Code 2008
— Bob Martin

Demo: `FizzBuzz`

According to http://wiki.c2.com/?FizzBuzzTest

The "Fizz-Buzz test" is an interview question designed to help filter out the 99.5% of programming job candidates who can't seem to program their way out of a wet paper bag. The text of the programming assignment is as follows:

Demo: `FizzBuzz`

Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz"

Sample output

shell

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
```

Creational Patterns


Factory Method Pattern

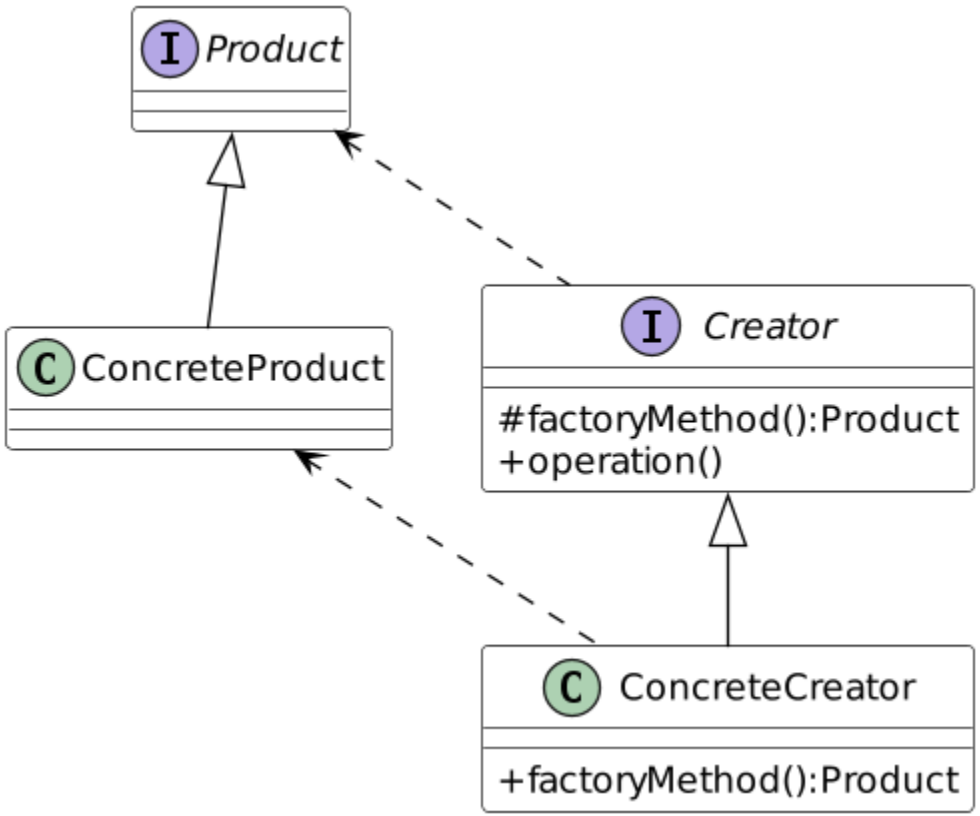
Factory Method Pattern Properties

- **Type:** Creational
- **Level:** Class

Factory Method Pattern Purpose

To define a standard method to create an object, apart from a constructor, but the decision of what kind of an object to create is left to subclasses.

Factory Method Canonical Diagram



Factory Method Ingredients

Product

The `interface` of objects created by the factory

ConcreteProduct

The implementing class of **Product**. Objects of this class are created by the **ConcreteCreator**.

Creator

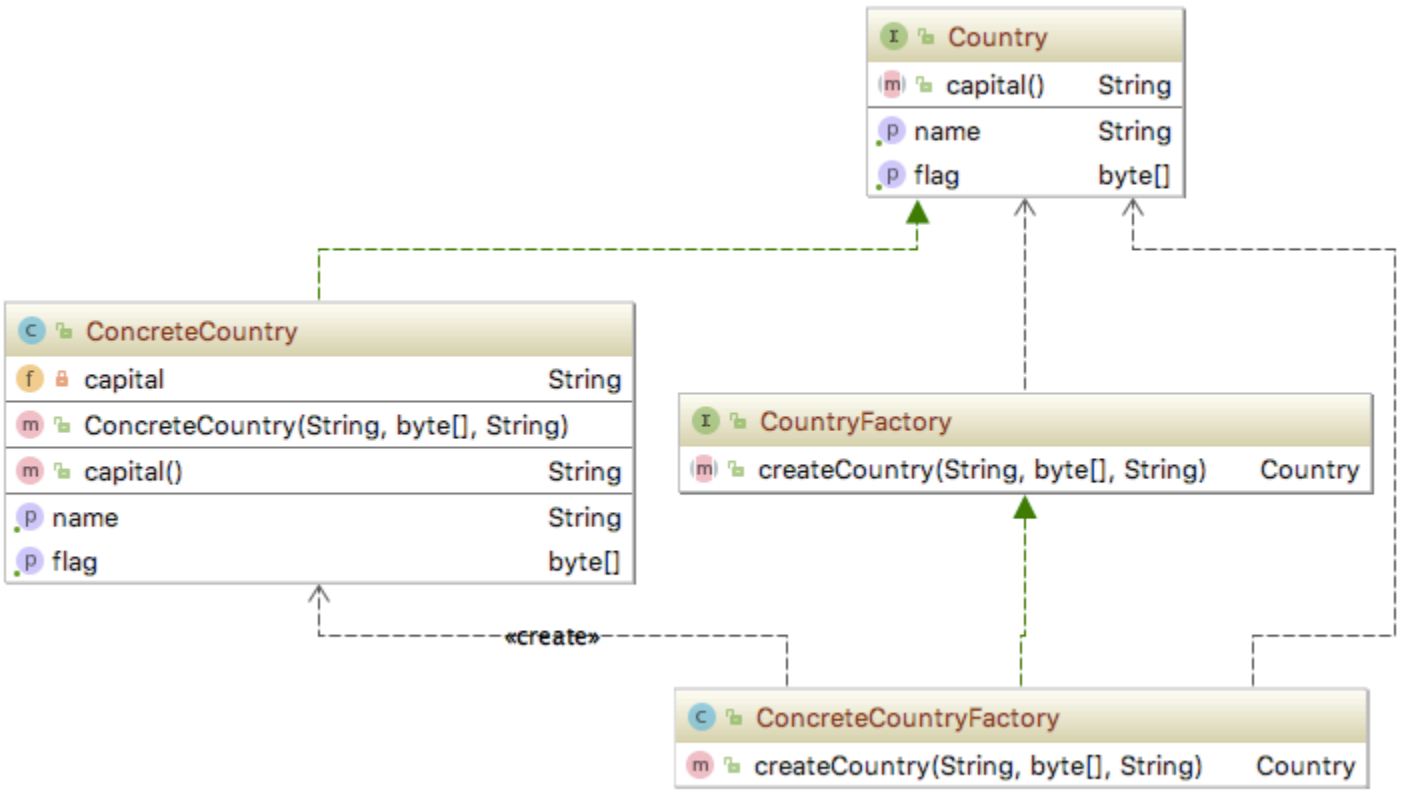
The `interface` that defines the factory methods

ConcreteCreator

The class that extends **Creator** and that provides an implementation for the factoryMethod. This can return any object that implements the **Product** interface.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Factory Method Demo Diagram



Factory Method Advantages

- Extensible
- Leave decision of specificity until later
- Subclass, not superclass, determines the kind of object to create
- You know when to create an object, but not what kind of an object.
- You need several overloaded constructors with the same parameter list, which is not allowed in Java. Instead, use several Factory Methods with different names.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Factory Method Disadvantages

- To create a new type you must create a separate subclass

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Demo: Factory Method


Builder Pattern

Builder Pattern Properties

- **Type:** Creational
- **Level:** Component

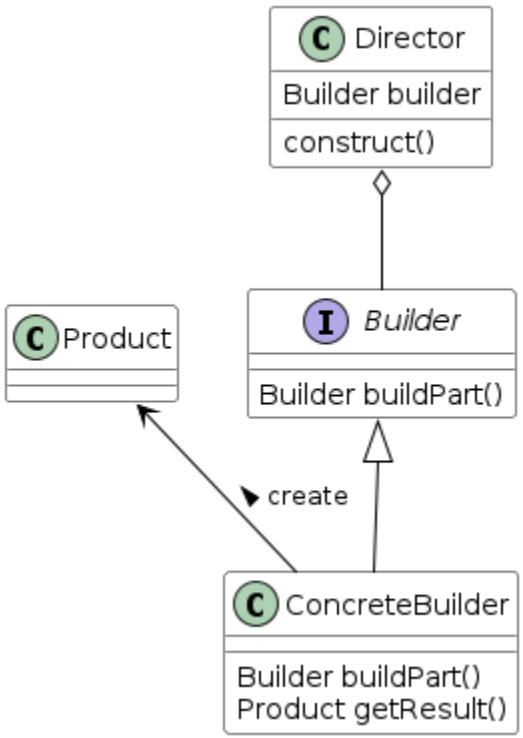
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Builder Pattern Purpose

To simplify complex object creation by defining a class whose purpose is to build instances of another class. The Builder produces one main product, such that there might be more than one class in the product, but there is always one main class.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Builder Pattern Canonical Diagram



Builder Pattern Ingredients

Director

Has a reference to an **AbstractBuilder** instance. The **Director** calls the creational methods on its builder instance to have the different parts and the Builder build.

AbstractBuilder

The interface that defines the available methods to create the separate parts of the product.

ConcreteBuilder

Implements the **AbstractBuilder** interface. The **ConcreteBuilder** implements all the methods required to create a real Product. The implementation of the methods knows how to process information from the **Director** and build the respective parts of a Product. The **ConcreteBuilder** also has either a `getProduct` method or a creational method to return the **Product** instance.

Product

The resulting object. You can define the product as either an interface (preferable) or class.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

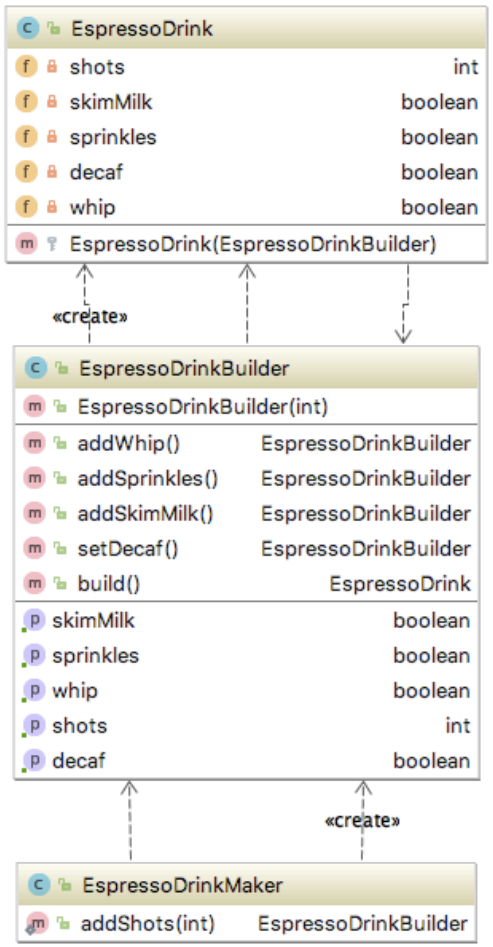
Builder Pattern Advantages

- Works well if you have complex state in an object
- Avoids complicated constructors
- Avoids complicated object graph initialization
- Works particularly well for dependencies that are difficult to setup

Builder Pattern Disadvantages

- Tight coupling in the builder and its product
- Any changes in the product would affect the builder

Builder Pattern Diagram



Demo: Builder


Singleton Pattern

Singleton Pattern Properties

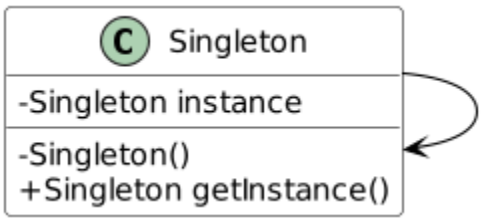
- **Type:** Creational
- **Level:** Object

Singleton Pattern Purpose

To have only one instance of this class in the system, while allowing other classes to get access to this instance.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Singleton Canonical Diagram

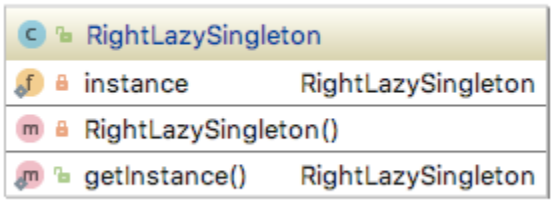


Singleton Ingredients

Singleton – Provides a private constructor, maintains a private static reference to the single instance of this class, and provides a static accessor method to return a reference to the single instance.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Singleton Demo Diagram



Singleton Pattern Advantages

- If done right, can delay use of an object
- Ensures a single object at all times

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Singleton Pattern Disadvantages

- Abuse especially among beginning programmers
- Difficulty in unit testing
- Often unnecessary, particularly in dependency injection frameworks
- No control over who accesses the object
- Once you go singleton, it’s tough to change
- Can expose threading issues, where duplicates can be created

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Demo: Singleton


Abstract Factory Pattern

Abstract Factory Pattern Properties

Type: Creational, Object
Level: Component

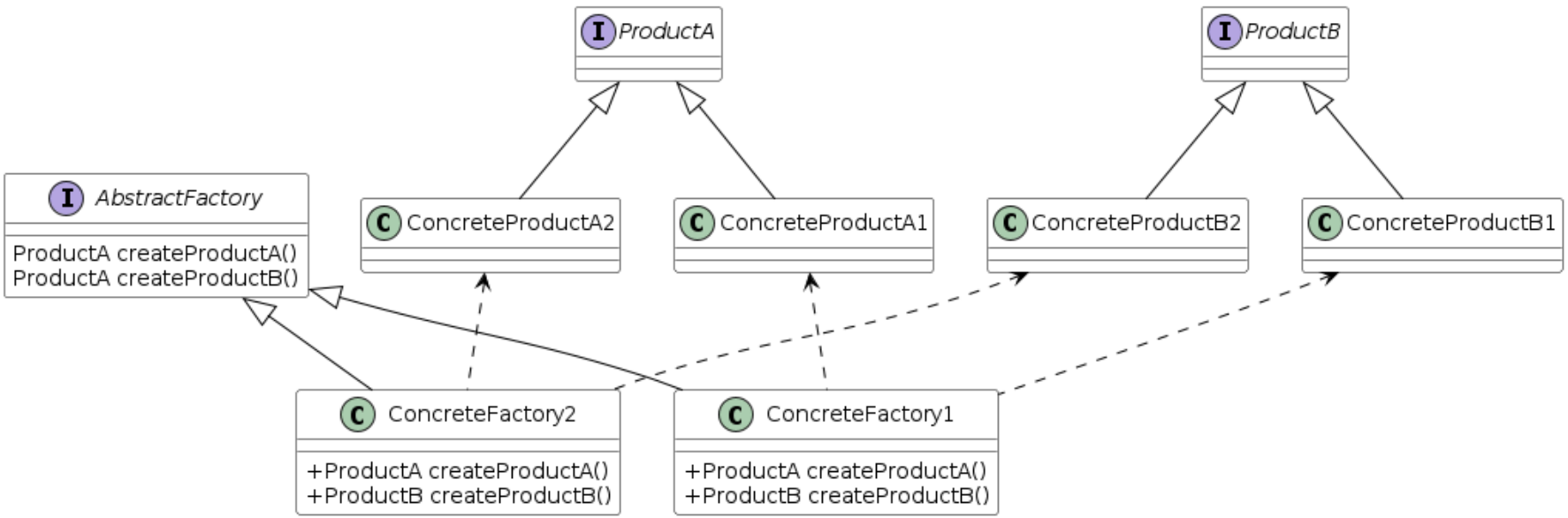
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Abstract Factory Pattern Purpose

To provide a contract for creating families of related or dependent objects without having to specify their concrete classes.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Abstract Factory Canonical Diagram



Abstract Factory Ingredients

AbstractFactory

An abstract class or interface that defines the create methods for abstract products.

