Design Patterns

Daniel Hinojosa

Conventions in the slides

The following typographical conventions are used in this material:

Italic
Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
Shows commands or other text that should be typed literally by the user.

`Constant width italic`
Shows text that should be replaced with user-supplied values or by values determined by context.

Contact Information

- Email: dhinojosa@evolutionnext.com
- Github: https://www.github.com/dhinojosa
- Twitter: http://twitter.com/dhinojosa
- Linked In: http://www.linkedin.com/in/dhevolutionnext

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 is 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

Setup

Pre-Class Check

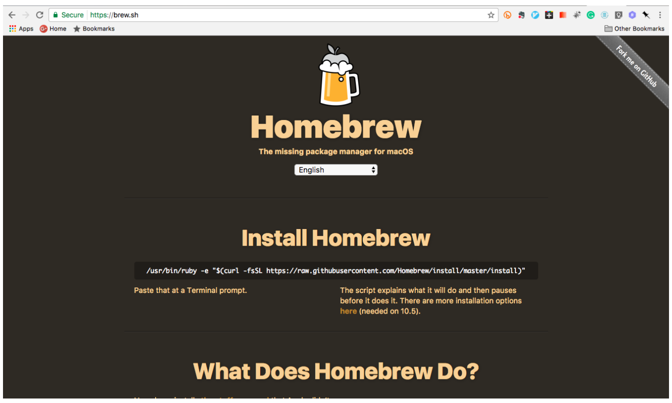
Before we begin it is assumed that all of you have the following tools installed:

- JDK (latest java is 10.0.1)
- Maven (latest maven is 3.5.4)

To verify that all your tools work as expected

```
% javac -version
javac 10.0.1

% java -version
java version "10.0.1"
Java(TM) SE Runtime Environment (build 10.0.1-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)

% mvn -v
Apache Maven 3.5.4 (1edded0938998edf8bf061f1ceb3cfdeccf443fe; 2018-06
-17T12:33:14-06:00)
Maven home: /opt/apache-maven
Java version: 10.0.1, vendor: Oracle Corporation, runtime:
/Library/Java/JavaVirtualMachines/jdk-10.0.1.jdk/Contents/Home
Default locale: en_US, platform encoding: UTF-8
OS name: "mac os x", version: "10.13.5", arch: "x86_64", family: "mac"
```

Installing Java, Maven on a Mac Automatically with Brew



If you have brew installed, you can run the following ***and be done***:

```
% brew update
% brew cask install java
% brew install maven
```

This will require an install of Homebrew. Visit https://brew.sh/ for details of installation if you want to use brew.

Depending on your company’s software and security constraints, you may not be able to use brew

If you don’t have Java installed

- Visit: http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html
- Select: *Accept License Agreement*
- Download the appropriate Java version based on your architecture.

Linux ARM 32 Hard Float ABI	Linux ARM 64 Hard Float ABI
Linux x86	Linux x86
Linux x64	Linux x64
Mac OS X	Solaris SPARC 64-bit
Solaris SPARC 64-bit	Solaris x64
Solaris x64	Windows x86

If you do not have Maven installed

- Visit http://maven.apache.org/download.cgi
- Select either binary tar.gz or .zip

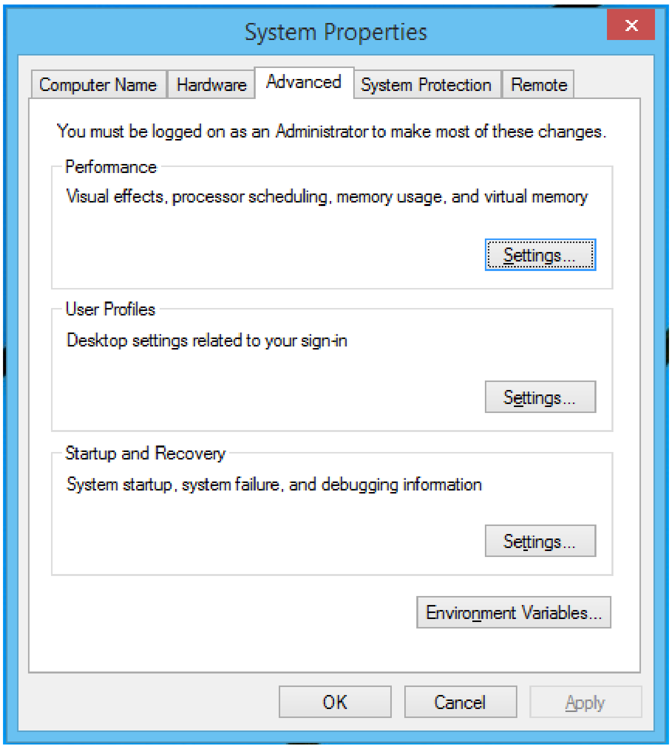
	Link	Checksums	Signature
Binary tar.gz archive	apache-maven-3.5.4-bin.tar.gz	apache-maven-3.5.4-bin.tar.gz.sha1	apache-maven-3.5.4-bin.tar.gz.asc
Binary zip archive	apache-maven-3.5.4-bin.zip	apache-maven-3.5.4-bin.zip.sha1	apache-maven-3.5.4-bin.zip.asc
Source tar.gz archive	apache-maven-3.5.4-src.tar.gz	apache-maven-3.5.4-src.tar.gz.sha1	apache-maven-3.5.4-src.tar.gz.asc
Source zip archive	apache-maven-3.5.4-src.zip	apache-maven-3.5.4-src.zip.sha1	apache-maven-3.5.4-src.zip.asc

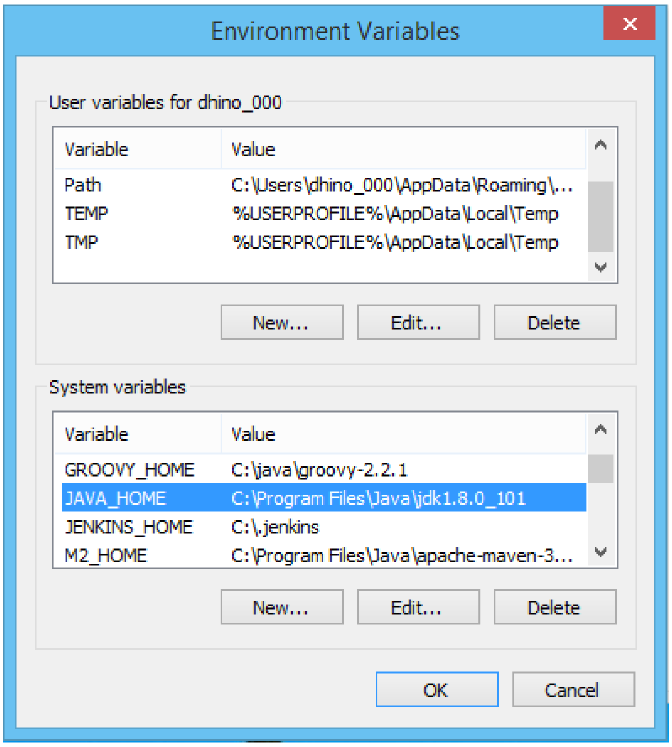
- For Mac and Linux you can expand with `tar -xvf apache-maven-3.5.4.tgz`

Windows Setup

Setting up the Windows Environment Variables for Java

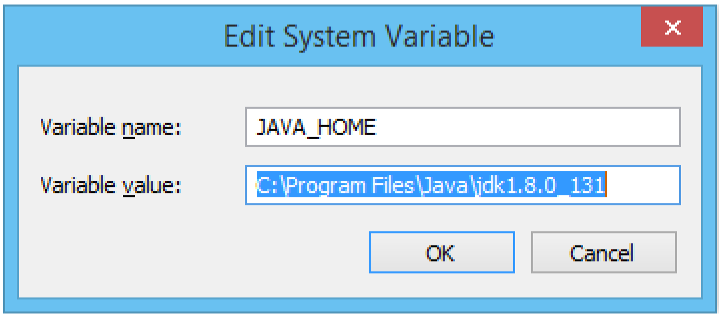
- Go to your *Environment Variables*, typically done by typing the Windows key() and type `env`





Setting up JAVA_HOME Environment Variable

- Edit JAVA_HOME in the System Environment Variable window with the location of your JDK

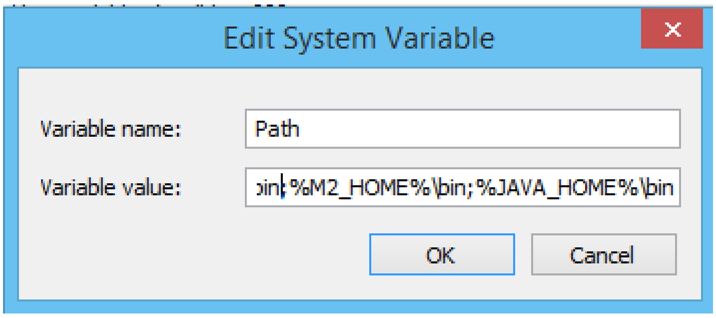


Using jdk1.8.0_131 in the image. Your version may vary.

Setting up PATH Environment Variable

- Once you establish JAVA_HOME, and M2_HOME, *append* to the PATH setting the following:

```
;%JAVA_HOME%\bin;%M2_HOME%\bin
```



Restart All Command Prompts And Try Again

```
% javac -version
javac 10.0.1

% java -version
java version "10.0.1"
Java(TM) SE Runtime Environment (build 10.0.1-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)

% mvn -v
Apache Maven 3.5.4 (1edded0938998edf8bf061f1ceb3cfdeccf443fe; 2018-06
-17T12:33:14-06:00)
Maven home: C:\Program-Files\apache-maven-3.5.4
Java version: 10.0.1, vendor: Oracle Corporation, runtime: C:\Program
Files\Java\jdk-10.0.1.jdk
Default locale: en_US, platform encoding: UTF-8
OS name: "Windows 9", version: "10.13.5", arch: "x86_64", family:
"windows"
```

Changes won't take effect until you open a new command prompt!

Mac OSX Setup

Editing your *.bash_profile* or *.zshrc*

- If you are using the Bash shell, edit the your *.bash_profile* in your home directory using your favorite editor
- If you are using the Zsh shell, edit the your *.zshrc* in your home directory using your favorite editor

For example, if using `nano`

```
% nano ~/.bash_profile
```

Replace *nano* with your favorite editor *vim*, *emacs*, *atom*, etc.

Make sure the following contents are in your *.bash_profile*

```
export JAVA_HOME= <location_of_java>
export M2_HOME= <location_of_mvn>
export PATH=$PATH:$JAVA_HOME/bin:$M2_HOME/bin
```

If you used `brew`, many of these application will not require their `PATH` setup.

You can locate where `mvn` by either doing

```
% which mvn
% whereis mvn
```

When done open a new terminal or if already on an open terminal type:

- For bash: `source .bash_profile`
- For zsh: `source .zshrc`

Verify the Results

Verify the results on the command line

```
% javac -version
javac 10.0.1

% java -version
java version "10.0.1"
Java(TM) SE Runtime Environment (build 10.0.1-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)

% mvn -v
Apache Maven 3.5.4 (1edded0938998edf8bf061f1ceb3cfdeccf443fe; 2018-06
-17T12:33:14-06:00)
Maven home: /opt/apache-maven
Java version: 10.0.1, vendor: Oracle Corporation, runtime:
/Library/Java/JavaVirtualMachines/jdk-10.0.1.jdk/Contents/Home
Default locale: en_US, platform encoding: UTF-8
OS name: "mac os x", version: "10.13.5", arch: "x86_64", family: "mac"
```

Linux Setup

Linux Users Only: Editing your *.bash_profile* or *.zshrc*

- If you are using the Bash shell, edit the your *.bash_profile* in your home directory using your favorite editor
- If you are using the Zsh shell, edit the your *.zshrc* in your home directory using your favorite editor

For example, if using nano

```
% nano ~/.bash_profile
```

Replace *nano* with your favorite editor *vim*, *emacs*, *atom*, etc.