AbstractProduct

An abstract class or interface describing the general behavior of the resource that will be used by the application.

ConcreteFactory

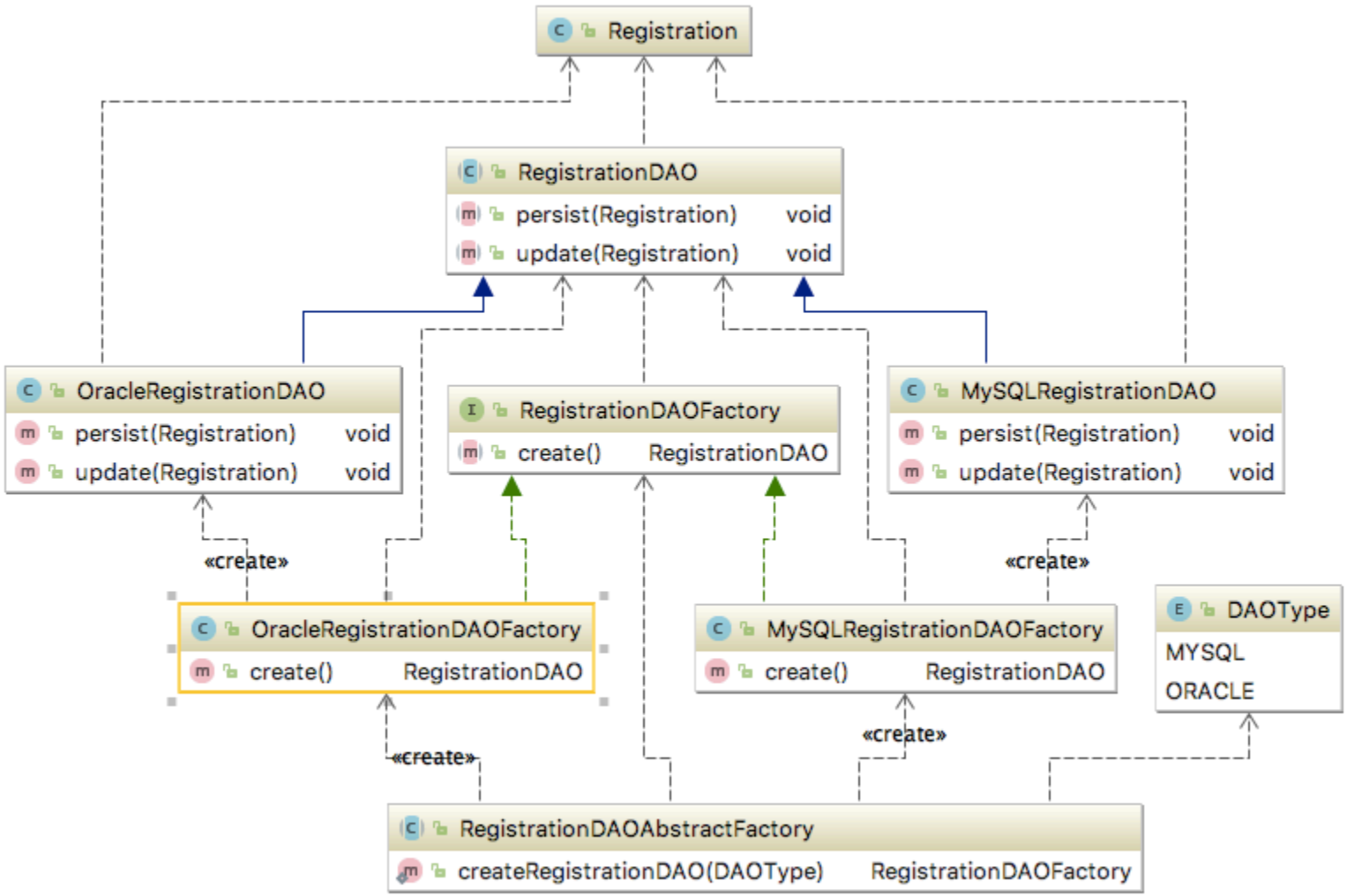
A class derived from the abstract factory . It implements create methods for one or more concrete products.

ConcreteProduct

A class derived from the abstract product, providing an implementation for a specific resource or operating environment.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Abstract Factory Demo Diagram



Abstract Factory Advantages

- Flexibility, the client is independent of how the products are created
- Application is configured with one of multiple families of products
- Objects need to be created as a set, in order to be compatible
- Provide a collection of classes and you want to reveal just their contracts and their relationships, not implementation

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Abstract Factory Disadvantages

- An ill-defined abstraction can make things difficult later

Demo: Abstract Factory

Behavioral Patterns


State Pattern

State Pattern Properties

Type: Behavioral
Level: Object

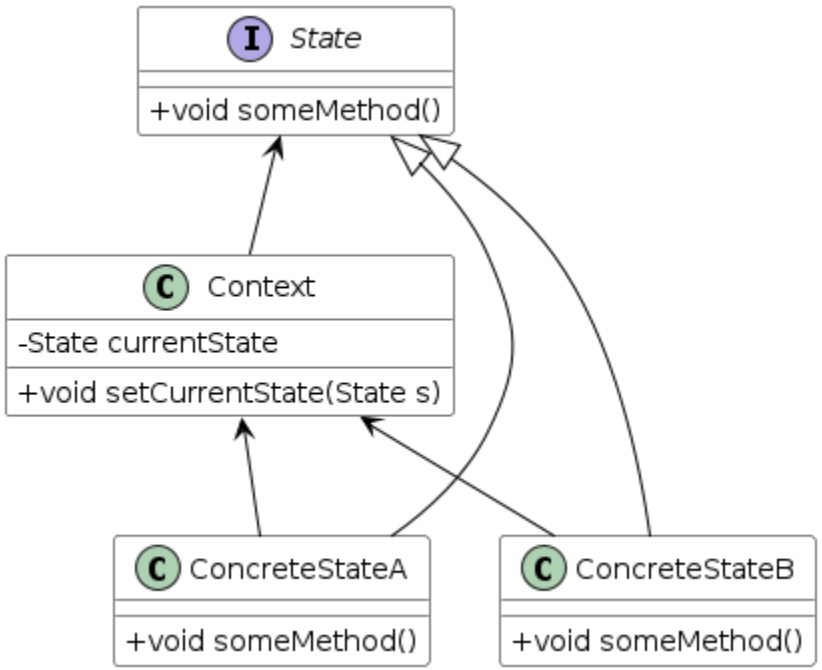
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

State Pattern Purpose

- Easily change an object's behavior at runtime.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

State Pattern Canonical Diagram



State Pattern Ingredients

Context

Keeps a reference to the current state, and is the interface for other clients to use. It delegates all state-specific method calls to the current State object.

State

Defines all the methods that depend on the state of the object.

ConcreteState

Implements the State interface, and implements specific behavior for one state.

State Pattern Advantages

- Behavior depends on its state and the state changes frequently
- Methods have large conditional statements that depend on the state of the object
- You need clarity on the change of state by focusing on the small segmentation
- Transitions are explicit and known
- States can be shared

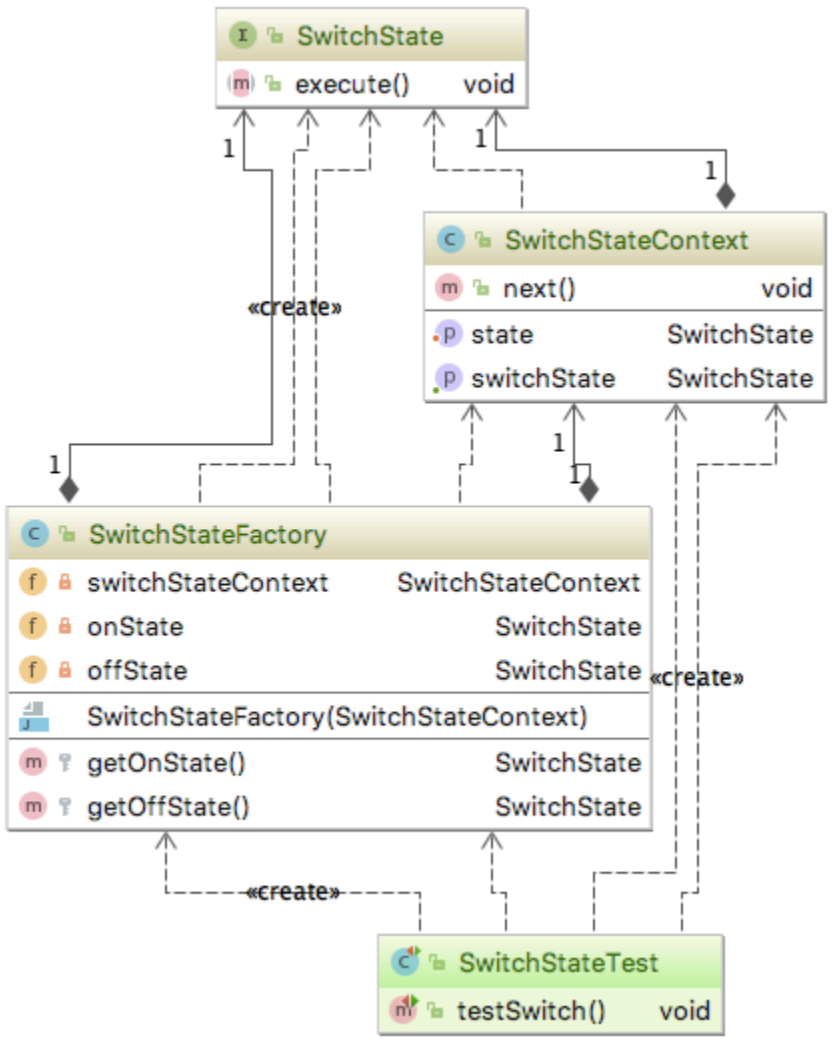
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

State Pattern Disadvantage

- Number of classes can increase
- Requires mutability, and therefore care in multi-threading

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

State Demo Diagram



Demo: State


Strategy Pattern

Strategy Pattern Properties

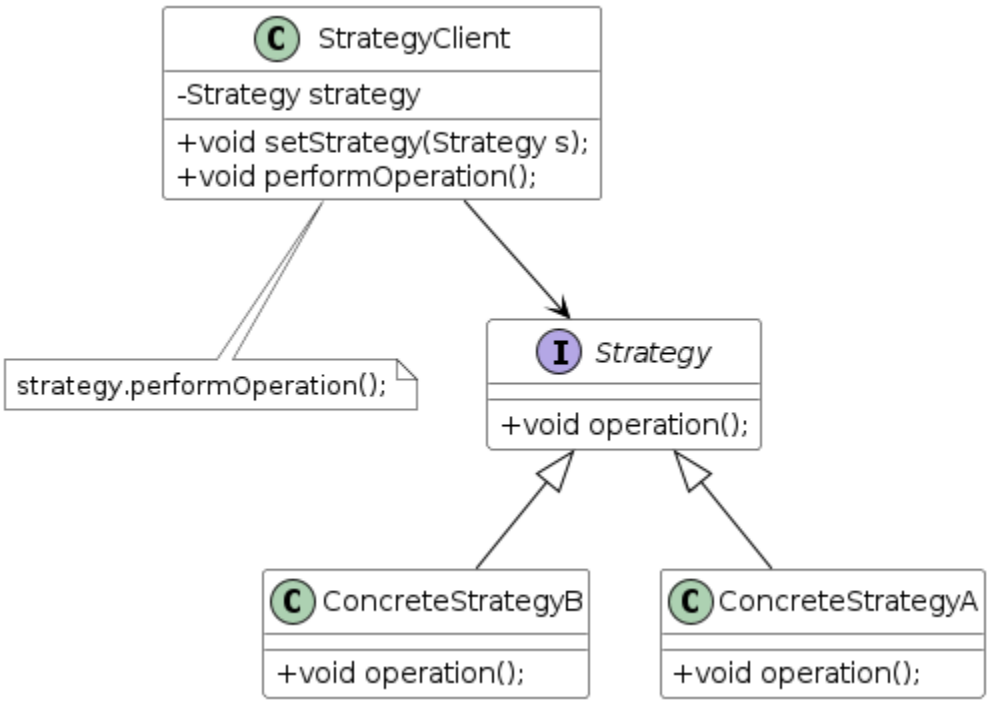
Type: Behavioral
Level: Component

Strategy Purpose

To define a group of classes that represent a set of possible behaviors. These behaviors can then be flexibly plugged into an application, changing the functionality on the fly.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Strategy Canonical Diagram



Strategy Ingredients

StrategyClient

This is the class that uses the different strategies for certain tasks. It keeps a reference to the Strategy instance that it uses and has a method to replace the current Strategy instance with another Strategy implementation.

Strategy

The interface that defines all the methods available for the StrategyClient to use.

ConcreteStrategy

A class that implements the Strategy interface using a specific set of rules for each of the methods in the interface.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Strategy Advantages

- You have a variety of ways to perform an action.
- You might not know which approach to use until runtime.
- You want to easily add to the possible ways to perform an action.
- You want to keep the code maintainable as you add behaviors.

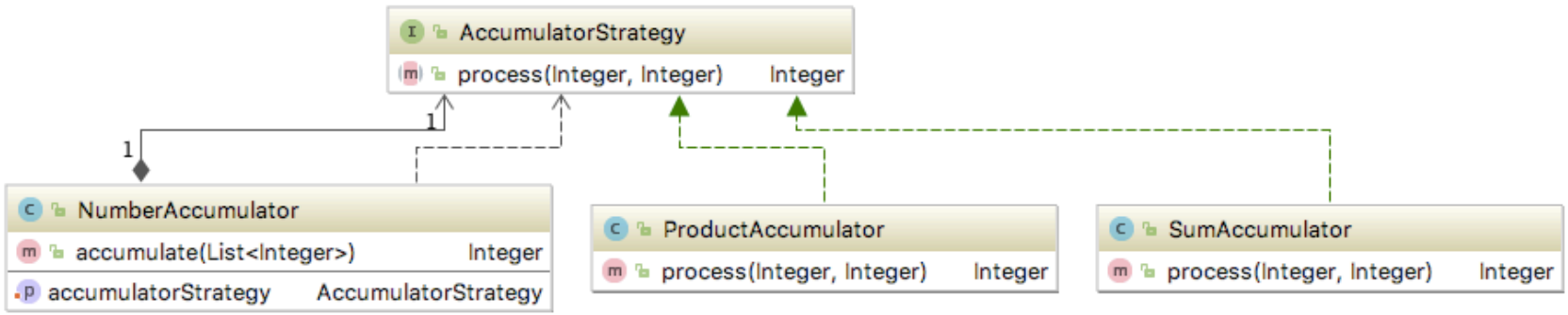
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Strategy Disadvantages

- Forethought and planning is required
- Identifying a strategy that is generic enough for this pattern

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Strategy Demo Diagram



Demo: Strategy


Chain of Responsibility Pattern

Chain of Responsibility Properties

Type: Behavioral
Level: Component

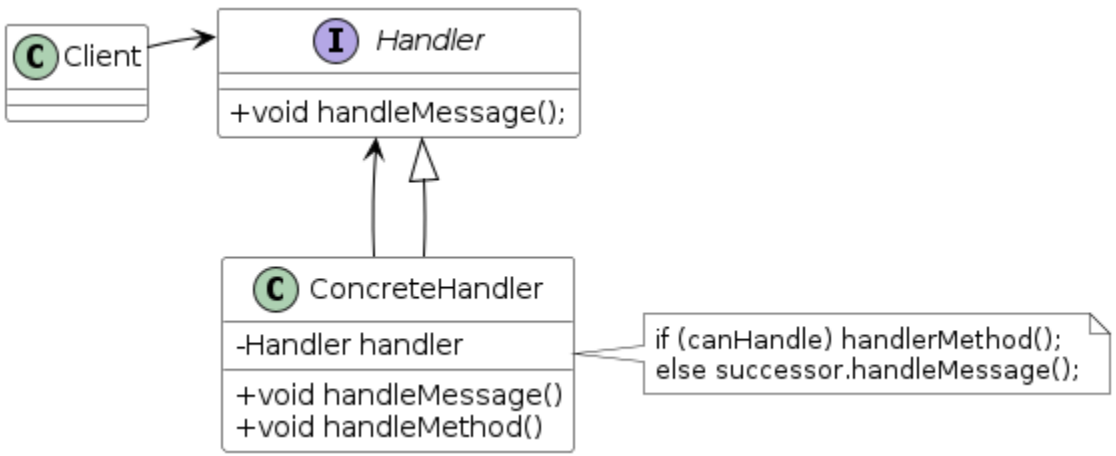
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Chain of Responsibility Purpose

To establish a chain within a system, so that a message can either be handled at the level where it is first received, or be directed to an object that can handle it.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Chain of Responsibility Canonical Diagram



Chain of Responsibility Ingredients

Handler

The interface that defines the method used to pass a message to the next handler. That message is normally just the method call, but if more data needs to be encapsulated, an object can be passed as well.

ConcreteHandler

A class that implements the Handler interface. It keeps a reference to the next Handler instance inline. This reference is either set in the constructor of the class or through a setter method. The implementation of the handleMessage method can determine how to handle the method and call a handleMethod, forward the message to the next Handler or a combination of both.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Chain of Responsibility Advantages

- There is a group of objects in a system that can all potentially respond to the same kind of message
- Offers complex message handling
- Messages must be handled by one of several objects within the system.
- Messages follow the “handle or forward” model—that is, some events can be handled at the level where they are received or produced, while others must be forwarded to some other object.

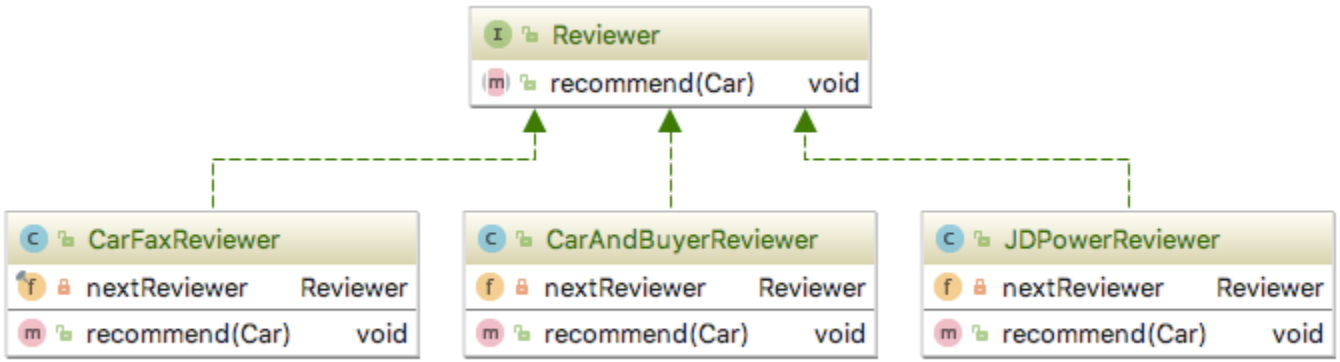
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Chain of Responsibility Disadvantages

- Difficult to test and debug
- Possible dropped message if not handled

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Chain of Responsibility Demo Diagram



Demo: Chain of Responsibility


Command Pattern

Command Pattern Properties

Type: Behavioral
Level: Object

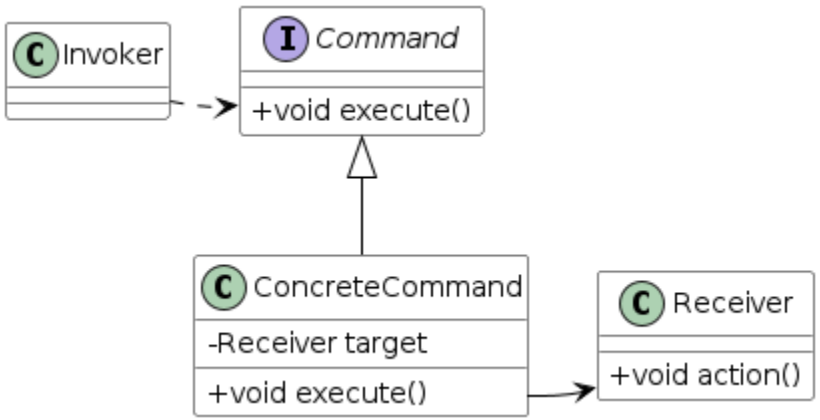
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Command Pattern Purpose

- To wrap a command in an object so that it can be stored, passed into methods, and returned like any other object.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Command Pattern Canonical Diagram



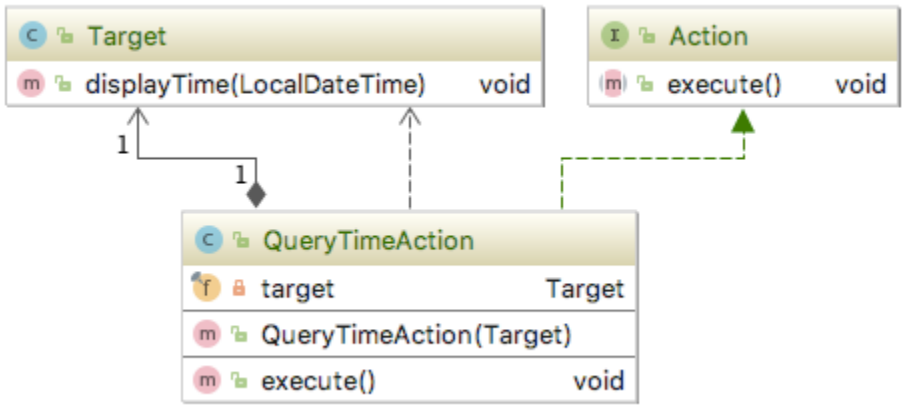
Command Pattern Advantages

- Decoupling the source or trigger of the event from the object that has the knowledge to perform the task.
- Sharing Command instances between several objects.
- Allowing the replacement of Commands and/or Receivers at runtime.
- Making Commands regular objects, thus allowing for all the normal properties.
- Easy addition of new Commands; just write another implementation of the interface and add it to the application.

Command Pattern Disadvantages

- Not beneficial with too few commands

Command Demo Diagram



Demo: Command


Iterator Pattern

Iterator Pattern Properties

Type: Behavioral, Object
Level: Component

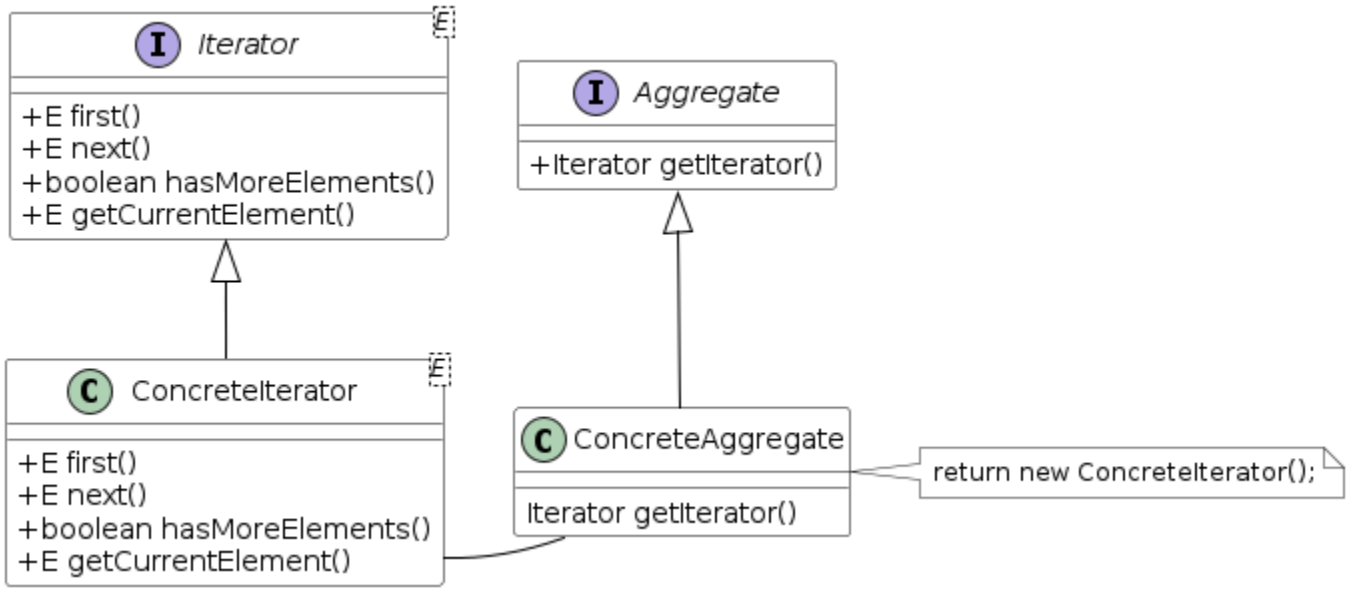
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Iterator Purpose

To provide a consistent way to sequentially access items in a collection that is independent of and separate from the underlying collection.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Iterator Canonical Diagram



Iterator Ingredients

Iterator

This interface defines the standard iteration methods. At a minimum, the interface defines methods for navigation, retrieval and validation (`first`, `next`, `hasMoreElements` and `getCurrentItem`)

ConcreteIterator

Classes that implement the `Iterator`. These classes reference the underlying collection. Normally, instances are created by the `ConcreteAggregate`. Because of the tight coupling with the `ConcreteAggregate`, the `ConcreteIterator` often is an inner class of the `ConcreteAggregate`.

Aggregate

This interface defines a factory method to produce the `Iterator`.

ConcreteAggregate

This class implements the Aggregate, building a `ConcreteIterator` on demand. The `ConcreteAggregate` performs this task in addition to its fundamental responsibility of representing a collection of objects in a system. `ConcreteAggregate` creates the `ConcreteIterator` instance.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Iterator Advantages

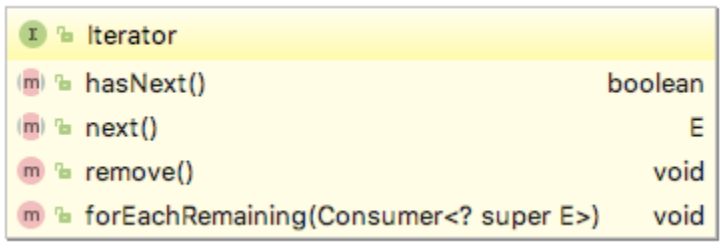
- A uniform interface for traversing a collection
- Not to tied to the implementation of the collection
- Enabling several clients to simultaneously navigate within the same underlying collection.