Make sure the following contents are in your *.bash_profile*

```
export JAVA_HOME= <location_of_jdk>
export M2_HOME= <location_of_maven>
export PATH=$PATH:$JAVA_HOME/bin:$M2_HOME/bin
```

You can locate where `mvn` is by either doing

```
% which mvn
% whereis mvn
```

When done open a new terminal or if already on an open terminal type:

- For bash: `source .bash_profile`
- For zsh: `source .zshrc`

Verify the Results

Verify the results on the command line

```
% javac -version
javac 10.0.1

% java -version
java version "10.0.1"
Java(TM) SE Runtime Environment (build 10.0.1-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)

% mvn -v
Apache Maven 3.5.4 (1edded0938998edf8bf061f1ceb3cfdeccf443fe; 2018-06
-17T12:33:14-06:00)
Maven home: /opt/apache-maven
Java version: 10.0.1, vendor: Oracle Corporation, runtime:
/Library/Java/JavaVirtualMachines/jdk-10.0.1.jdk/Contents/Home
Default locale: en_US, platform encoding: UTF-8
OS name: "mac os x", version: "10.13.5", arch: "x86_64", family: "mac"
```

Installing IDEs

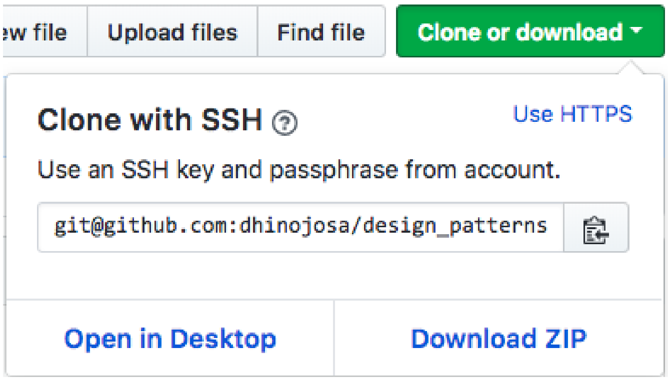
- **IMPORTANT Be sure to download and configure the latest version of your IDE!**
- Eclipse - Be sure to have Photon (Preferable) or Oxygen
- IntelliJ IDEA (Professional or Community): 2018.1 (Preferable) or 2017.3

- **IMPORTANT Be sure to backup any IDE settings that you believe are critical**

> Using IntelliJ IDEA Professional has a wonderful diagramming tool for design patterns

Clone or Download Project

- Project is located at https://github.com/dhinojosa/design_patterns_training
- If you are familiar with Git, clone the project:



- If you are not familiar with Git, you can download the project by clicking the "Download ZIP" on the bottom right

Import into IntelliJ

- If you have no current projects open:
    - Select **Import Project**
    - Locate the *design_patterns_training* project
- If you have a current project open and you want to keep it open:
    - **CTRL+SHIFT+A** or **CMD+SHIFT+A** and type "Import Project"
    - Locate the *design_patterns_training* project

Import into Eclipse

- Import the project, using **File | Import**
- Select **Maven | Existing Maven Projects**
- Locate the *design_patterns_training* project
- Click **Finish**

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

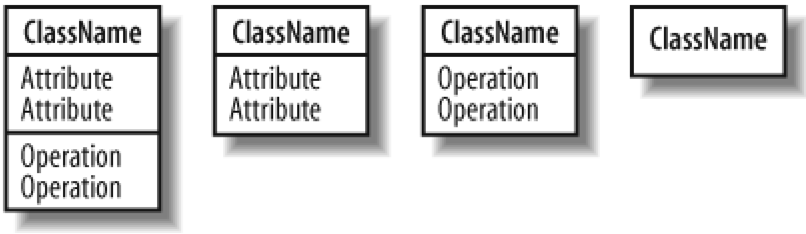
- Provide a detailed specification of a system with UML diagrams
- Would not be disposable but would be generated with a UML tool
- Generally associated with software systems and usually involves keeping models synchronized with the code

UML as a programming language

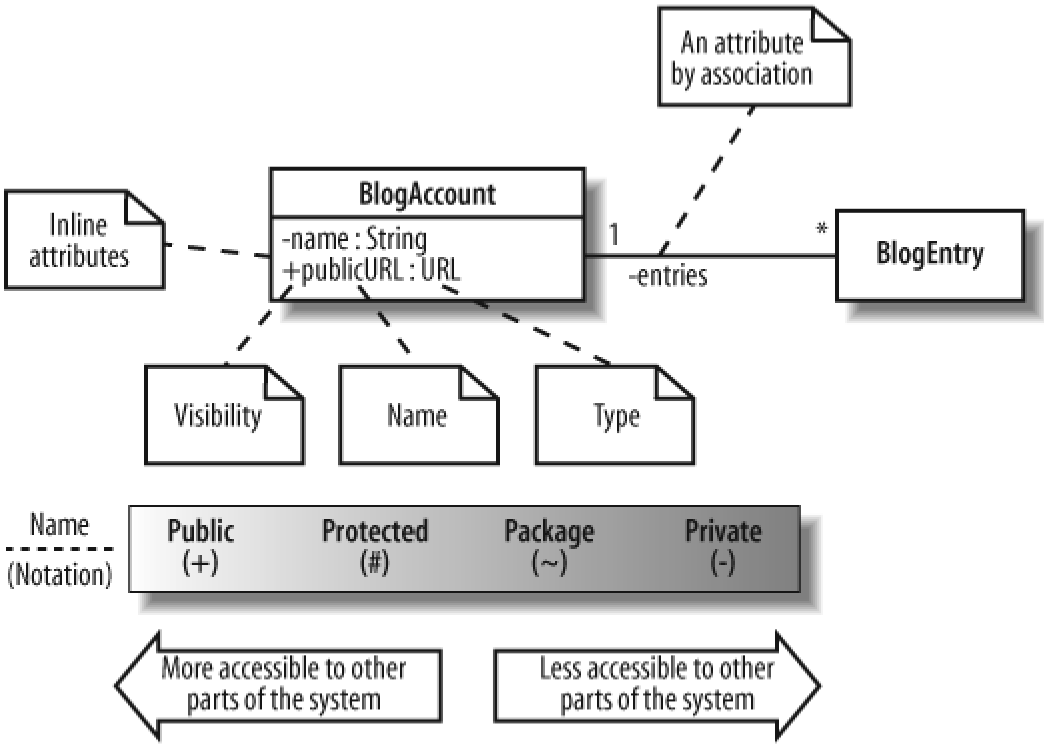
- UML model to executable code
- Meaning that every aspect of the system is modeled
- You can keep your model indefinitely and use transformations and code generation to deploy to different environments

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

Classes

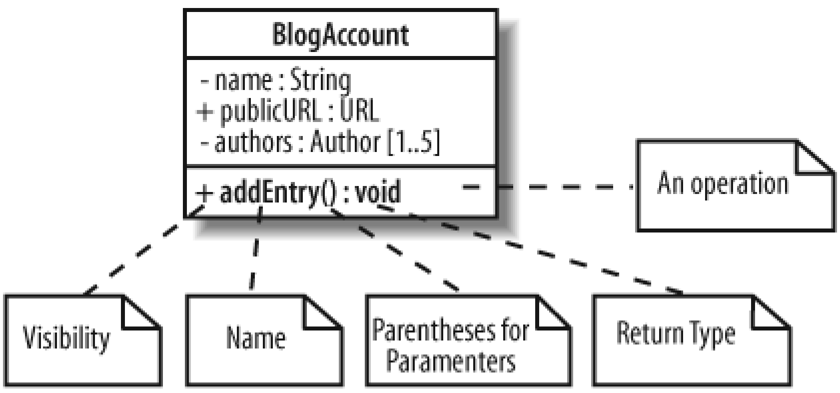


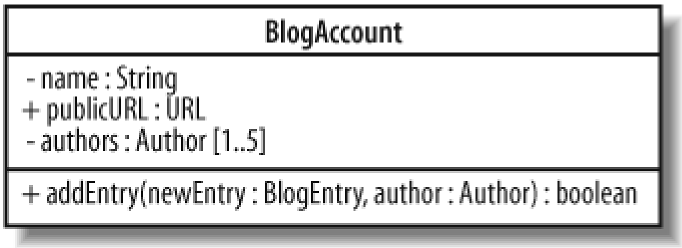
Atributes and Relations in UML



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

Class Methods

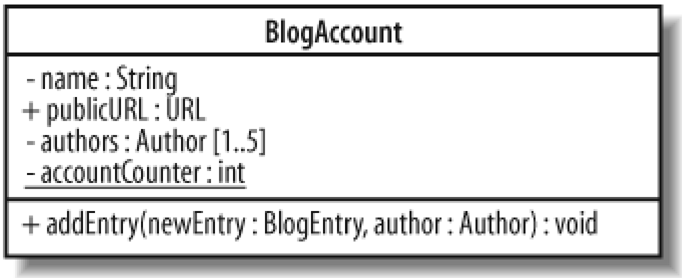




Source: Learning UML 2.0 by Russ Miles; Kim Hamilton Published by O'Reilly Media, Inc., 2006