- You can think of an Iterator as a cursor or pointer into the collection

Iterator Disadvantages

- They give the illusion of order to unordered structures

Iterator Demo Diagram

JDK 8+ UML Diagram



Demo: Iterator

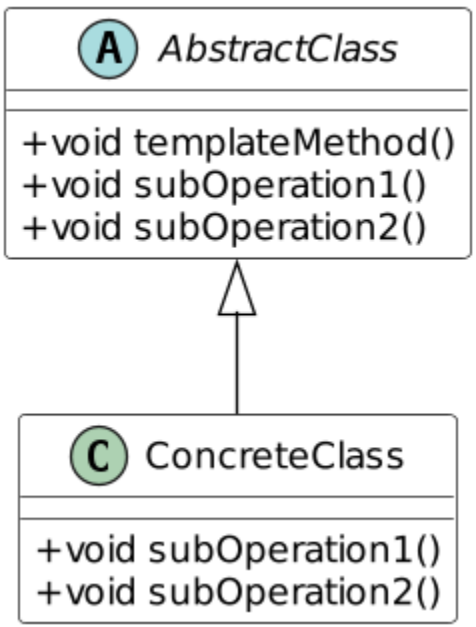

Template Method Pattern

Template Method Pattern Properties

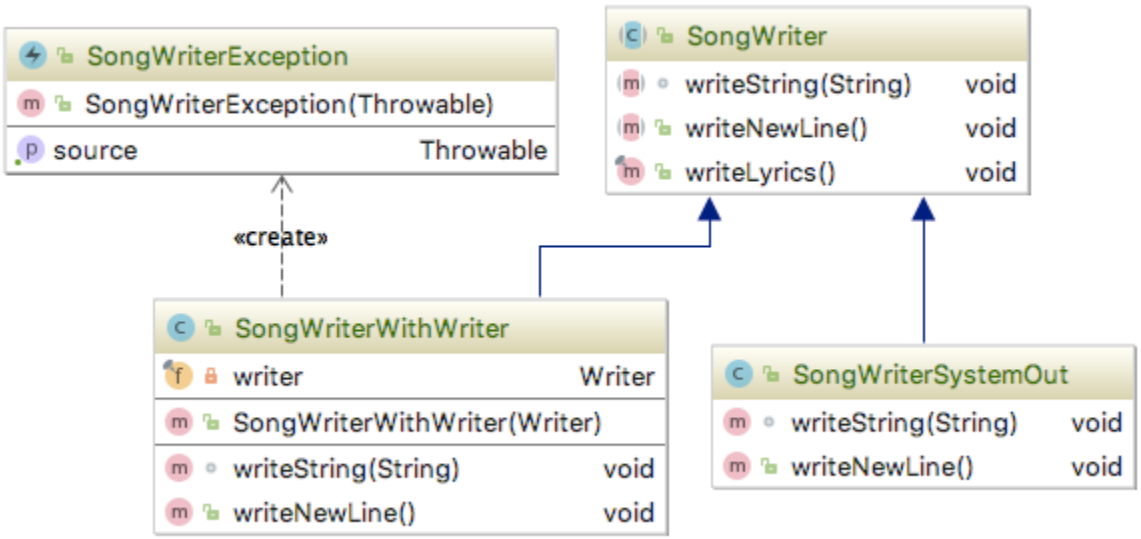
Type: Behavioral
Level: Object

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Template Method Pattern Canonical Diagram



Template Method Pattern Diagram



Template Method Pattern Purpose

- To provide a method that allows subclasses to override parts of the method without rewriting it.
- To provide a skeleton structure for a method
- Allow subclasses to redefine specific parts of the method.
- To centralize pieces of a method that are defined in all subtypes of a class
- Always have a small difference in each subclass.
- Control which operations subclasses are required to override.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Demo: Template Method


Visitor Pattern

Visitor Pattern Properties

Type: Behavioral, Object
Level: Component to System

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Visitor Purpose

To provide a maintainable, easy way to perform actions for a family of classes. Visitor centralizes the behaviors and allows them to be modified or extended without changing the classes they operate on.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Visitor Applicability

Use the Visitor pattern when the following conditions are met

- A system contains a group of related classes.
- Several non-trivial operations need to be carried out on some or all of the related classes.
- The operations must be performed differently for different classes.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Visitor Advantages

- Makes adding behavior easy
- Allows you to centralize functional code for an operation
- Allows centralized state as it *visits* every element in a structure

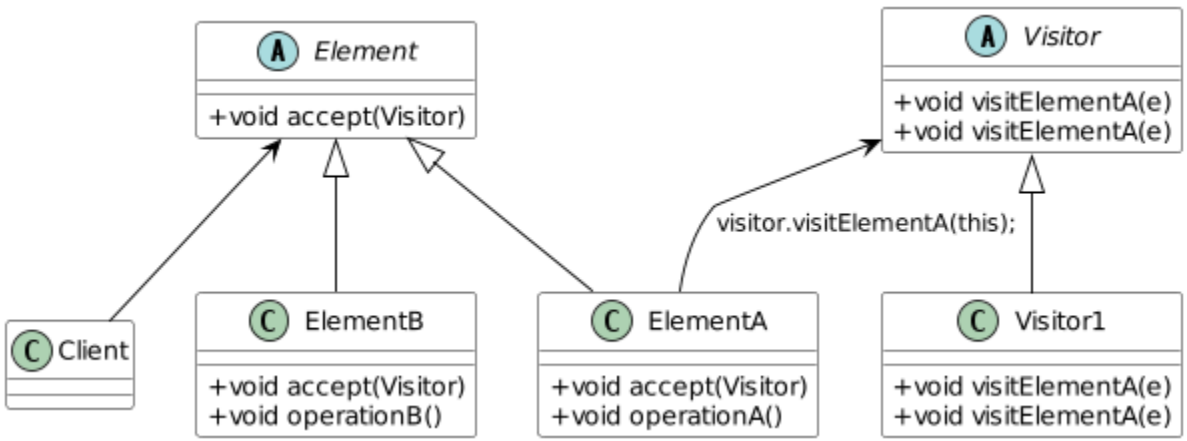
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Visitor Disadvantages

- Violates Encapsulation
- While `Element` doesn't have to know the internals for `Visitor`, `Visitor` would require to know the `Element`
- A change in `Element` would likely require a rewrite in `Visitor`

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Visitor Pattern Canonical Diagram



Visitor Ingredients

Visitor

The abstract class or interface that defines a visit method for each of the `ConcreteElement` classes.

ConcreteVisitor

Each concrete visitor class represents a specific operation to be performed for the system. It implements all the methods defined in `Visitor` for a specific operation or algorithm.

Element

An `abstract class` or `interface` that represents the objects upon which the `Visitor` operates. At a minimum, it defines an accept method that receives a Visitor as an argument.

ConcreteElement

A concrete element is a specific entity in the system. It implements the accept method defined in Element, calling the appropriate visit method defined in Visitor.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Demo: Visitor Pattern


Mediator Pattern

Mediator Pattern Properties

Type: Behavioral
Level: Component

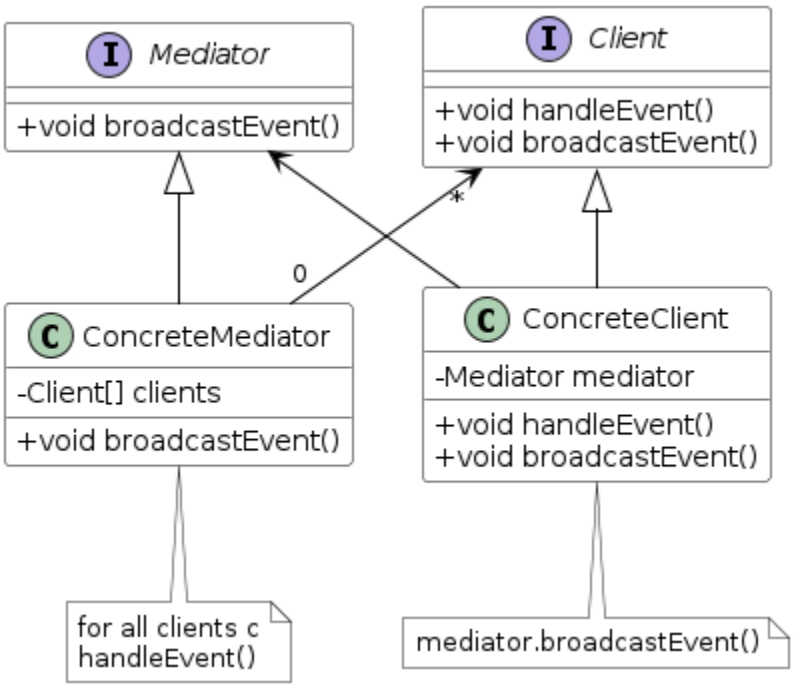
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Mediator Purpose

Simplify communication among objects in a system by introducing a single object that manages message distribution among the others

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Mediator Canonical Diagram



Mediator Ingredients

Mediator

The interface that defines the methods clients can call on a `Mediator`.

ConcreteMediator

The class that implements the `Mediator` interface. This class mediates among several client classes. It contains application-specific information about processes, and the `ConcreteMediator` might have some hardcoded references to its clients. Based on the information the Mediator receives, it can either invoke specific methods on the clients, or invoke a generic method to inform clients of a change or a combination of both.

Client

The interface that defines the general methods a `Mediator` can use to inform client instances.

ConcreteClient

A class that implements the `Client` interface and provides an implementation to each of the client methods. The `ConcreteClient` can keep a reference to a `Mediator` instance to inform colleague clients of a change (through the `Mediator`).

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Mediator Advantages

- There are complex rules for communication among objects in a system (often as a result of the business model).
- You want to keep the objects simple and manageable.
- You want the classes for these objects to be redeployable, not dependent on the business model of the system.
- The individual components become simpler and easier to deal with, since they no longer need to directly pass messages to each other.
- Components are more generic, no longer need to contain logic to deal with their communication with other components
- Communications strategy becomes easier, since it is now the exclusive responsibility of the mediator

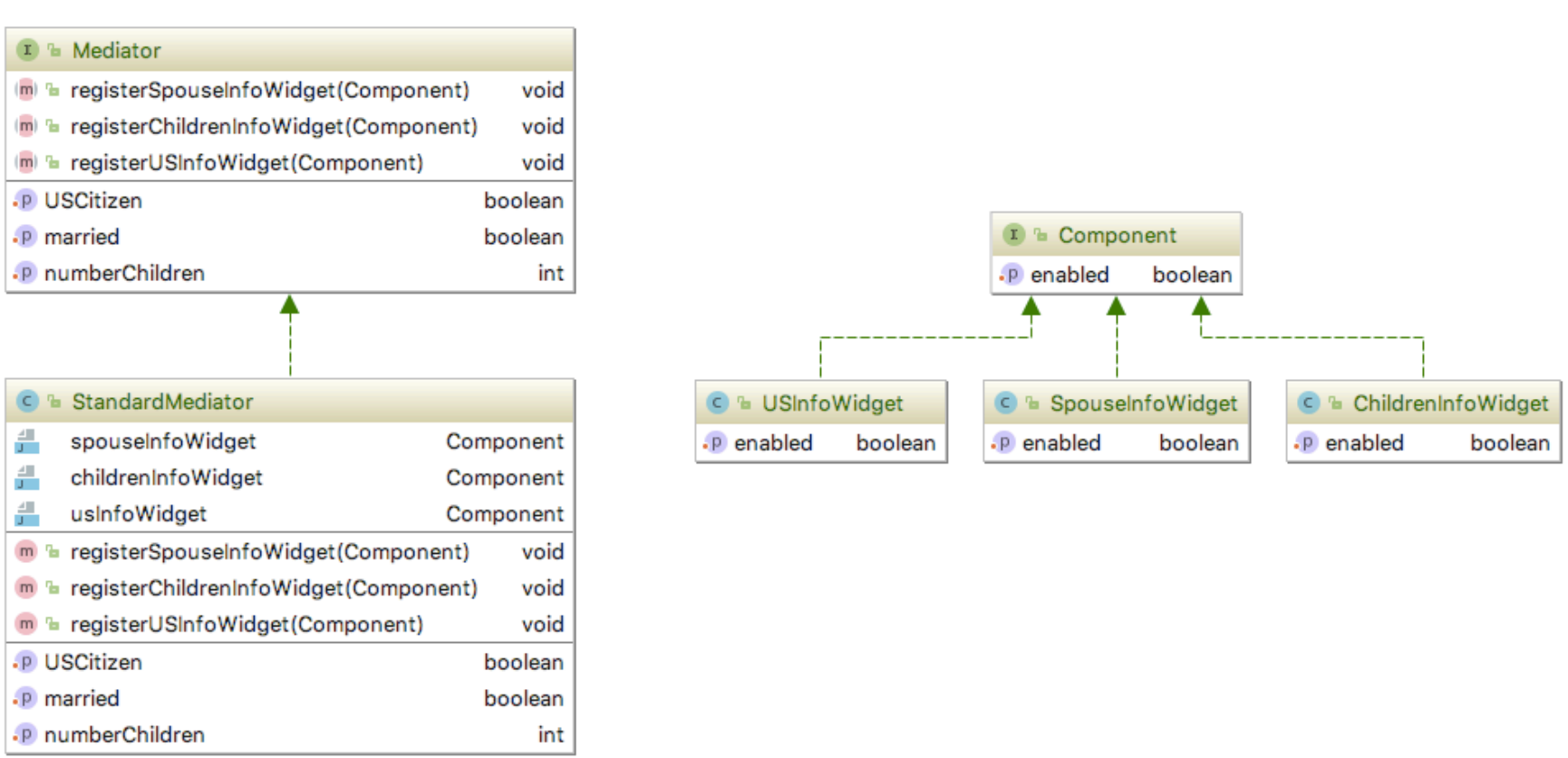
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Mediator Disadvantages

- The Mediator is often application specific and difficult to redeploy
- Testing and debugging complex Mediator implementations can be challenging
- The Mediator's code can become hard to manage as the number and complexity of participants increases

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Mediator Demo Diagram



Demo: Mediator


Memento Pattern

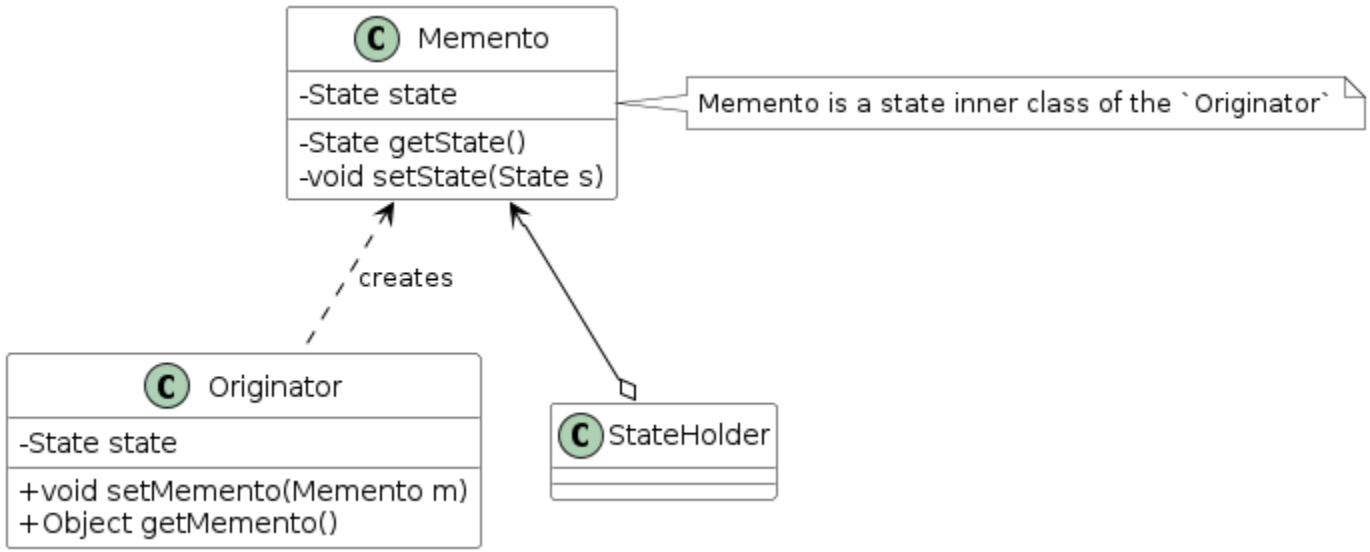
Memento Pattern Properties

Type: Behavioral
Level: Object

Memento Purpose

- To preserve a "snapshot" of an object’s state
- Object can return to its original state without having to reveal its content to the rest of the world

Memento Canonical Diagram



Memento Advantages

- A snapshot of the state of an object should be taken.
- That snapshot is used to recreate the original state.
- Doesn’t not expose internal state

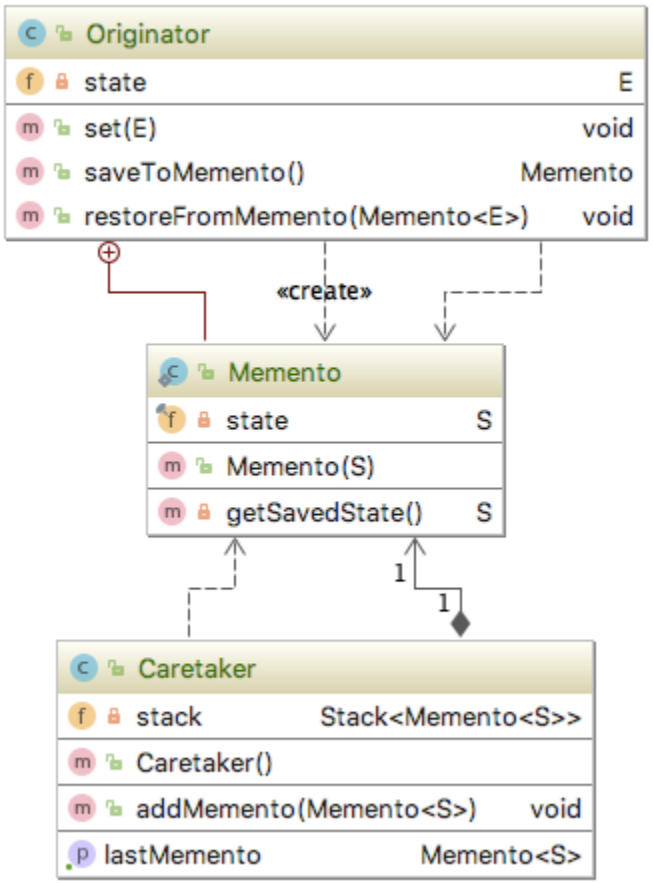
Source: https://en.wikipedia.org/wiki/Memento_pattern

Memento Disadvantages

- Expensive Storage Overtime
- Requires thought with large graphs

Source: https://en.wikipedia.org/wiki/Memento_pattern

Memento Demo Diagram



Demo: Memento


Observer Pattern

Observer Pattern Properties

Type: Behavioral
Level: Component

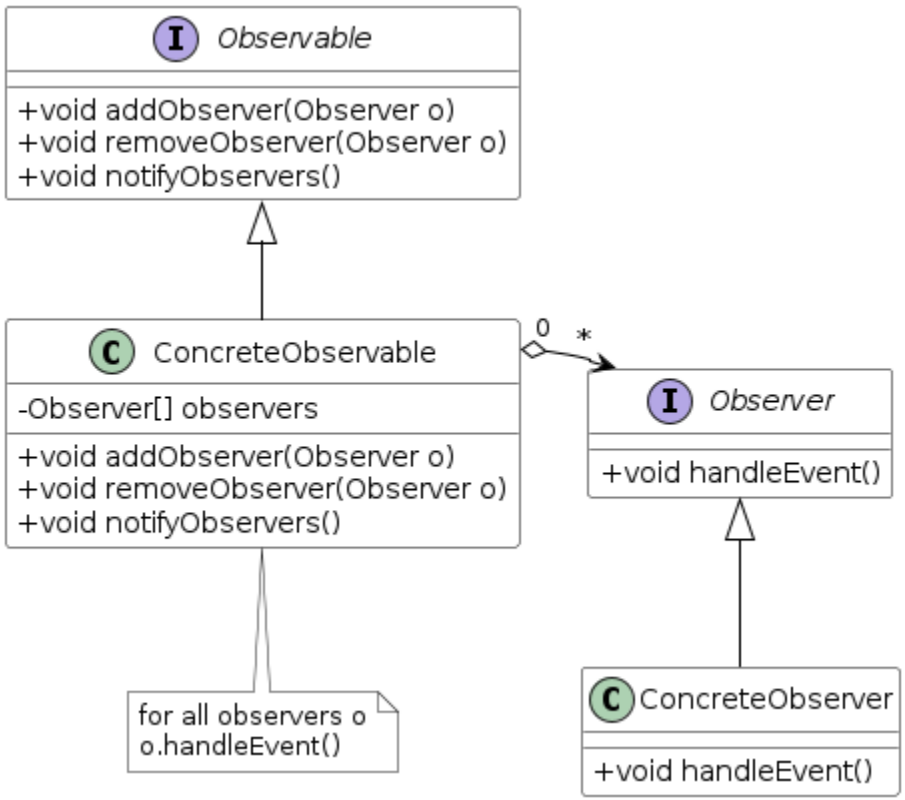
Source: https://en.wikipedia.org/wiki/Memento_pattern

Observer Pattern Purpose

To provide a way for a component to flexibly broadcast messages to interested receivers.

Source: https://en.wikipedia.org/wiki/Memento_pattern

Observer Canonical Pattern



Observer Ingredients

Observable

The interface that defines how the observers/clients can interact with an `Observable`. These methods include adding and removing observers, and one or more notification methods to send information through the `Observable` to its clients.

ConcreteObservable

A class that provides implementations for each of the methods in the `Observable` interface. It needs to maintain a collection of Observers. The notification methods copy (or clone) the `Observer` list and iterate through the list, and call the specific listener methods on each `Observer`.

Observer

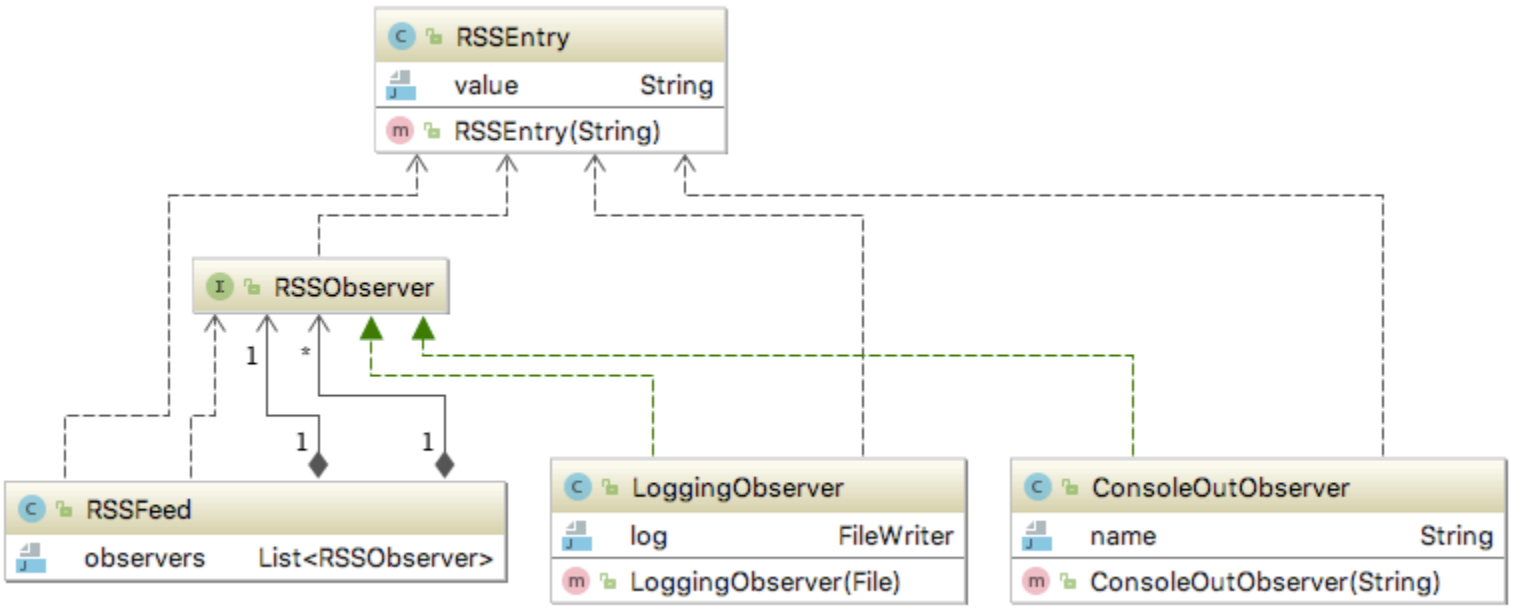
The interface the `Observer` uses to communicate with the clients.

ConcreteObserver

Implements the `Observable` interface and determines in each implemented method how to respond to the message received from the `Observable`.

Source: https://en.wikipedia.org/wiki/Memento_pattern

Observer Demo Diagram



Demo: Observer

Structural Patterns


Adapter Pattern

Adapter Pattern Properties

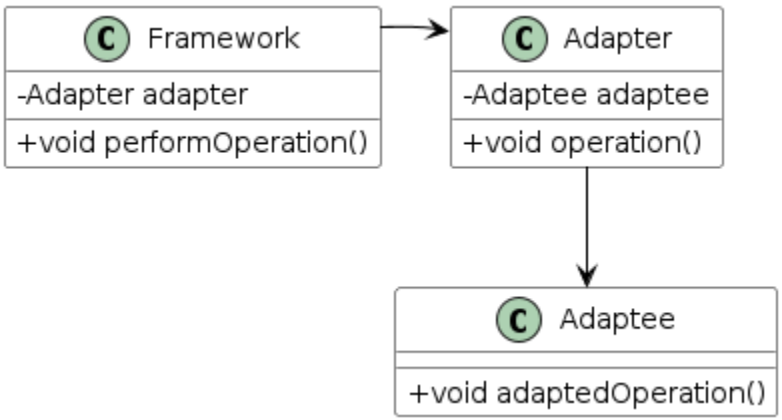
Type: Structural, Object
Level: Component

Adapter Pattern Purpose

- To act as an intermediary between two classes
- Converting the interface of one class so that it can be used with another

Source: https://en.wikipedia.org/wiki/Memento_pattern

Adapter Pattern Canonical Diagram



Adapter Pattern Advantages

- Code Reuse
- Apply a different interface
- Translate code from another language

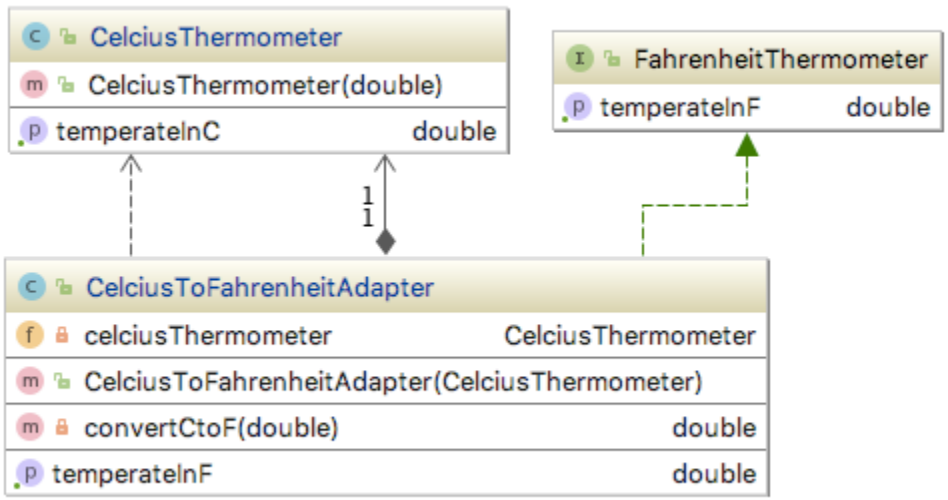
Source: https://en.wikipedia.org/wiki/Memento_pattern

Adapter Pattern Disadvantages

- The parameters may not be the same
- May require more work if the methods are substantial

Source: https://en.wikipedia.org/wiki/Memento_pattern

Adapter Pattern Demo Diagram



Demo: Adapter Pattern


Bridge Pattern

Bridge Pattern Properties

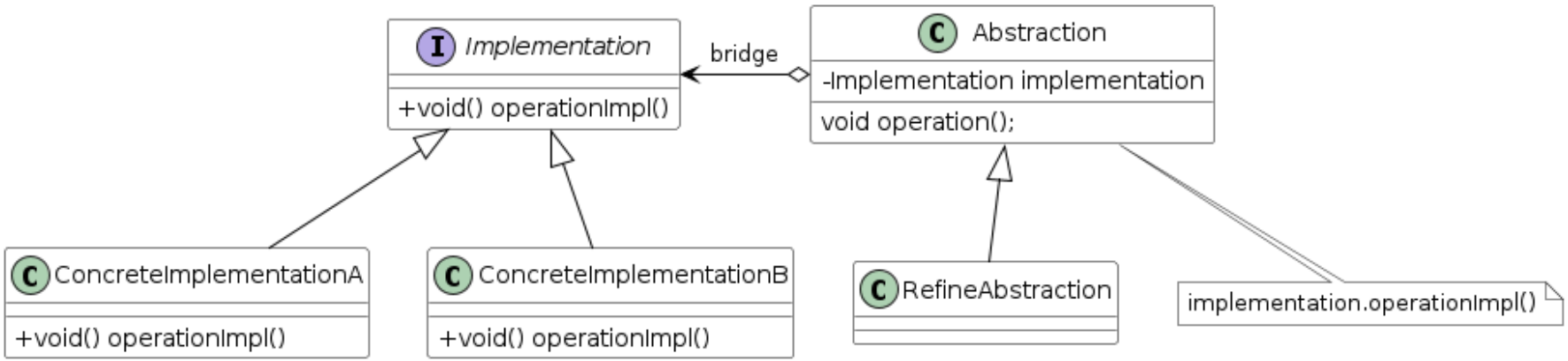
Type: Structural, Object
Level: Component

Bridge Pattern Purpose

- To divide a complex component into two separate but related inheritance hierarchies:
  - The functional Abstraction
  - The internal implementation
- This makes it easier to change either aspect of the component

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Bridge Pattern Canonical Diagram



Bridge Pattern Advantages

- Decouples an implementation so that it is not bound permanently to an interface.