Static Fields

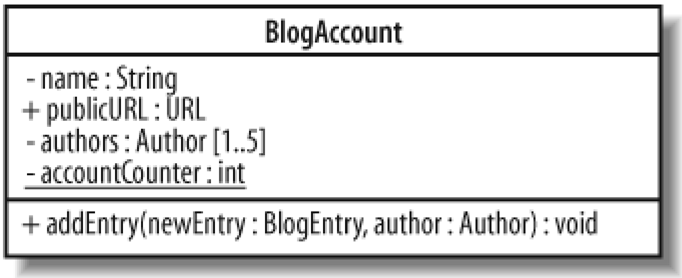
`static` is conveyed with an underline



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

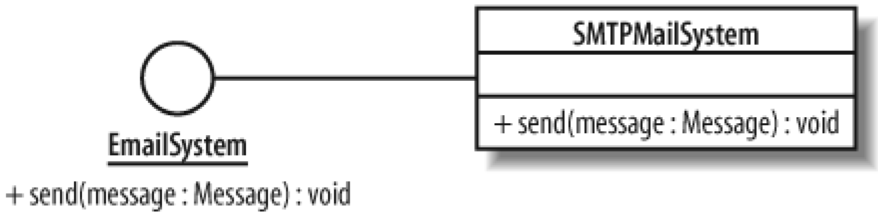


Figure 1. Ball Notation

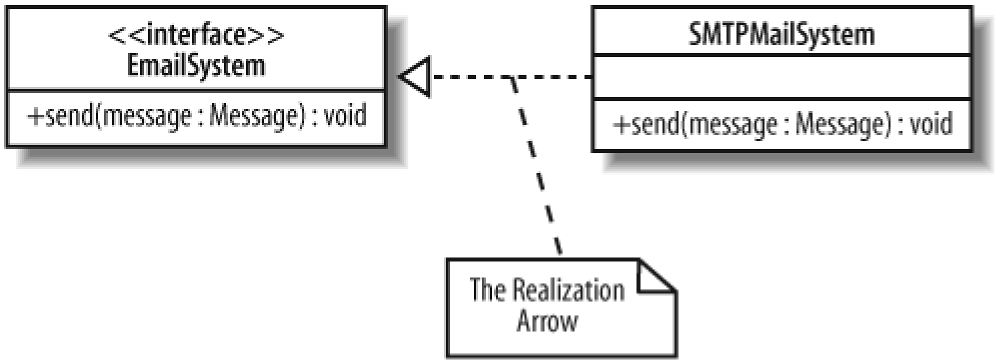
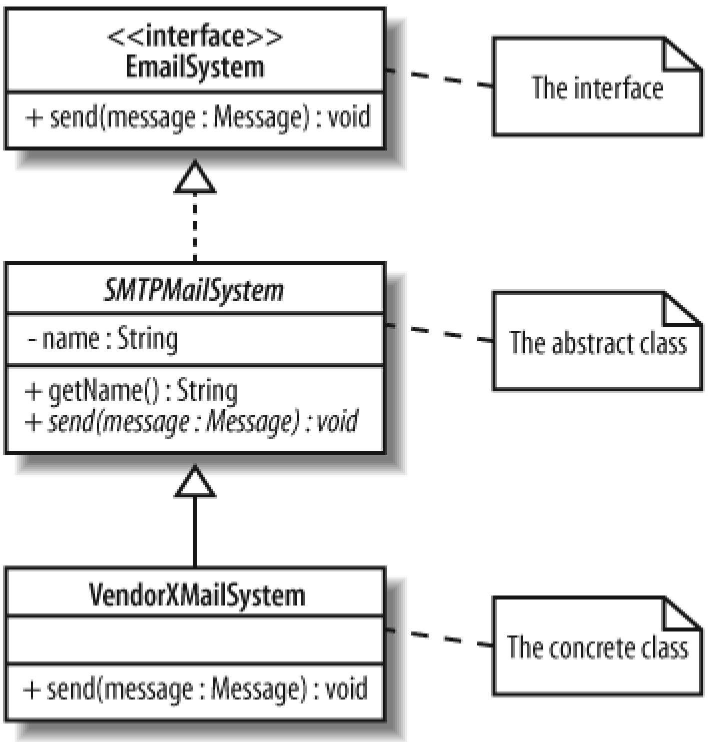


Figure 2. Stereotype Notation

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

Concrete to Abstract to Interface

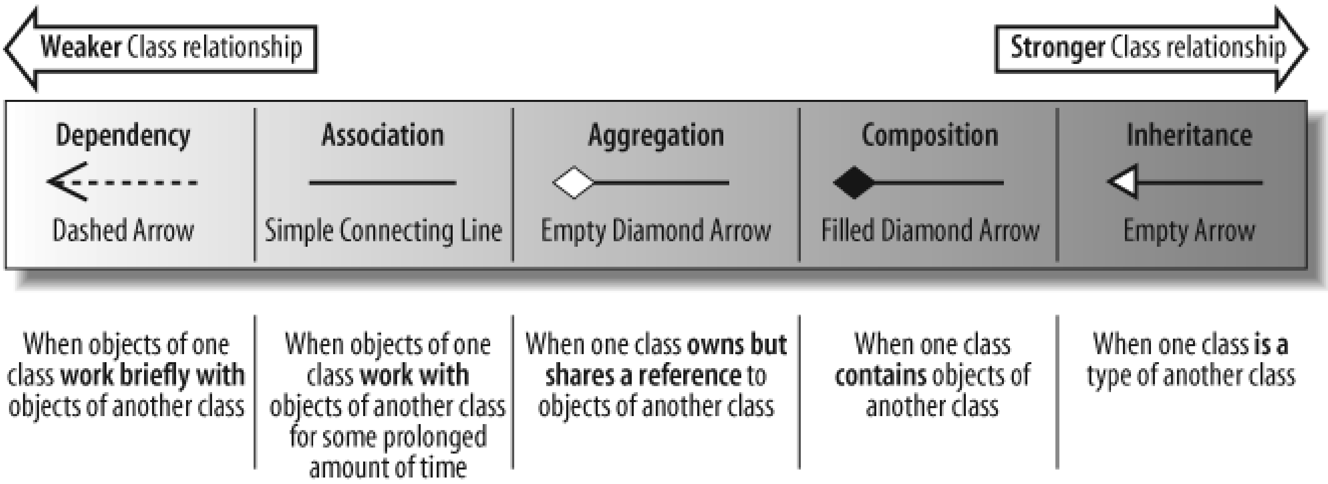
Here is a complete inheritance chain using concrete, `abstract`, and `interface`



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

Dependencies

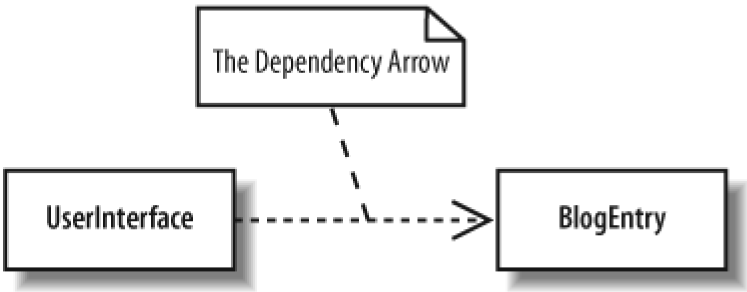
Diagram of all the different kinds of dependencies



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

Use Dependency

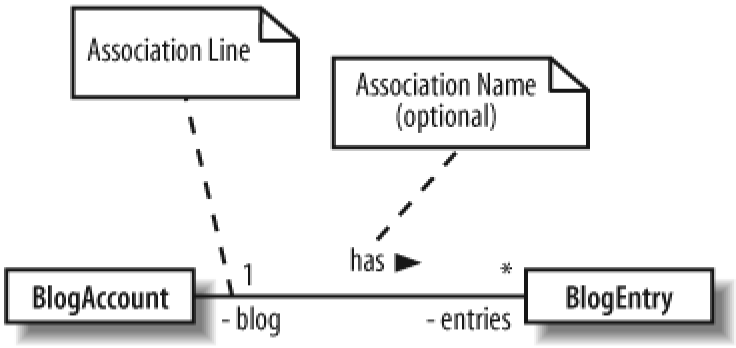
A standard dependency declares that a class needs to know about another class to use objects of that class



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

Association Dependency

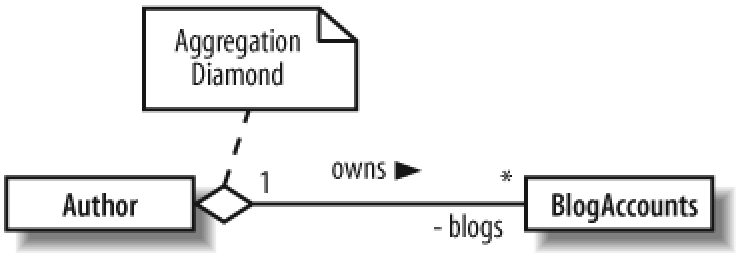
Association - A class will actually contain a reference to an object, or objects, of the other class in the form of an attribute



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

Aggregation Dependency

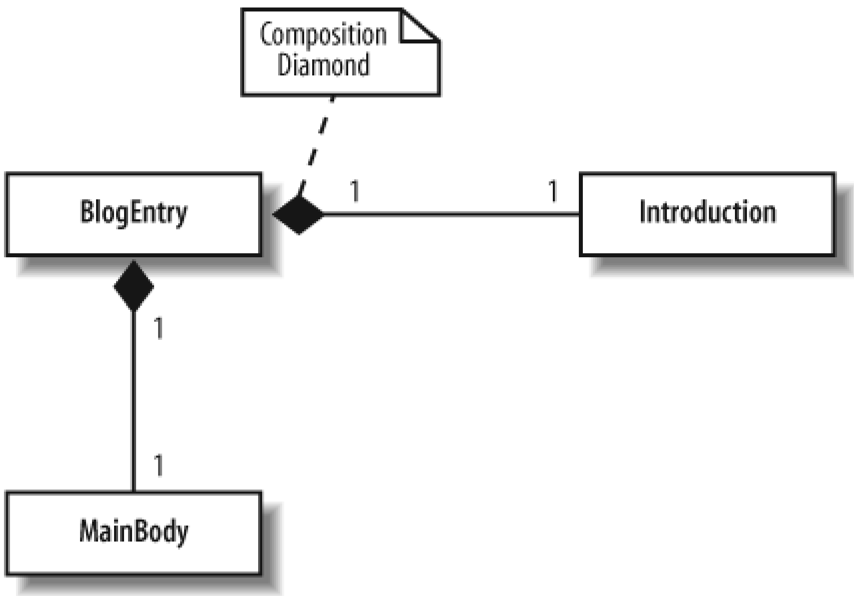
A stronger version of association and is used to indicate that a class actually owns but may share objects of another class.



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

Composition Dependency

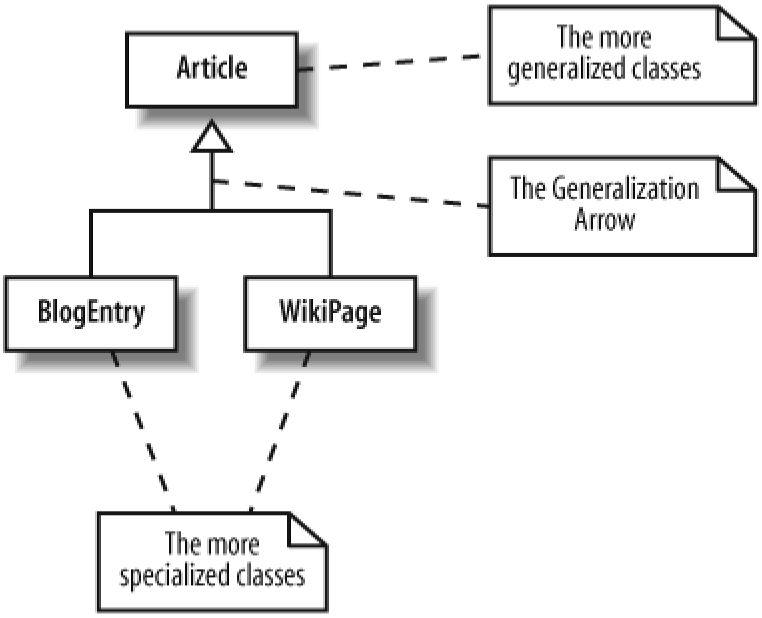
Composition – association of an element or elements that are not exposed



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

Generalization

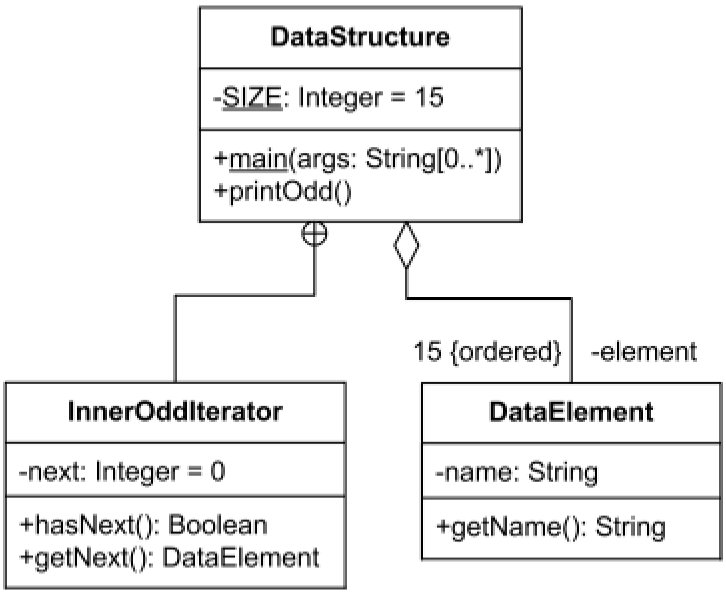
Inheritance - One type has a "is-a" relationship with another



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: https://www.uml-diagrams.org/nested-classifier.html

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.
>
> — Bob Martin, Clean Code 2008

Law 2: Keep Unit Tests Light

> You may not write more of a unit test than is sufficient to fail
>
> — Bob Martin, Clean Code 2008

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.
>
> — Bob Martin, Clean Code 2008

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:

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
```

Lab: Test Driven Development

Step 1: In design_patterns_training, ensure that a *src/test/java* folder exists and a package called `com.xyzcorp.tdd`, if not create them.

Step 2: In the `com.xyzcorp.tdd` package, create a test class called `ProgrammerTest`

Step 3: In the `ProgrammerTest`, create a set of tests using Test Driven Development that will inevitably develop the following code:

Test the full name

```
import java.time.*

Programmer programmer = new Programmer(
   "Bjarne", "Strousoup", LocalDate.of(1950, 12, 30), () -> LocalDate.
of(2018, 6, 1))
assertEquals("Bjarne Strousoup", programmer.getFullName)
```

Test the age

```
import java.time.*

Programmer programmer = new Programmer(
   "Bjarne", "Strousoup", LocalDate.of(1950, 12, 30), () -> LocalDate.
of(2018, 6, 1))
assertEquals(67, programmer.getAge())
```

Step 4: Answer the question: Why the function `() → LocalDate.of(2018, 6, 1)`? What purpose does it serve?

Step 5: If you have extra time, add tests for `toString`, `equals`, and `hashCode`

Design by Contract (DbC)

Benefits of Design by Contract

The benefits of Design by Contract include the following:

- A better understanding of the object-oriented method and, more generally, of software construction
- A systematic approach to building bug-free object-oriented systems
- An effective framework for debugging, testing and, more generally, quality assurance
- A method for documenting software components
- Better understanding and control of the inheritance mechanism
- A technique for dealing with abnormal cases, leading to a safe and effective language construct for exception handling

Bertrand Meyer, Eiffel

https://archive.eiffel.com/doc/manuals/technology/contract

Defining Precisely

The immediate objection is that specifying a module's purpose will not ensure that it will achieve that specification; this is obviously true, but:

- One may reverse this proposition and note that it we don't state what a module should do, there is little likelihood that it will do it. (The law of excluded miracles.)
- In practice, it is amazing to see how far just stating what a module should do goes towards helping to ensure that it does it.

The Design by Contract theory, then, suggests associating a specification with every software element. These specifications (or contracts) govern the interaction of the element with the rest of the world.

https://archive.eiffel.com/doc/manuals/technology/contract

The notion of a contract

Table 1. Table of Contracts

-	Obligations	Benefits
Client	Ensure Precondition: *Be at the Santa Barbara airport at least 5 minutes before scheduled departure time. Bring only acceptable baggage. Pay ticket price.*	Benefit from Postcondition: *Reach Chicago*

Supplier	Ensure Postcondition: *Bring customer to Chicago*	Assume Precondition: *No need to carry passenger who is late, has unacceptable baggage, or has not paid ticket price*

https://archive.eiffel.com/doc/manuals/technology/contract

Assuming the same rules for software

- The same ideas apply to software.
- Consider a software element `E`, to achieve its purpose (fulfil its own contract):
  - `E` uses a certain strategy, which involves a number of subtasks, `t1`, … `tn`.
  - If subtask `ti` is non-trivial, it will be achieved by calling a certain routine `R`.
  - In other words, `E` contracts out the subtask to R.

Such a situation should be governed by a well-defined roster of obligations and benefits — **a contract**

https://archive.eiffel.com/doc/manuals/technology/contract

The notion of a contract in software

Table 2. Table of Contracts for a Map

-	Obligations	Benefits
Client	Ensure Precondition: *Make sure table is not full and key is a non-empty string*	Benefit from Postcondition: *Get updated table where the given element now appears, associated with the given key*
Supplier	Ensure Postcondition: *Record given element in table, associated with given key*	Assume Precondition: *No need to do anything if table is full, or key is empty string*

https://archive.eiffel.com/doc/manuals/technology/contract

Example of a contract

The following is in the language, Eiffel

```
put (x: ELEMENT; key: STRING) is
   -- Insert x so that it will be retrievable through key.
   require
     count <= capacity
     not key.empty
   do
     ... Some insertion algorithm ...
   ensure
     has (x)
     item (key) = x
     count = old count + 1
end
```

- The `require` clause introduces an input condition, or precondition
- The `ensure` clause introduces an output condition, or postcondition.
- Both of these conditions are examples of assertions, or logical conditions (contract clauses) associated with software elements.

https://archive.eiffel.com/doc/manuals/technology/contract

Invariants

- Preconditions and postconditions apply to individual routines.
- Other kinds of assertions will characterize a class as a whole, rather than its individual routines.
- An assertion describing a property which holds of all instances of a class is called a class *invariant*.

```
class interface DICTIONARY [ELEMENT] feature
   put (x: ELEMENT; key: STRING) is
     -- Insert x so that it will be retrievable
     -- through key.
     require
        count <= capacity
        not key.empty
     ensure
        has (x)
        item (key) = x
        count = old count + 1
     ... Interface specifications of other features ...

  invariant
     0 <= count
     count <= capacity

end -- class interface DICTIONARY
```

https://archive.eiffel.com/doc/manuals/technology/contract

Further notes on `invariant`

- `invariant` are consistency constraints.
- Important for configuration management and regression testing,
- It describes the deeper properties of a class
- It also describes the constraints which must also apply to subsequent changes.
- Viewed from the contract theory, an invariant is a general clause which applies to the entire set of contracts defining a class

https://archive.eiffel.com/doc/manuals/technology/contract

DbC and Testing

- We should be able to prove mathematically that the routine implementations are consistent with the assertions
- Powerful tools for finding mistakes

https://archive.eiffel.com/doc/manuals/technology/contract

Contracts and Subclassing

- An subclass or subcontractor ***must*** be bound to original semantic contract
- In DbC, Strengthening the precondition, or weakening the postcondition, would be a case of "dishonest subcontracting" and could lead to disaster
- See also, Liskov Substitution Principle:

> If S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e. an object of type T may be substituted with any object of a subtype S) without altering any of the desirable properties of the program (correctness, task performed, etc.)
>
> — Barbara Liskov, Liskov Substitution Principle

Exceptions

- An exception is an inability to fulfill a contract
- The only responses that make sense:
  a. Retrying: an alternative strategy is available. The routine will restore the invariant and make another attempt, using the new strategy
  b. Organized panic: no such alternative is available. Restore the invariant, terminate, and report failure to the caller by triggering a new exception

c. False alarm: it is in fact possible to continue, perhaps after taking some corrective measures. This case seldom occurs

Exceptions in Eiffel

```
attempt_transmission (message: STRING) is
  -- Attempt to transmit message over a communication line
  -- using the low-level (e.g. C) procedure unsafe_transmit,
  -- which may fail, triggering an exception.
  -- After 100 unsuccessful attempts, give up (triggering
  -- an exception in the caller).

  local
    failures: INTEGER
  do
    unsafe_transmit (message)

  rescue
    failures := failures + 1
    if failures < 100 then
       retry
  end
end
```

https://archive.eiffel.com/doc/manuals/technology/contract

Design by Contract Conclusion

- Design by Contract has already been widely applied
- The theory provides a powerful thread throughout the object-oriented method
- Addresses many of the issues that many people are encountering as they start applying O-O techniques and languages seriously
  - What kind of "methodology" to apply
  - What concepts to base the analysis step
  - How to specify components
  - How to document object-oriented software
  - How to guide the testing process
  - How to build software so that bugs do not show up in the first place.

In software development, reliability should be built-in, not an afterthought.

Creational Patterns

Factory Method Pattern

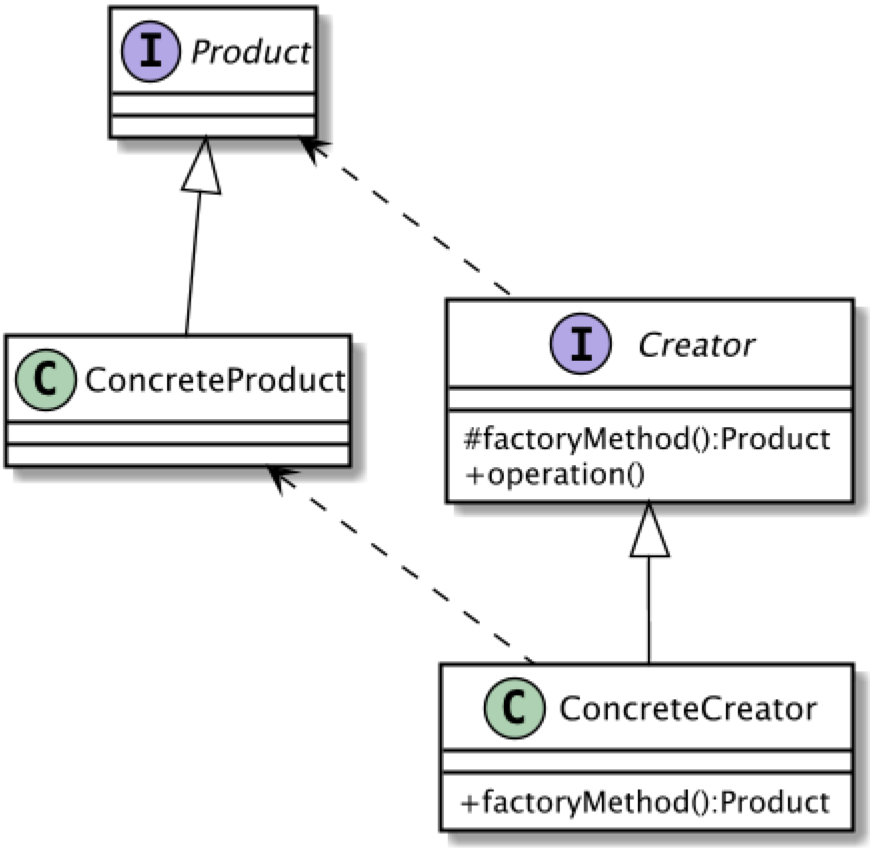
Factory Method Pattern Properties

- **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

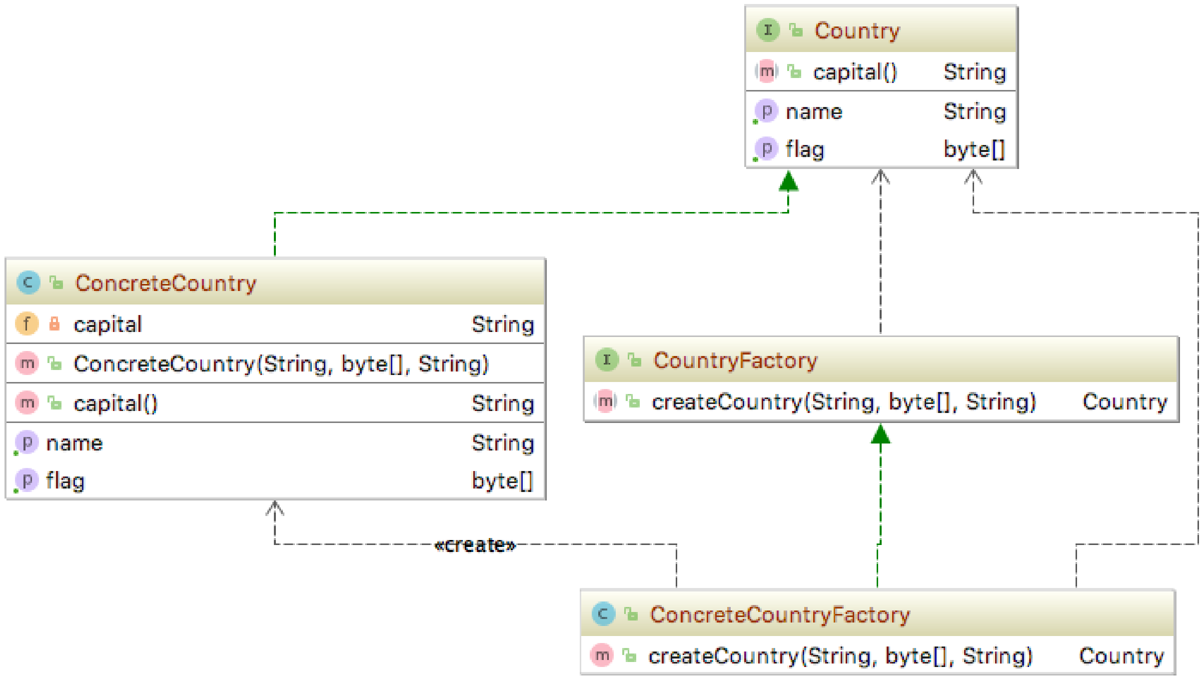
ConcreteProduct – The implementing class of **Product**. Objects of this class are created by the **ConcreteCreator**.

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

Factory Method: Product