- Abstraction and implementation can be extended independently.
- Changes to the concrete abstraction classes don’t affect the client.

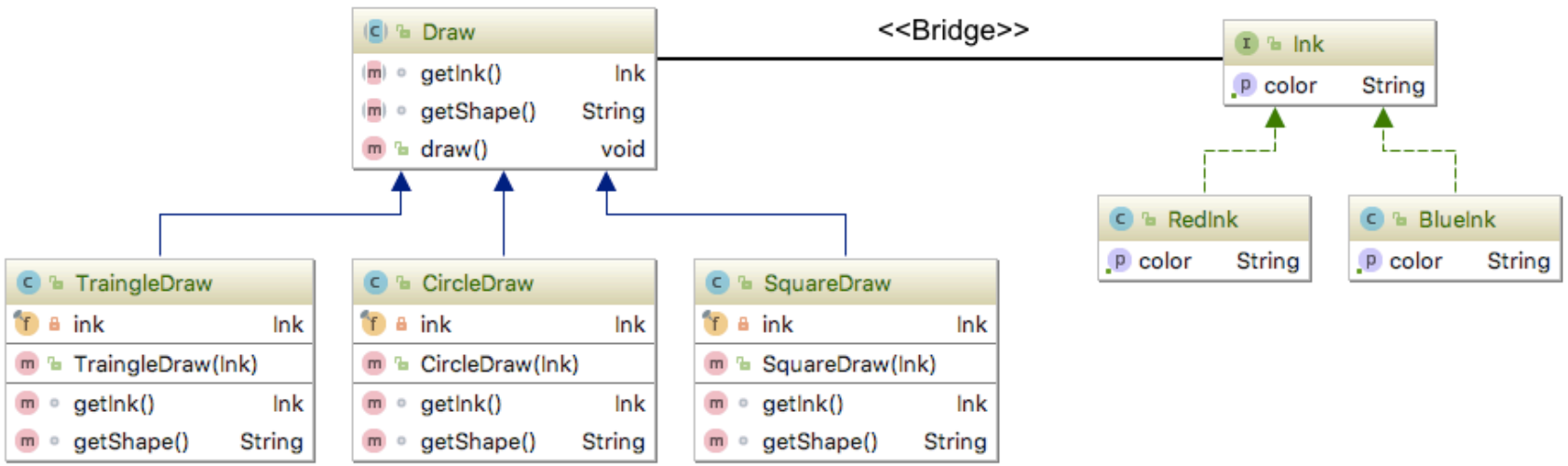
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Bridge Pattern Disadvantages

- Useful in graphics and windowing systems that need to run over multiple platforms.
- Useful any time you need to vary an interface and an implementation in different ways.
- Increases complexity and components

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Bridge Demo Diagram



Demo: Bridge


Composite Pattern

Composite Properties

Type: Structural, Object **Level:** Component

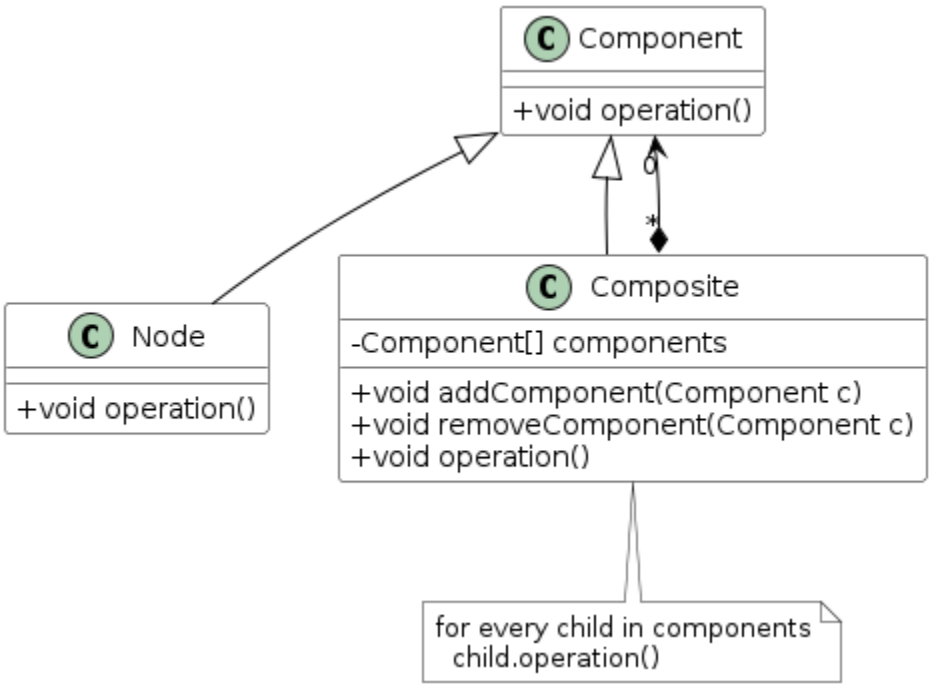
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Composite Purpose

- To develop a flexible way to create hierarchical tree structures of arbitrary complexity
- While enabling every element in the structure to operate with a uniform interface

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Composite Pattern Canonical Diagram



Composite Ingredients

Component

The `Component` interface defines methods available for all parts of the tree structure. `Component` may be implemented as abstract class when you need to provide standard behavior to all of the sub-types. Normally, the component is not instantiable; its subclasses or implementing classes, also called nodes, are instantiable and are used to create a collection or tree structure.

Composite

This class is defined by the components it contains; it is composed by its components. The `Composite` supports a dynamic group of `Components` so it has methods to add and remove `Component` instances from its collection. The methods defined in the `Component` are implemented to execute the behavior specific for this type of `Composite` and to call the same method on each of its nodes. These `Composite` classes are also called branch or container classes.

Composite Ingredients (Continued)

Leaf

The class that implements the `Component` interface and that provides an implementation for each of the `Component`'s methods. The distinction between a Leaf class and a `Composite` class is that the Leaf contains no references to other `Components`. The Leaf classes represent the lowest levels of the containment structure.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

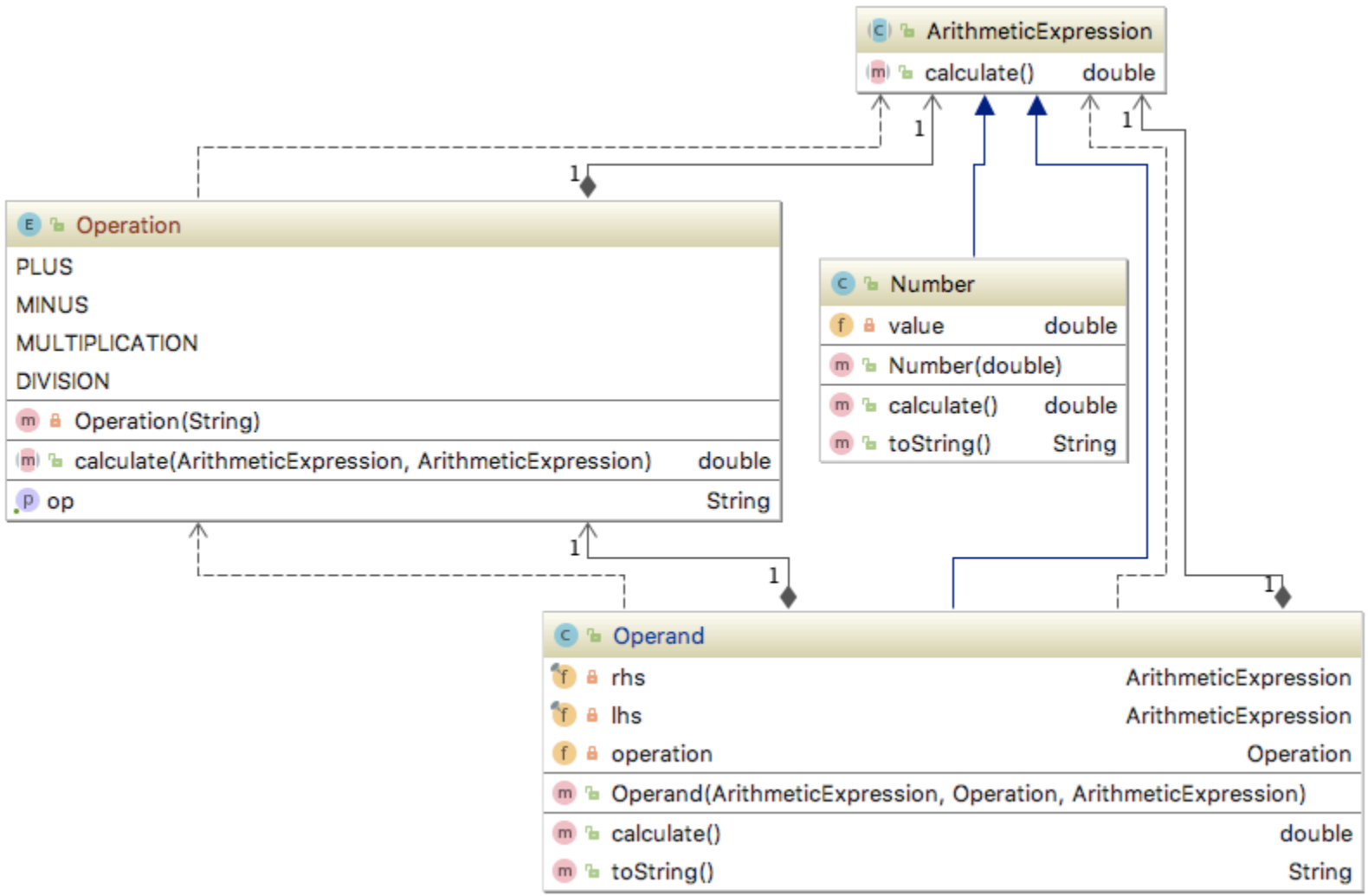
Composite Advantages

- Users perceive a unified structure
- Users can also add or remove components
- Great for
  - UI Development
  - Organizational Charts
  - Schedules
  - Outlines

Composite Disadvantages

- Because it is so dynamic, the Composite pattern is often difficult to test and debug
- It normally requires a more sophisticated test/validation strategy that is designed around the concept of the whole-part object hierarchy
- Requires full advance knowledge of the structure being modeled

Composite Pattern Diagram



Demo: Composite


Decorator Pattern

Decorator Pattern Properties

Type: Structural
Level: Component

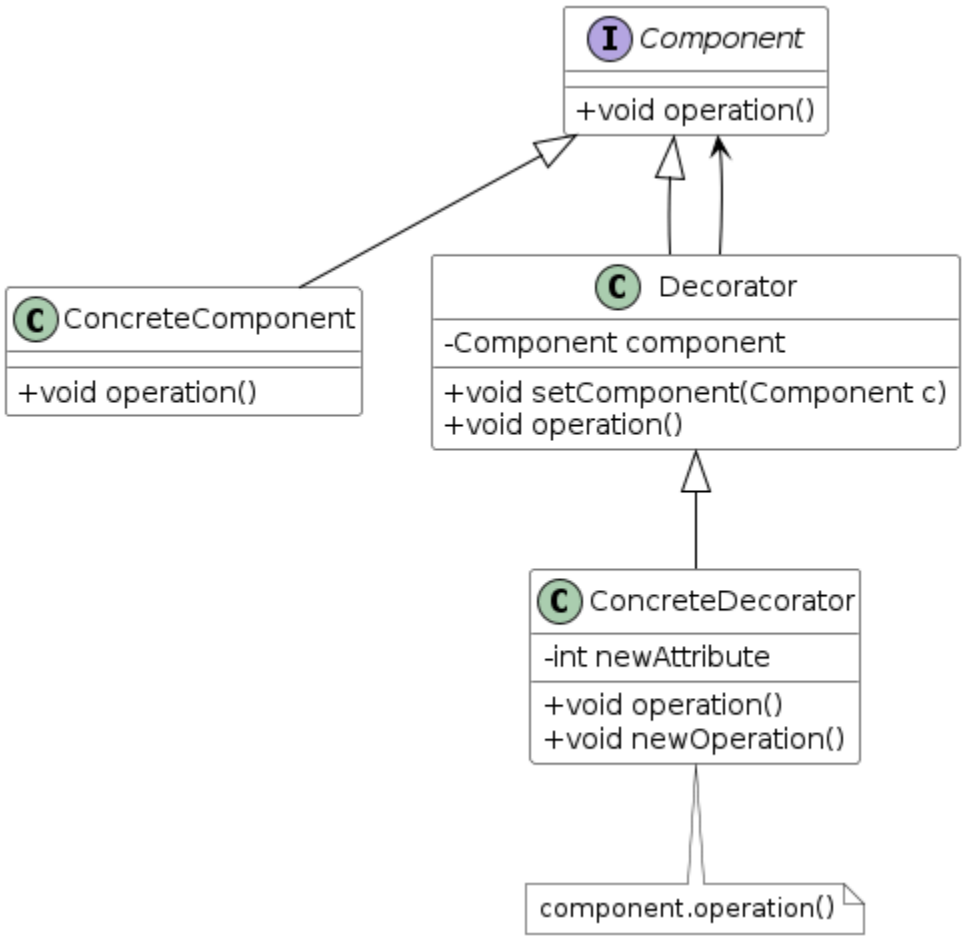
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Decorator Pattern Purpose

- Also known as a Wrapper
- To provide a way to flexibly add or remove component functionality without changing its external appearance or function

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Decorator Canonical Diagram



Decorator Pattern Advantages

- "Delegation over Inheritance" - Effective Java Josh Bloch
- Produce classes with plugin capabilities
- You want to make dynamic changes that are transparent to users, without the restrictions of subclassing
- Offers the opportunity to easily adjust and augment the behavior of an object during runtime
- Can reduce memory by reusing layers

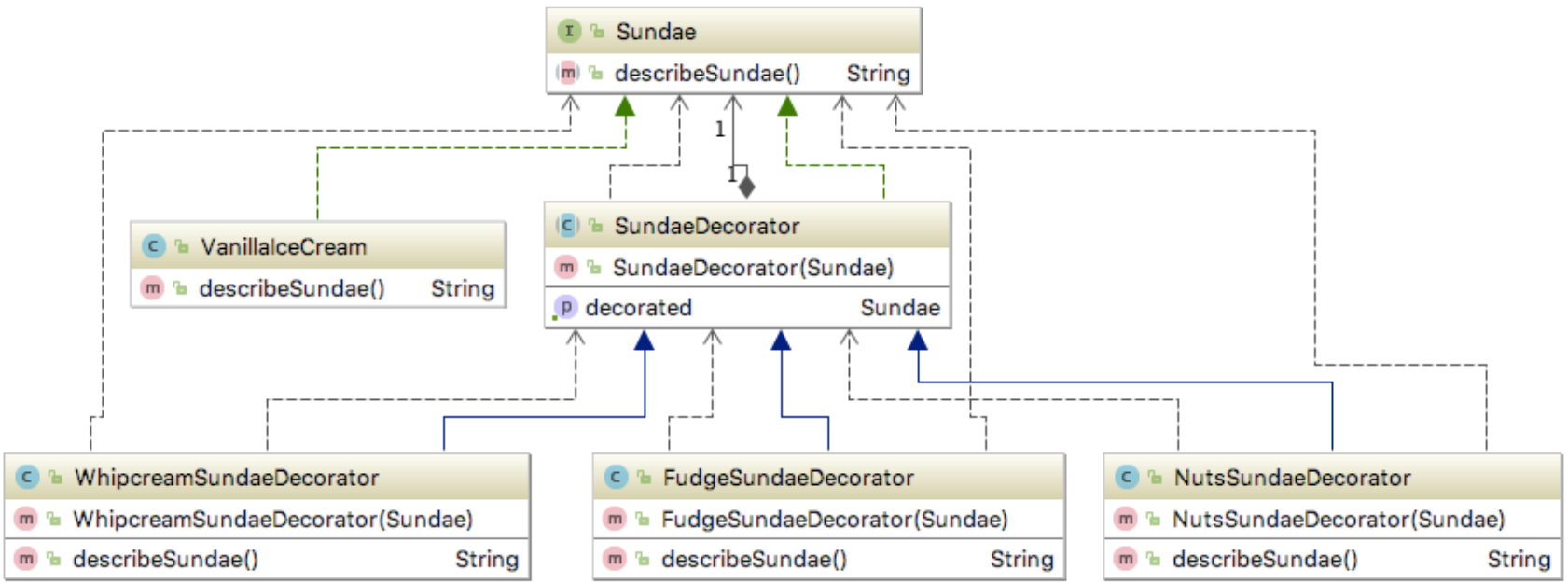
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Decorator Pattern Disadvantages

- Can produce large number of layers
- Debugging and testing can be difficult
- Can be slow if done incorrectly

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Decorator Demo Diagram



Demo: Decorator

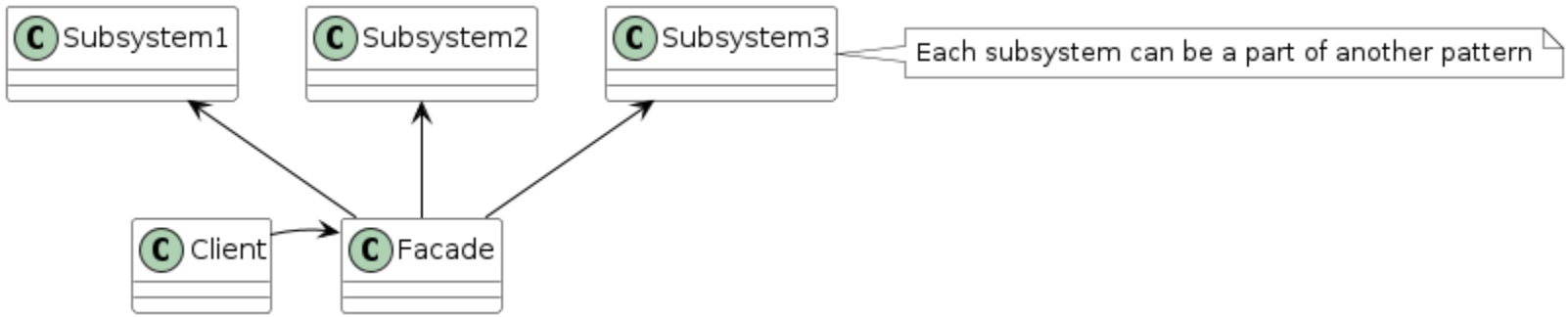

Facade Pattern

Facade Pattern Properties