```
public interface Country {
    public String getName();

    public byte[] getFlag();

    public String capital();
}
```

Factory Method: Concrete Product

```
public class ConcreteCountry implements Country {

    private String name;
    private String capital;
    private byte[] flagBytes;

    public ConcreteCountry(String name, byte[] flagBytes, String
capital) {
        this.name = name;
        this.capital = capital;
        this.flagBytes = flagBytes;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public byte[] getFlag() {
        return flagBytes;
    }

    @Override
    public String capital() {
        return capital;
    }
}
```

Factory Method: Creator

```
public interface CountryFactory {
    public Country createCountry(String name, byte[] flag, String
capital);
}
```

Factory Method: Concrete Creator

```
public class ConcreteCountryFactory implements CountryFactory {
    @Override
    public Country createCountry(String name, byte[] flag, String
capital) {
        return new ConcreteCountry(name, flag, capital);
    }
}
```

Builder Pattern

Builder Pattern Properties

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

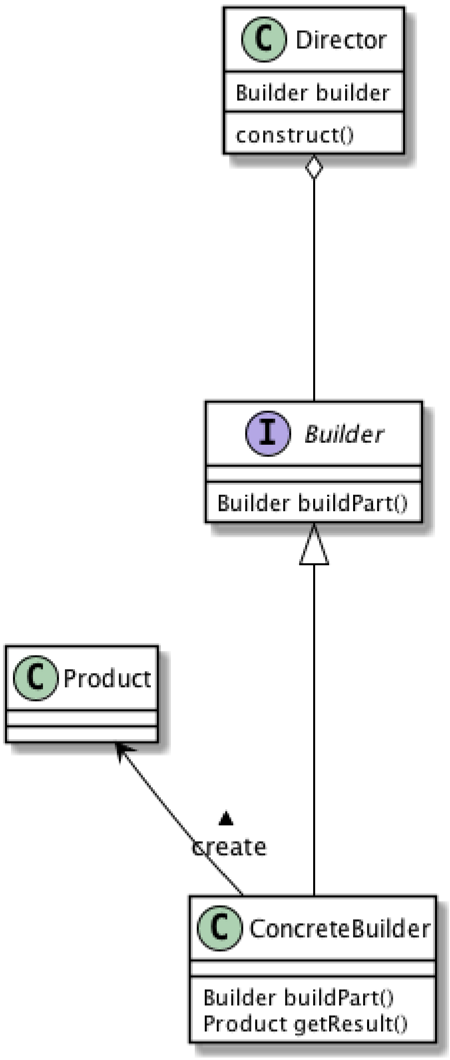
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

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

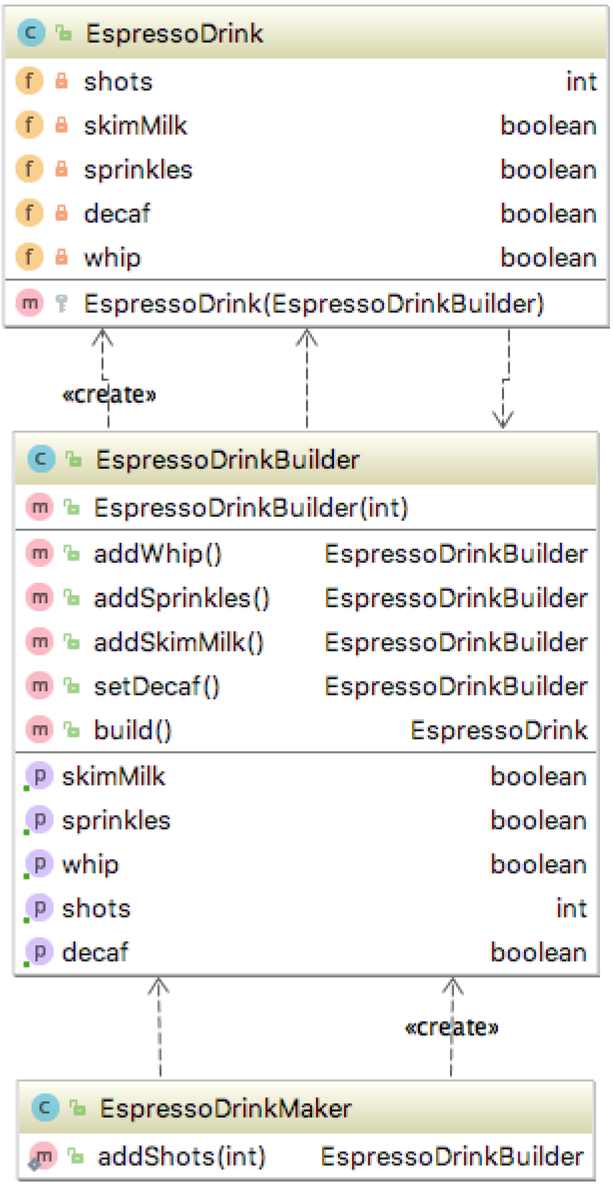
Builder Pattern Advantages

- 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 Demo Diagram



Builder: Director

```
public class EspressoDrinkMaker {

    public static EspressoDrinkBuilder addShots(int i) {
        return new EspressoDrinkBuilder(i);
    }
}
```

Builder: Concrete Builder

```java
public class EspressoDrinkBuilder {
    private boolean whip;
    private boolean sprinkles;
    private int shots;

    public EspressoDrinkBuilder(int shots) {
        this.shots = shots;
    }

    public EspressoDrinkBuilder addWhip() {
        this.whip = true;
        return this;
    }

    public EspressoDrinkBuilder addSprinkles() {
        this.sprinkles = true;
        return this;
    }

    public EspressoDrink build() {
        return new EspressoDrink(this);
    }
}
```

Builder: Product

```java
public class EspressoDrink {
    private int shots;
    private boolean sprinkles;
    private boolean whip;

    protected EspressoDrink(EspressoDrinkBuilder espressoDrinkBuilder) {
        this.shots = espressoDrinkBuilder.getShots();
        this.sprinkles = espressoDrinkBuilder.isSprinkles();
        this.whip = espressoDrinkBuilder.isWhip();
    }
}
```

Lab: Making a Builder

Step 1: In design_patterns_training, ensure that a *src/test/java* folder exists and a package called `com.xyzcorp.builder`, if not create them.

Step 2: In the `com.xyzcorp.builder` package, create a test class called `BuilderTest`

Step 3: In the `BuilderTest`, create a set of tests using Test Driven Development that will inevitably develop the following code:

```
Flower f = Flower.builder()
  .petals(7)
  .color("Yellow")
  .latinName("Narcissus")
  .build();
```

Step 4: Ensure that your tests will test different combinations

Lab: Using a Prefabricated Builder

Step 1: In the *src/test/java* folder an in the `com.xyzcorp.builder` package, create a test class called `GuavaBuilderTest`

Step 2: Guava is amazing with a wealth of utilities. Go to http://javadoc.scijava.org/Guava and look around

Step 3: In the website that you just open go to the `com.google.common.collect` package, and select either one of these interesting immutable collections:

- `ImmutableBiMap`
- `ImmutableMultiMap`
- `ImmutableMultiSet`
- `ImmutableSortedMultiSet`

Step 4: Given your choice, write a battery of tests that shows what that collection does, be prepared to explain it. All of these use a Builder pattern.

Singleton Pattern

Singleton Pattern Properties

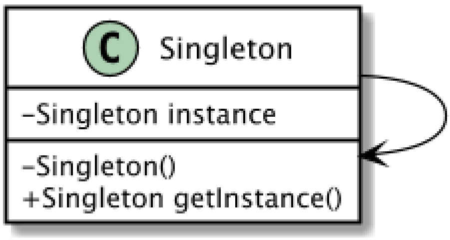
- **Type:** Creational
- **Level:** Object

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

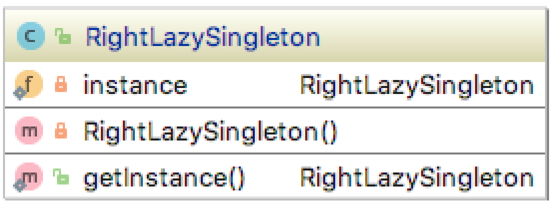


Singleton Ingredients

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

Singleton: An Eager Implementation

```
public class EagerSingleton {
    private static EagerSingleton instance = new EagerSingleton();

    private EagerSingleton() {
    }

    public static EagerSingleton getInstance() {
        return instance;
    }
}
```

Singleton: A Lazy Non-Thread-Safe Implementation

```java
public class WrongLazySingleton {
    private static WrongLazySingleton instance = null;

    private WrongLazySingleton() { }

    public static WrongLazySingleton getInstance() {
        if (instance == null) {
            instance = new WrongLazySingleton();
        }
        return instance;
    }
}
```

Singleton: A Lazy Thread-Safe Implementation

```java
public class RightLazySingleton {
    private static RightLazySingleton instance;

    private RightLazySingleton() {}

    public static synchronized RightLazySingleton getInstance() {
        if (instance == null) {
            instance = new RightLazySingleton();
        }
        return instance;
    }
}
```

`synchronized` would obtain a lock on the class since the method is `static`

Singleton: An `enum` of one

Recommended way to create a Singleton in Java:

```java
public enum EnumSingleton {
     INSTANCE;
}
```

To use this...