Type: Structural
Level: Component

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Facade Pattern Canonical Diagram



Facade Ingredients

Facade

The class for clients to use. It knows about the subsystems it uses and their respective responsibilities. Normally all client requests will be delegated to the appropriate subsystems.

Subsystem

This is a set of classes. They can be used by clients directly or will do work assigned to them by the `Facade`. It does not have knowledge of the `Facade`; for the subsystem the `Facade` will be just another client.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Facade Purpose

- To provide a simplified interface to a group of subsystems or a complex subsystem
- Reduce coupling between clients and subsystems.
- Layer subsystems by providing Facades for sets of subsystems.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Facade Advantages

- Protects the client with an overabundance of options, parameters, and setup methods
- One request can be translated to multiple subsystems
- Promotes low coupling between subsystems

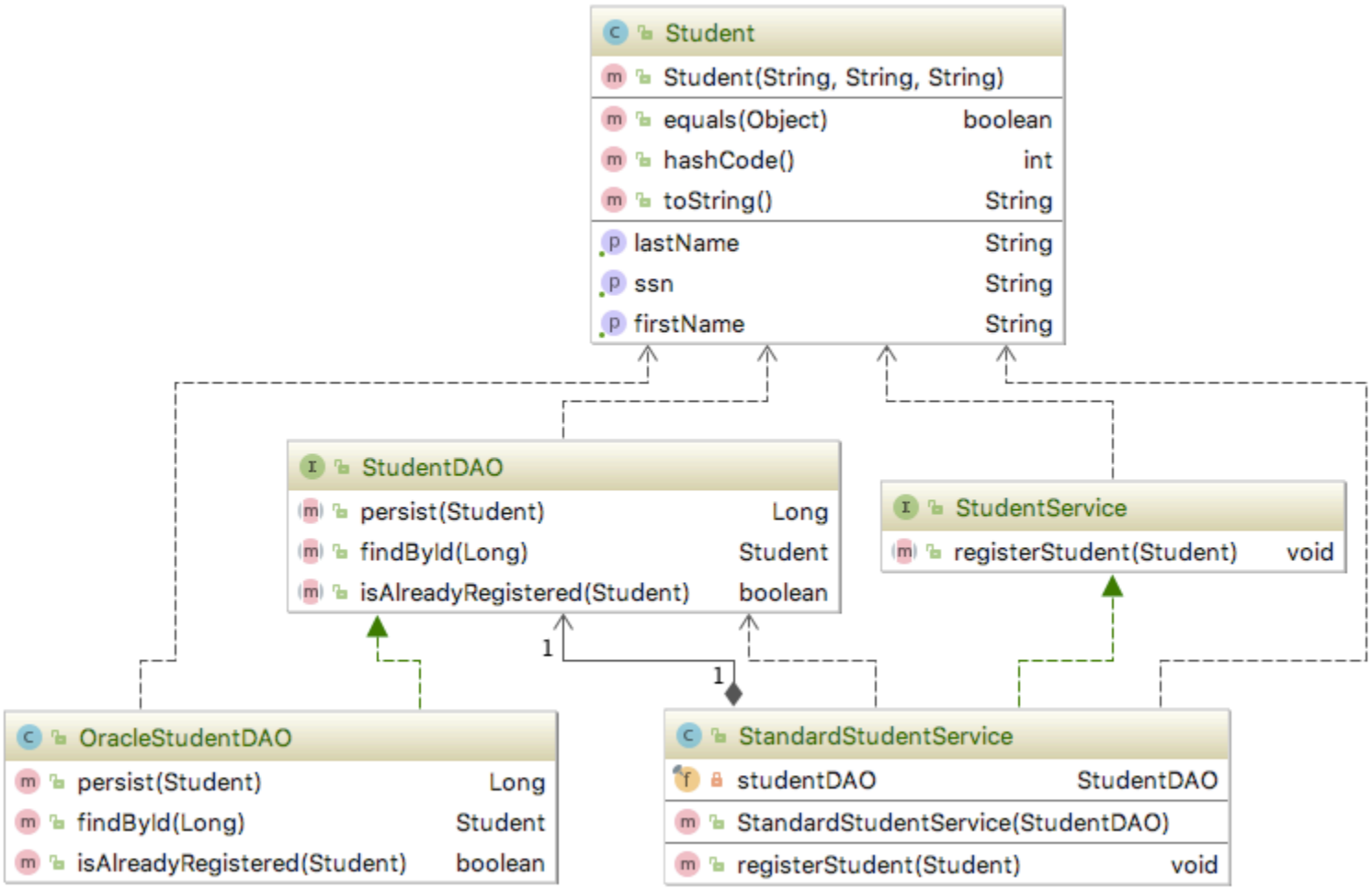
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Facade Disadvantages

- Can be difficult to debug if subsystems gets too high

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Facade Demo Diagram



Demo: Facade


Proxy Pattern

Proxy Pattern Properties

Type: Structural
Level: Component

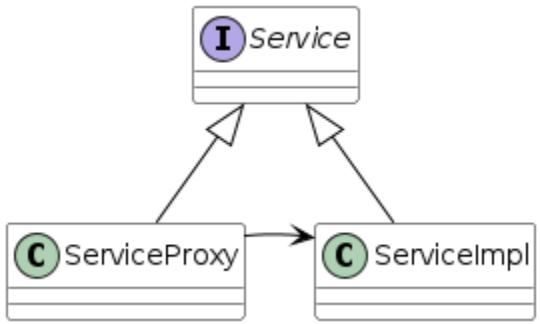
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Proxy Purpose

To provide a representative of another object, for reasons such as access, speed, or security.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Proxy Canonical Diagram



Proxy Ingredients

Service

The interface that both the proxy and the real object will implement.

ServiceProxy

ServiceProxy implements `Service` and forwards method calls to the real object (`ServiceImpl`) when appropriate.

ServiceImpl

The real, full implementation of the interface. This object will be represented by the Proxy object.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Proxy Advantages

- Kind of an adapter pattern, but for complex, remote, or both objects
- Delays creation of those expensive objects
- Can be used to constrain access based on access control

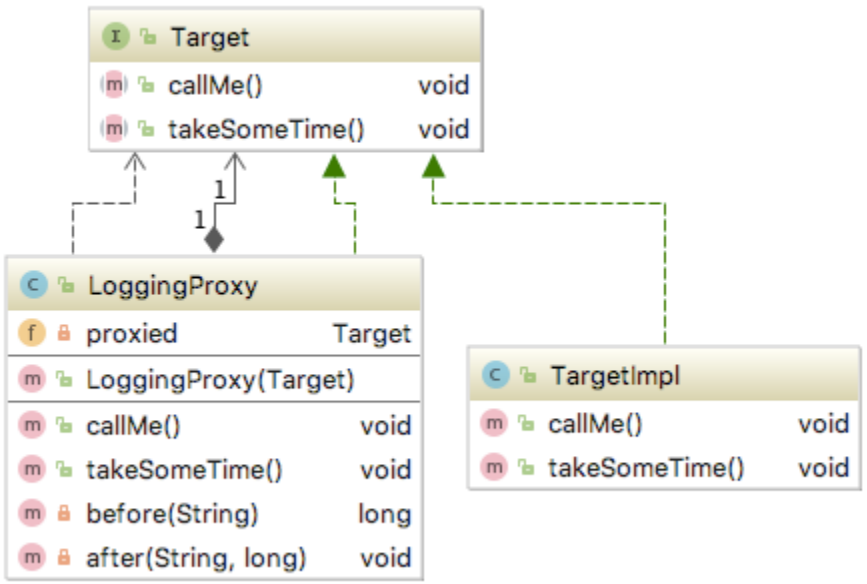
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Proxy Disadvantages

- Complicated setup
- Unnecessary for simple local reference

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Proxy Demo Diagram



Demo: Proxy


Flyweight Pattern

Flyweight Pattern Properties

Type: Structural
Level: Component

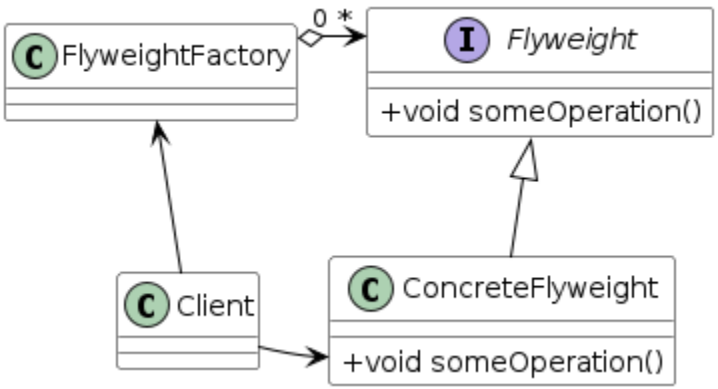
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Flyweight Purpose

- Provides for sharing an object between clients
- Creating a responsibility for the shared object that normal objects do not need to consider
- An ordinary object doesn't have to worry much about shared responsibility
- Most often, only one client will hold a reference to an object at any one time
- When the object's state changes, it's because the client changed it, and the object does not have any responsibility to inform any other clients
- Sometimes, though, you will want to arrange for multiple clients to share access to an object

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Flyweight Canonical Diagram



Flyweight Ingredients

Flyweight

The interface defines the methods clients can use to pass external state into the flyweight objects.

ConcreteFlyweight

This implements the `Flyweight` interface, and implements the ability to store internal data. The internal data has to be representative for all the instances where you need the Flyweight.

FlyweightFactory

This factory is responsible for creating and managing the Flyweights. Providing access to Flyweight creation through the factory ensures proper sharing. The factory can create all the flyweights at the start of the application, or wait until they are needed.

Client

The client is responsible for creating and providing the context for the flyweights. The only way to get a reference to a flyweight is through `FlyweightFactory`.