```java
var a = EnumSingleton.INSTANCE;
var b = EnumSingleton.INSTANCE;

a.equals(b);
a.hashCode() == b.hashCode();
a.toString().equals(b.toString());
```

The above uses Java 10’s new `var`

Factory Method Pattern

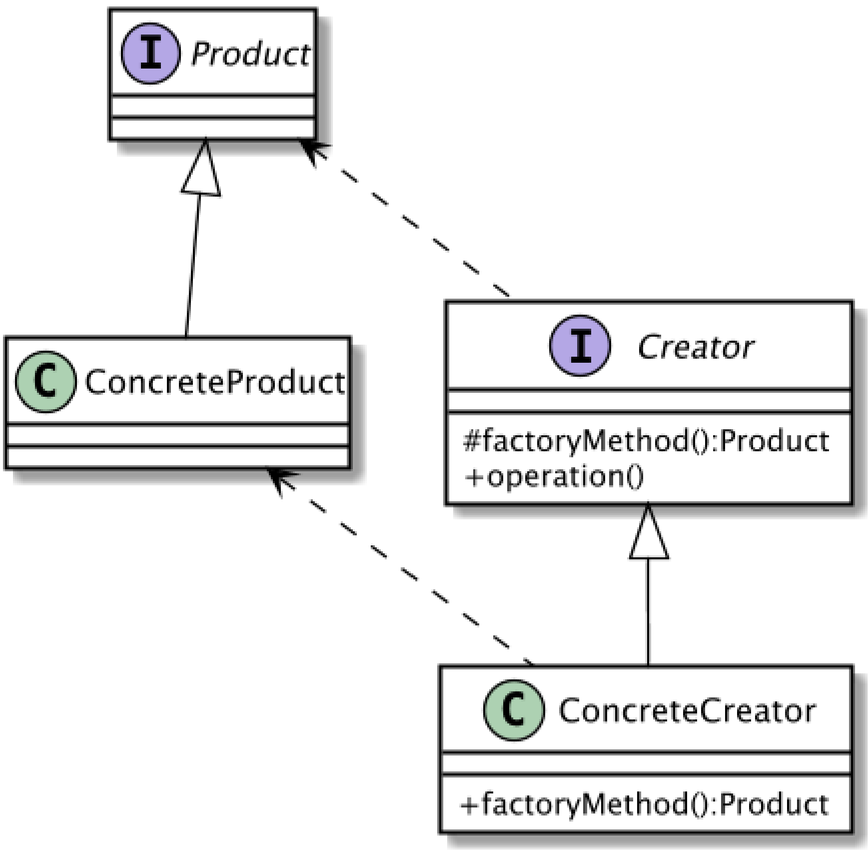
Factory Method Pattern Properties

- **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

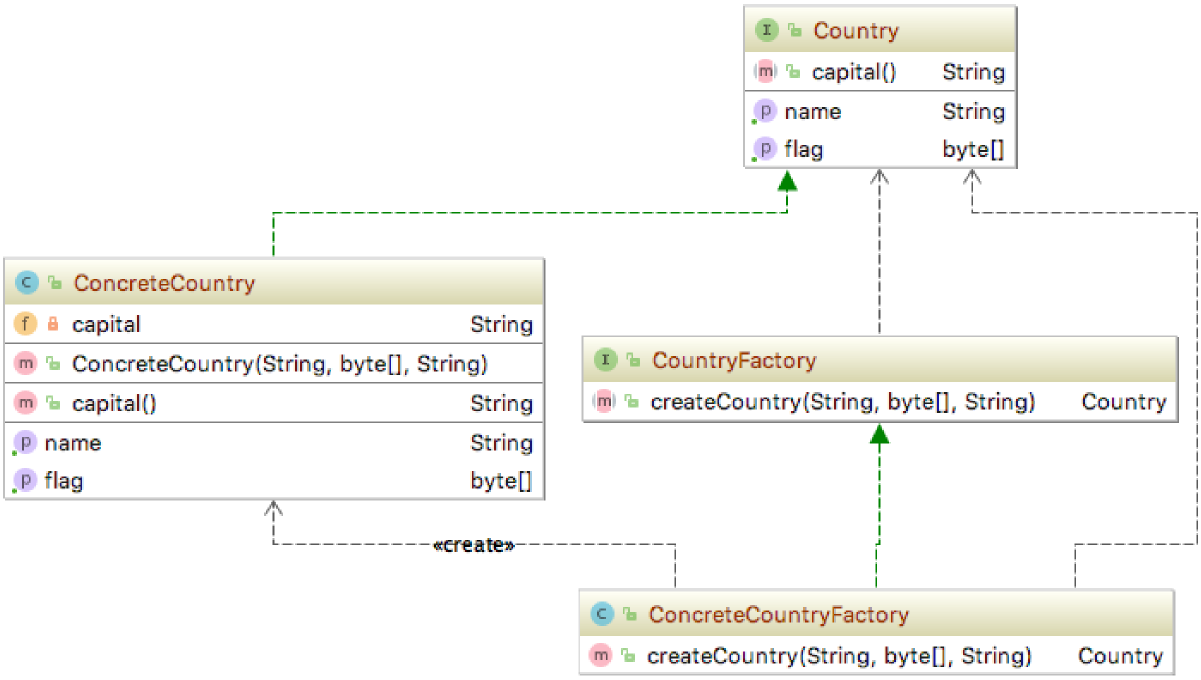
ConcreteProduct – The implementing class of **Product**. Objects of this class are created by the **ConcreteCreator**.

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

Factory Method: Product

```
public interface Country {
    public String getName();

    public byte[] getFlag();

    public String capital();
}
```

Factory Method: Concrete Product

```
public class ConcreteCountry implements Country {

    private String name;
    private String capital;
    private byte[] flagBytes;

    public ConcreteCountry(String name, byte[] flagBytes, String
capital) {
        this.name = name;
        this.capital = capital;
        this.flagBytes = flagBytes;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public byte[] getFlag() {
        return flagBytes;
    }

    @Override
    public String capital() {
        return capital;
    }
}
```

Factory Method: Creator

```
public interface CountryFactory {
    public Country createCountry(String name, byte[] flag, String
capital);
}
```

Factory Method: Concrete Creator

```
public class ConcreteCountryFactory implements CountryFactory {
    @Override
    public Country createCountry(String name, byte[] flag, String
capital) {
        return new ConcreteCountry(name, flag, capital);
    }
}
```

Abstract Factory Pattern

Abstract Factory Pattern Properties

Type: Creational, Object
Level: Component

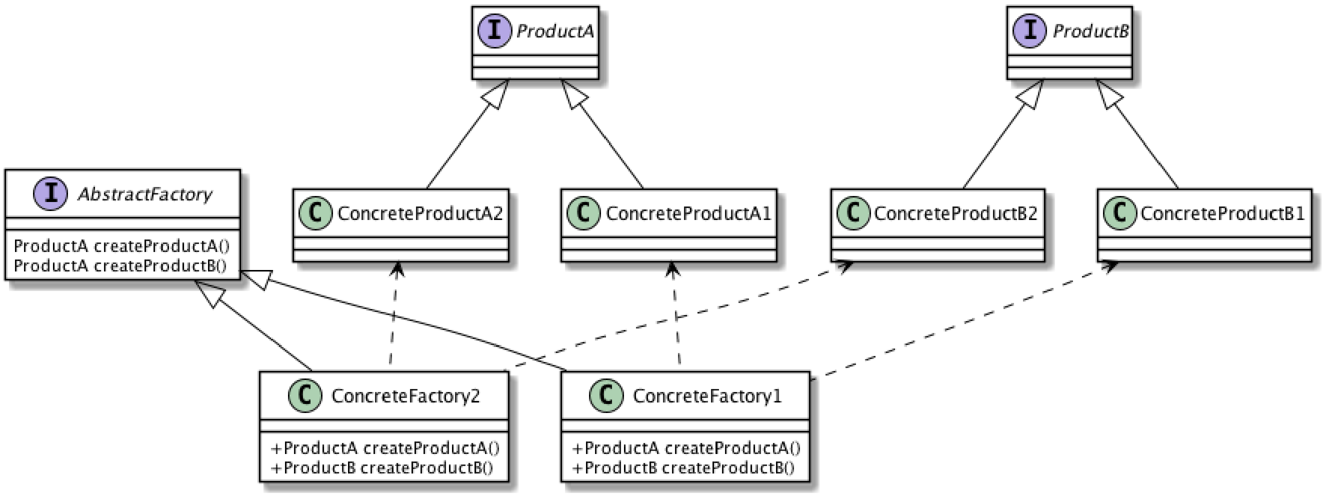
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

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.

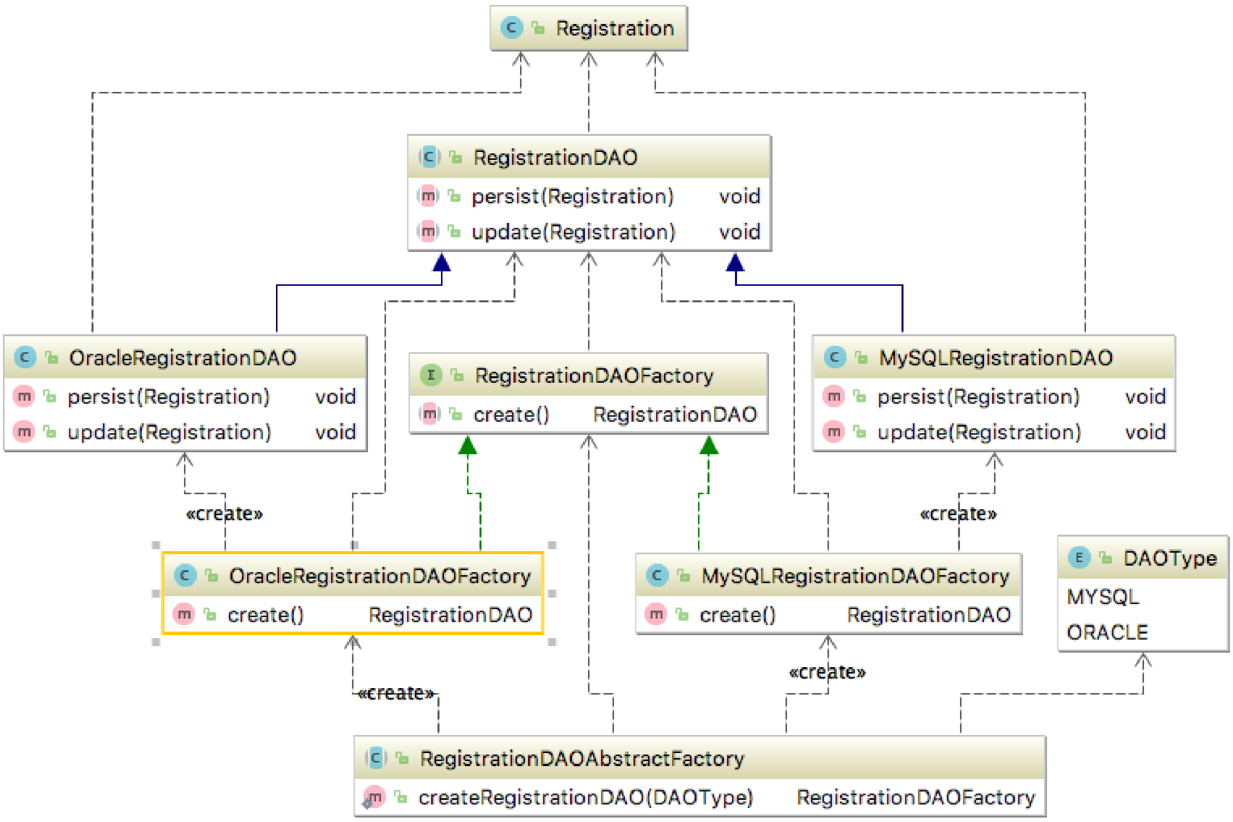
AbstractProduct – An abstract class or interface describing the general behavior of the resource that will be used by the application.

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

Abstract Factory: AbstractFactory