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Flyweight Advantages

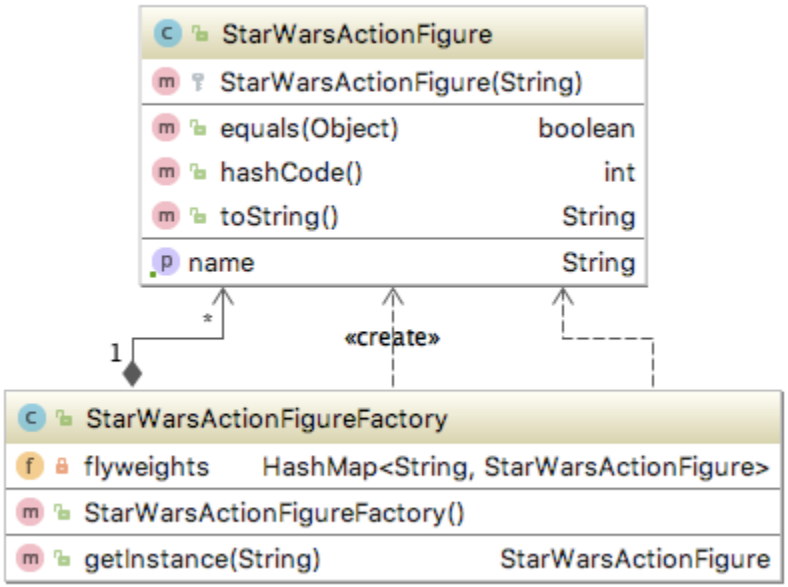
- Sharing an object among multiple clients occurs when you must manage thousands
- Provides for sharing an object between clients
- Save in memory
- Runtime can be efficient

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Flyweight Disadvantages

- None

Flyweight Demo Diagram



Demo: Flyweight


Interpreter Pattern

Interpreter Pattern Properties

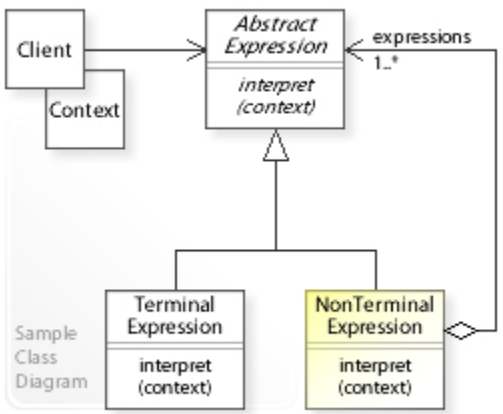
Type: Structural
Level: Component

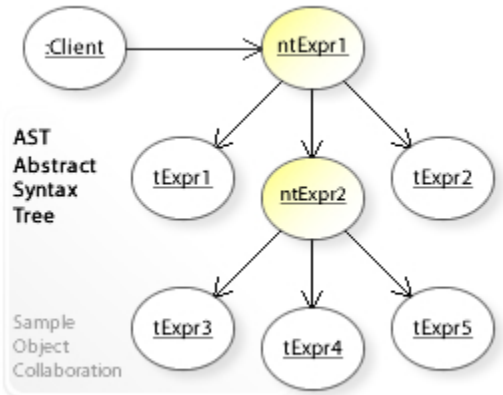
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Interpreter Purpose

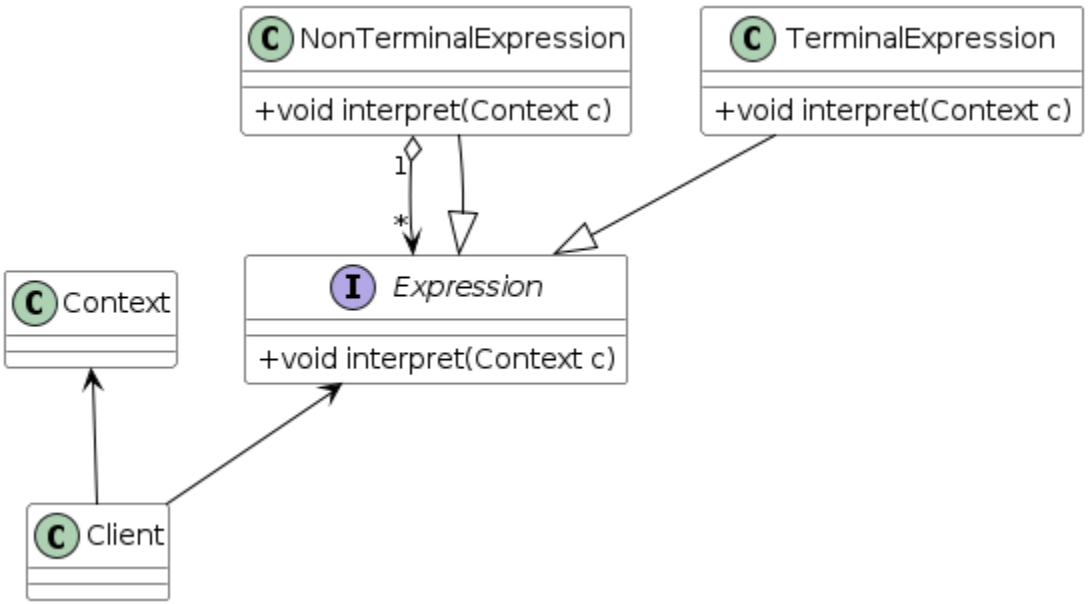
- Take a complex problem and break it down
- Creates a language within a language
- The language is meant to describe relationships
- You solve the smaller parts to solve a greater solution

Interpreter Canonical Pattern





Interpreter Canonical Pattern



Interpreter Advantages

- The interpreter can be very easily changed to reflect changes in the grammar.
- To add a rule, create another class that implements the Expression interface.
- Interpreters can be reused

Interpreter Disadvantages

- Can be difficult if the grammar is large
- Can result in a large number of classes, every rule requires one or more classes
- Might be difficult to create the AST (Abstract Syntax Tree)

Interpreter Ingredients

- **Expression** – The `interface` through which the client interacts with the expressions.
- **TerminalExpression** – Implementation of the `Expression` interface intended for terminal nodes in the grammar and the syntax tree.
- **NonterminalExpression** – The other implementation of the `Expression` interface, intended for the nonterminal nodes in the grammar and syntax tree. It keeps a reference to the next `Expression` (s) and invokes the interpret method on each of its children.
- **Context** – Container for the information that is needed in several places in the interpreter. It can serve as a communication channel among several `Expression` instances.
- **Client** – Either builds or receives an instance of an abstract syntax tree. This syntax tree is composed of instances of `TerminalExpression`s and `NonterminalExpression`s to model a specific sentence. The `client` invokes the interpret method with the appropriate context where necessary.

Demo: Interpreter Pattern

Discussion of Active Projects

Discussion of Active Projects

- What frameworks do you currently use?
- What libraries do you currently use?
- What patterns are enforced with those projects?

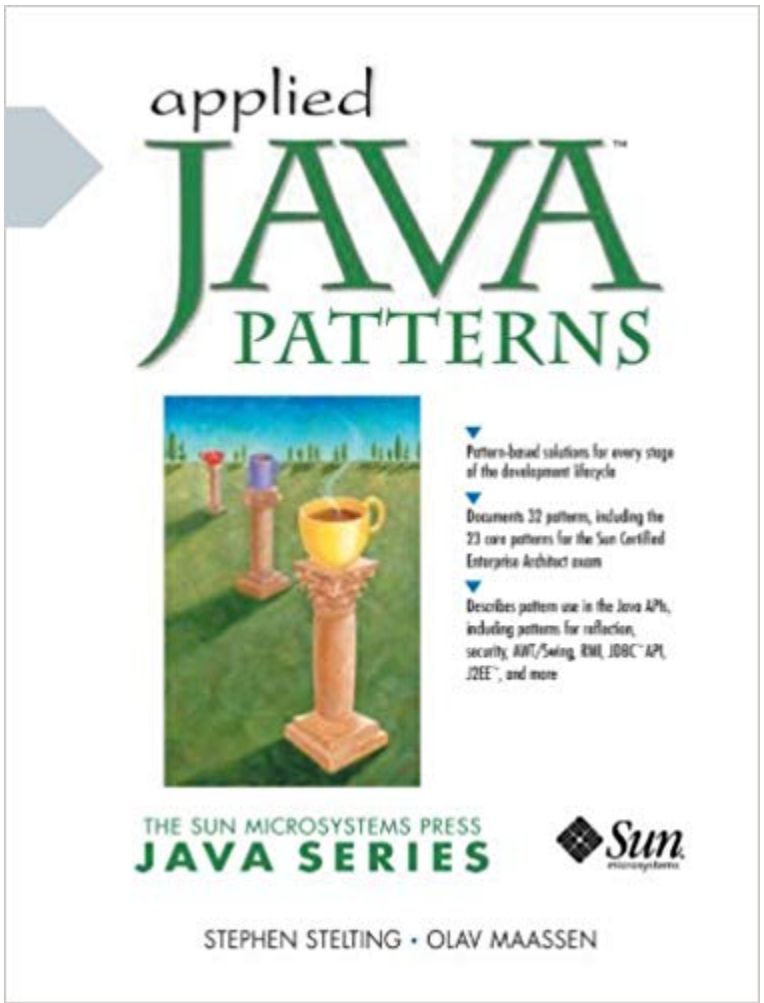
Recommended Books

Head First Design Patterns

Head First Design Patterns
Building Extensible and Maintainable Ob
By Eric Freeman, Elisabeth Robson
Publisher: O'Reilly Media
Release Date: December 2020
Pages: 669

http://shop.oreilly.com/product/9780596007126.do
Available on O'Reilly Safari

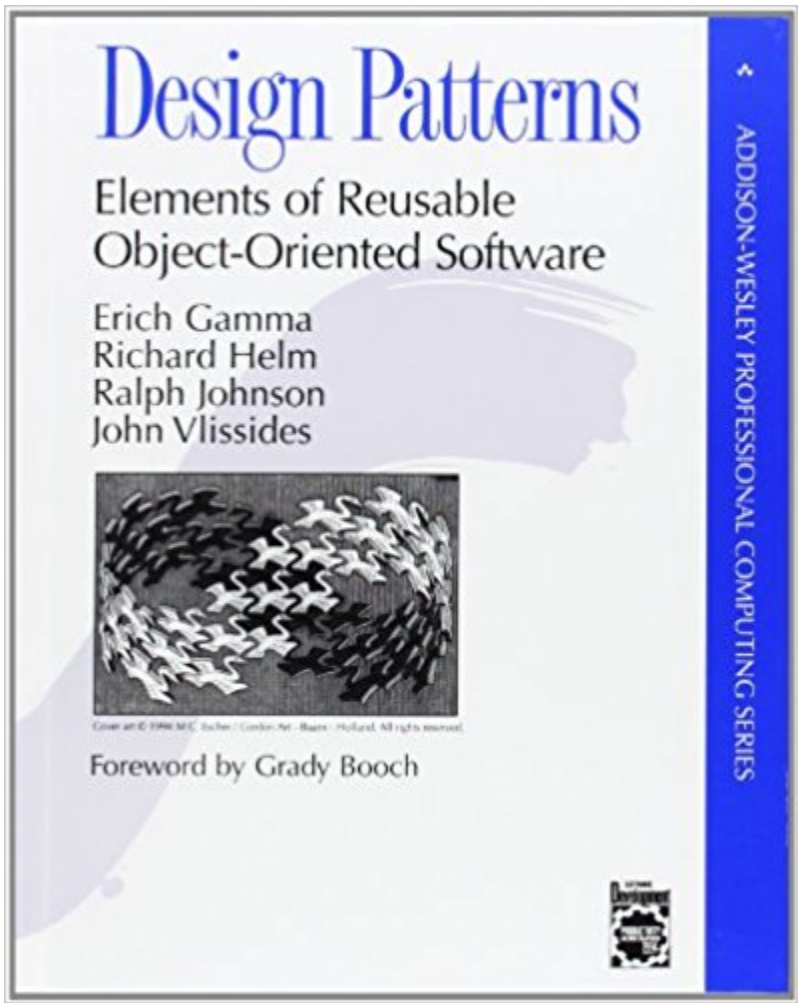
Applied Java Patterns



Applied Java Patterns
By Stephen Stelting, Olav Massen
Publisher: Prentice Hall
Release Date: January 2002
Pages: 608

https://www.amazon.com/Applied-Java-Patterns-Stephen-Stelting/dp/0130935387
Available on O'Reilly Safari

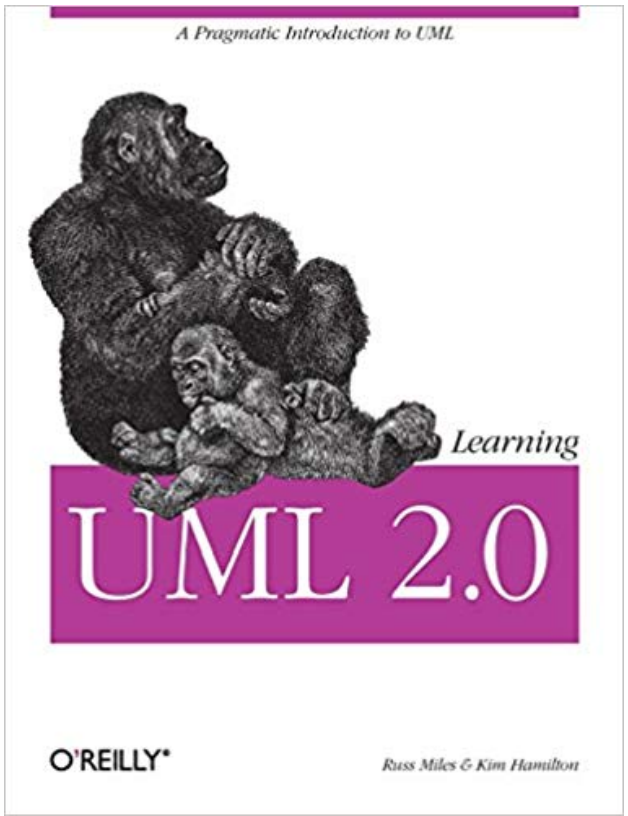
Design Patterns Elements Reusable Object Oriented Software



Design Patterns, Element of Reusable Object Oriented Software:
By Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Grady Booch
Publisher: Addison-Wesley Professional
Release Date: October 31, 1994
Pages: 395

https://www.amazon.com/Design-Patterns-Object-Oriented-Addison-Wesley-Professional-ebook/dp/B000SEIBB8
Available on O'Reilly Safari

Learning UML 2.0



by Kim Hamilton, Russ Miles
Publisher: O'Reilly Media, Inc.
Release Date: April 2006
ISBN: 9780596009823
Topic: Software Development

http://shop.oreilly.com/product/9780596009823.do

Thank You

- Email: dhinojosa@evolutionnext.com
- Github: https://www.github.com/dhinojosa
- Mastodon: https://mastodon.social/@dhinojosa
- Linked In: http://www.linkedin.com/in/dhevolutionnext