```java
public abstract class RegistrationDAO {
    public abstract void persist(Registration registration);
    public abstract void update(Registration registration);
}
```

Abstract Factory: ConcreteFactory1

```
public class OracleRegistrationDAOFactory extends RegistrationDAOFactory
{
   public RegistrationDAO create() {
      return new OracleRegistrationDAO();
   }
}
```

Abstract Factory: ConcreteFactory2

```
public class MySQLRegistrationDAOFactory extends RegistrationDAOFactory
{
   public RegistrationDAO create() {
      return new MySQLRegistrationDAO();
   }
}
```

Abstract Factory: Abstract Product

```
public abstract class RegistrationDAO {
   public abstract void setDataSource(DataSource dataSource);
   public abstract void persist(Registration registration);
   public abstract void update(Registration registration);
}
```

Abstract Factory: Concrete Product

```
public class MySQLRegistrationDAO extends RegistrationDAO {
   private DataSource dataSource;

   public MySQLRegistrationDAO() { }

   @Override
   public void setDataSource(DataSource dataSource) {..}

   @Override
   public void persist(Registration registration) {..}

   @Override
   public void update(Registration registration) {..}
}
```

Behavioral Patterns

State Pattern

State Pattern Properties

Type: Behavioral
Level: Object

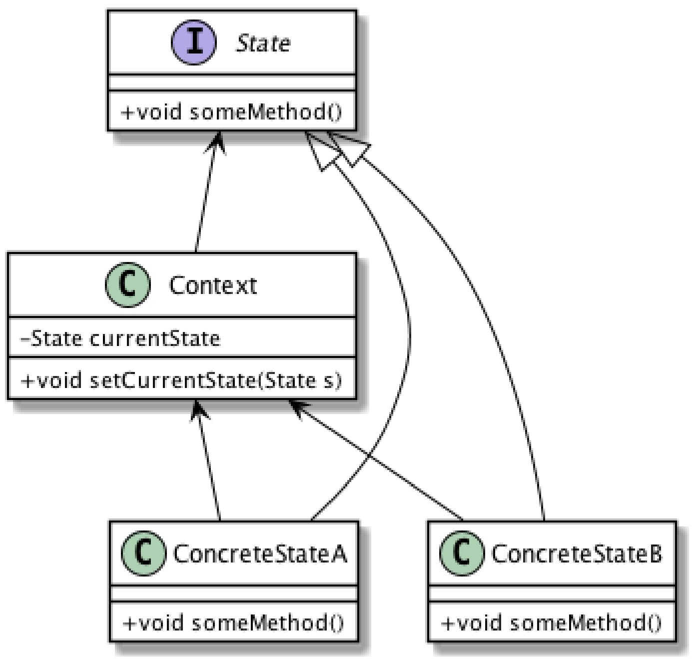
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

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

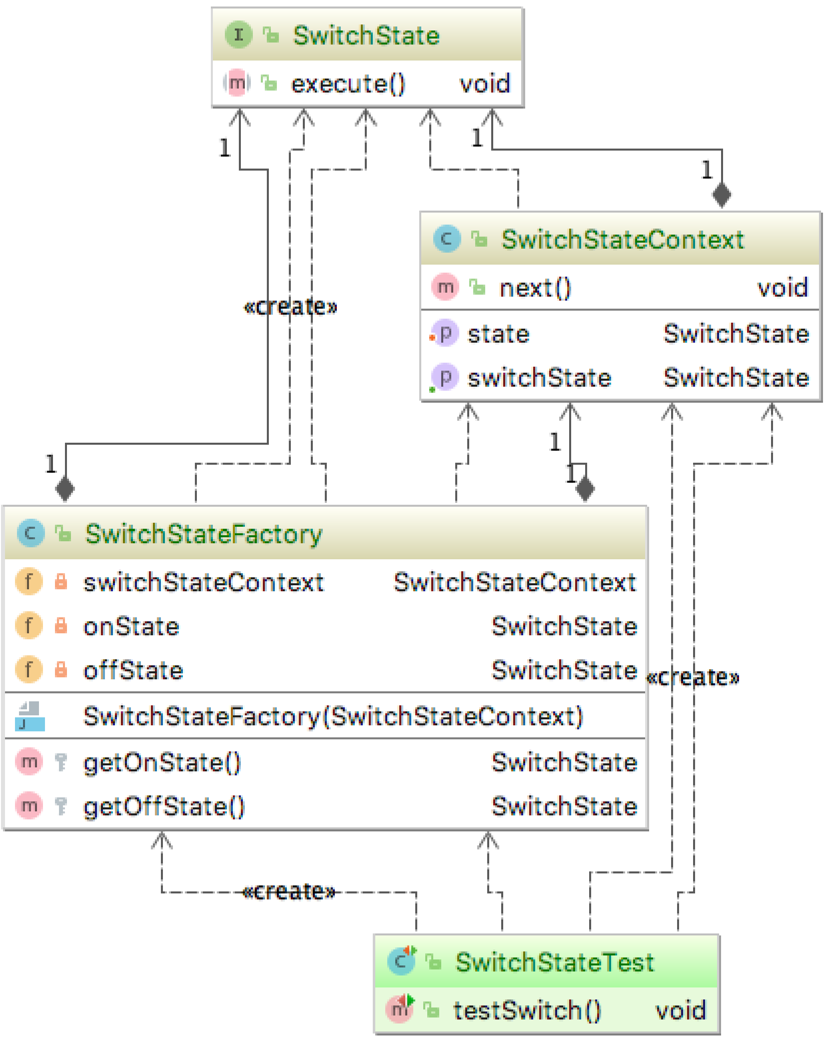
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

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



State: Defining `State`

```java
public interface SwitchState {
    void execute();
}
```

State: Create an `onState` with a `StateFactory`

The following is setting up an on state in a factory

```java
public class SwitchStateFactory {
    private SwitchStateContext switchStateContext;
    private SwitchState onState;

    ...

    public SwitchStateFactory(SwitchStateContext switchStateContext) {
        this.switchStateContext = switchStateContext;
    }

    protected SwitchState getOnState() {
        if (this.onState == null) {
            this.onState = new SwitchState() {
                @Override
                public void execute() {
                    switchStateContext.setState(getOffState());
                }

                @Override
                public String toString() {
                    return "on";
                }
            };
        }
        return onState;
    }
}
```

State: Create an `offState` with a `StateFactory`

The following is setting up an off state in the `StateFactory`

```java
public class SwitchStateFactory {
    private SwitchStateContext switchStateContext;
    private SwitchState offState;

    ...

    public SwitchStateFactory(SwitchStateContext switchStateContext) {
        this.switchStateContext = switchStateContext;
    }

    protected SwitchState getOffState() {
        if (this.offState == null) {
            this.offState = new SwitchState() {
                @Override
                public void execute() {
                    switchStateContext.setState(getOnState());
                }

                @Override
                public String toString() {
                    return "off";
                }
            };
        }
        return offState;
    }
}
```

State: Context

```java
public class SwitchStateContext {
    private SwitchState switchState;

    public void setState(SwitchState switchState) {
        this.switchState = switchState;
    }

    public SwitchState getSwitchState() {
        return switchState;
    }

    public void next() {
        switchState.execute();
    }
}
```

Strategy Pattern

Strategy Pattern Properties

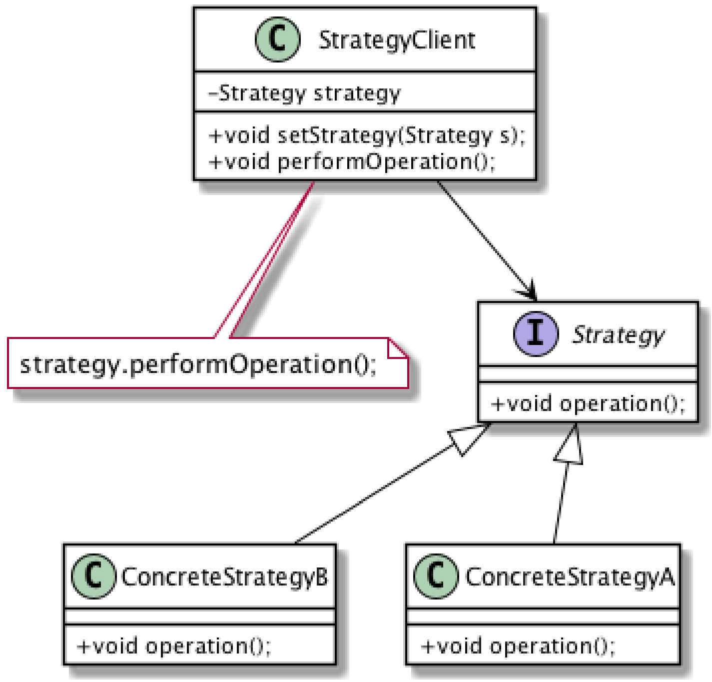
Type: Behavioral
Level: Component

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.

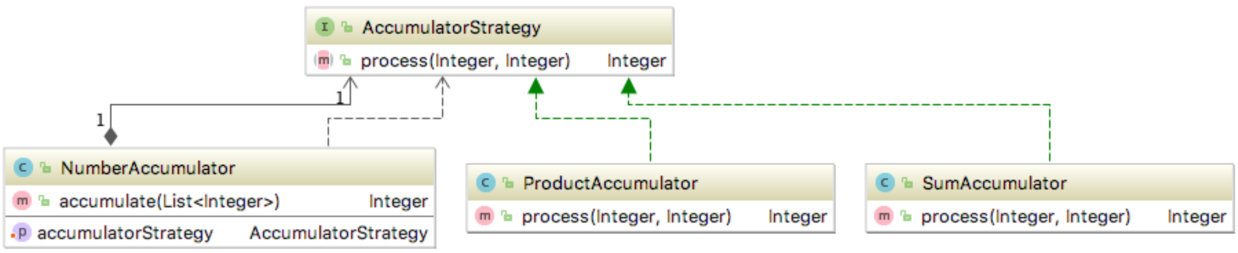
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

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



Strategy: Strategy Interface

```
public interface AccumulatorStrategy {
    public Integer process(Integer a, Integer b);
}
```

Strategy: One Concrete Strategy

```
public class NumberAccumulator {
    private AccumulatorStrategy strategy;

    public void setAccumulatorStrategy(AccumulatorStrategy strategy) {
        this.strategy = strategy;
    }

    public Integer accumulate(List<Integer> integers) {
        if (integers.size() == 1) return integers.get(0);
        return strategy.process(integers.get(0),
            accumulate(integers.subList(1, integers.size())));
    }
}
```

Strategy: Another Concrete Strategy

```
public class ProductAccumulator implements AccumulatorStrategy {
    public Integer process(Integer a, Integer b) {
        return a * b;
    }
}
```

Strategy: Yet Another Concrete Strategy

```
public class SumAccumulator implements AccumulatorStrategy {
    public Integer process(Integer a, Integer b) {
        return a + b;
    }
}
```

Strategy: Use of the Strategy Pattern

```
NumberAccumulator numberAccumulator = new NumberAccumulator();
numberAccumulator.setAccumulatorStrategy(new ProductAccumulator());

List<Integer> integers = new ArrayList<Integer>();
integers.add(1);
integers.add(2);
integers.add(3);
integers.add(4);
```

Strategy: Use of the Strategy Pattern using Lambdas (Java 8+)

The following is the same as the above, but using lambdas and functional programming, much of the boilerplate is evaporated away

Using lambdas as a Strategy Pattern

```
List.of(1,2,3,4).stream().reduce((total, next) -> total + next);
```

Renders:

```
Optional[10]
```

Lab: Using the `Collections.sort`

Step 1: In design_patterns_training, go to the *src/test/java* folder and a package called `com.xyzcorp.strategy`, if they don't exist, create it.

Step 2: In the `com.xyzcorp.strategy` package, create a test class called `StrategyTest`

Step 3: In `StrategyTest`, create a class called `MusicArtist` that is plain old java object, `MusicArtist` should have a `firstName` of type `String`, a `lastName` of type `String`, an `alias` of type `String`. Be sure to include an `equals`, `toString`, `hashCode`

Step 4: In `StrategyTest`, create a test method called `testSortLastNameStrategy`. In the test, create a `List<Artist>` of the following artists (add your own favorites):

Billy Idol
Louis Armstrong
Beyoncé Knowles (alias: Beyoncé)
Paul Hewson (alias: Bono)
Prince Nelson (alias: Prince)
Muddy Waters
George Harrison
Lata Mangeshkar
Zhang Liangying

Lab: Using the `Collections.sort`

Step 5: In `testSortLastNameStrategy`, use `Collections.sort(list, comparator)` to sort the list you created and provide a `Comparator<Artist>` that would sort the list by `lastName`

Step 6: If time is available, create another test called `testSortFirstNameStrategy` that will sort the `List<Artist>` by `firstName`. You will have to extract the list of artists as a property of the `StrategyTest` class or make a copy from `testSortLastNameStrategy` to `testSortFirstNameStrategy`.

Chain of Responsibility Pattern

Chain of Responsibility Properties

Type: Behavioral
Level: Component

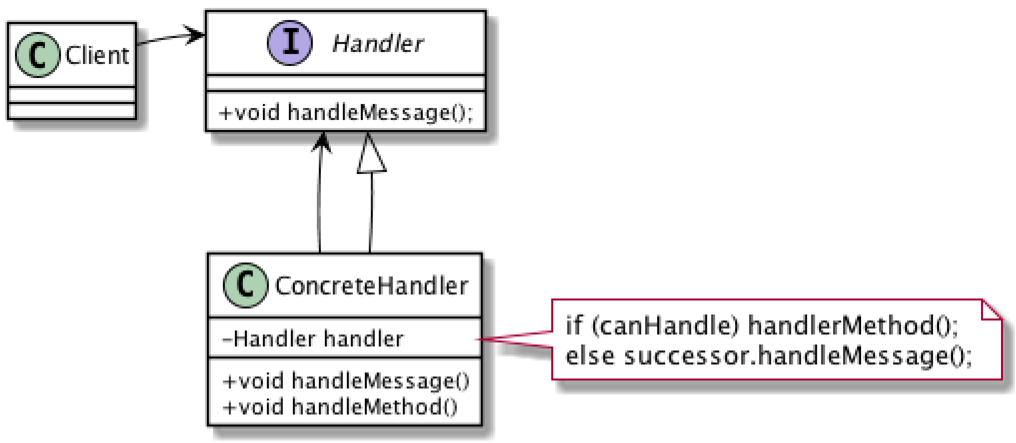
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

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

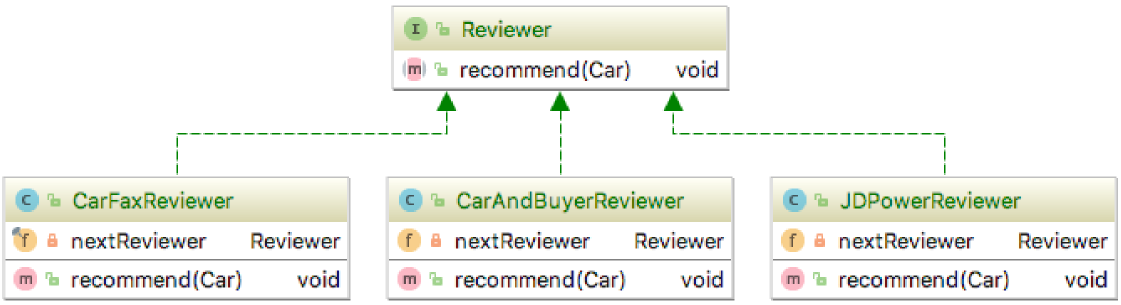
Chain of Responsibility Advantages

- 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.

Chain of Responsibility Disadvantages

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

Chain of Responsibility Demo Diagram



Chain of Responsibility: Interface of a Model

```java
import java.util.List;

public interface Car {
    String getMake();
    String getModel();
    int getYear();

    boolean powerSteering();
    boolean driverAirBags();
    boolean passengerAirBags();
    boolean seatHeaters();
    boolean seatCoolers();
    boolean driveLaneAssist();
    boolean rearCamera();
    void addRecommendation(String name);
    List<String> getRecommendations();
}
```

Chain of Responsibility: Handler

```java
public interface Reviewer {
    void recommend(Car car);
}
```

Chain of Responsibility: Concrete Handler

```
public class CarAndBuyerReviewer implements Reviewer {

    private Reviewer nextReviewer;

    public CarAndBuyerReviewer(Reviewer nextReviewer) {
        this.nextReviewer = nextReviewer;
    }

    public CarAndBuyerReviewer() {
        this.nextReviewer = null;
    }

    @Override
    public void recommend(Car car) {
        if (car.passengerAirBags() && car.driverAirBags())
           car.addRecommendation("Car and Buyer");
        if (nextReviewer != null)
           nextReviewer.recommend(car);
    }
}
```

Chain of Responsibility: Another Concrete Handler

```
public class CarFaxReviewer implements Reviewer {

    private final Reviewer nextReviewer;

    public CarFaxReviewer(Reviewer nextReviewer) {
        this.nextReviewer = nextReviewer;
    }

    public CarFaxReviewer() {
        this.nextReviewer = null;
    }

    @Override
    public void recommend(Car car) {
        if (car.driverAirBags())
           car.addRecommendation("CarFax");
        if (nextReviewer != null)
           nextReviewer.recommend(car);
    }
}
```

Chain of Responsibility: Yet Another Concrete Handler

```
public class JDPowerReviewer implements Reviewer {

    private Reviewer nextReviewer;

    public JDPowerReviewer(Reviewer reviewer) {
        this.nextReviewer = reviewer;
    }

    public JDPowerReviewer() {
        this.nextReviewer = null;
    }

    @Override
    public void recommend(Car car) {
        if (car.rearCamera() && car.driveLaneAssist() &&
            car.powerSteering())
            car.addRecommendation("JD Power");
        if (nextReviewer != null) nextReviewer.recommend(car);
    }
}
```

Chain of Responsibility: Running

```
CarFaxReviewer carFaxReviewer = new CarFaxReviewer();
JDPowerReviewer jdPowerReviewer = new JDPowerReviewer(carFaxReviewer);
CarAndBuyerReviewer carAndBuyerReviewer = new CarAndBuyerReviewer
(jdPowerReviewer);

carAndBuyerReviewer.recommend(car);

System.out.println(car.getRecommendations());
```

Command Pattern

Command Pattern Properties

Type: Behavioral
Level: Object

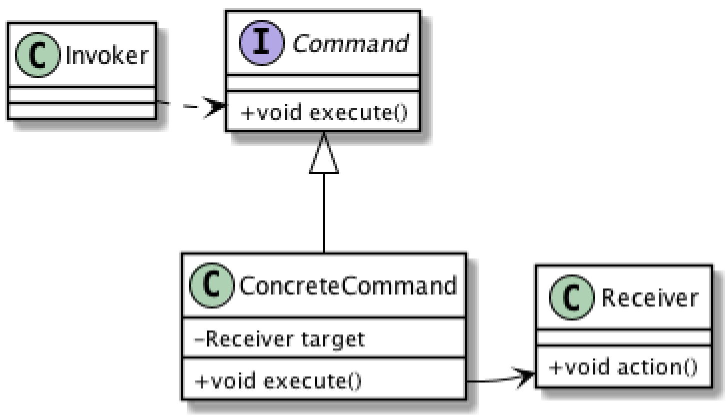
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

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



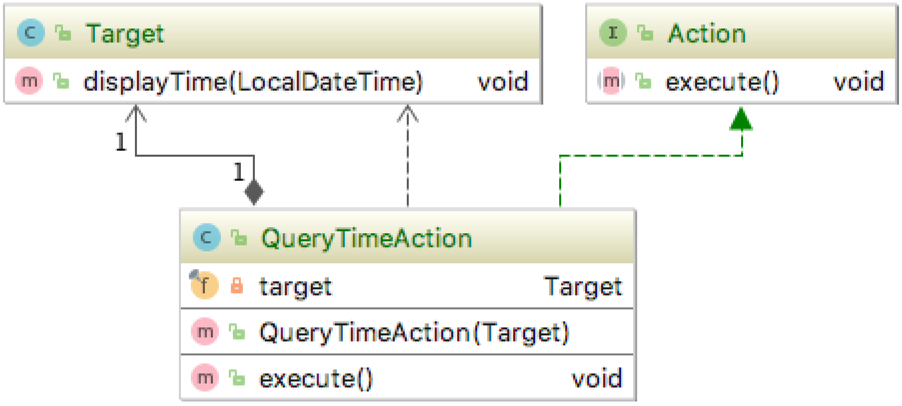
Command Pattern Advantages

- 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



Command: The command interface

```
public interface Action {
    public void execute();
}
```

Command: The command target

```
import javax.swing.<strong>;
import java.awt.</strong>;
import java.time.LocalDateTime;

public class Target {
    public void displayTime(LocalDateTime localDateTime) {
        JFrame jFrame = new JFrame("Title");
        JPanel contentPane = new JPanel(new FlowLayout());
        JLabel jLabel = new JLabel("The time is: " +
           localDateTime.toString());
        contentPane.add(jLabel);
        jFrame.setContentPane(contentPane);
        jFrame.pack();
        jFrame.setVisible(true);
        jFrame.setDefaultCloseOperation(
           WindowConstants.DISPOSE_ON_CLOSE);
    }
}
```

Command: A command action

The nice thing about the command pattern is that it is easy to add runtime actions!

```java
import java.time.LocalDateTime;

public class QueryTimeAction implements Action{

    private final Target target;

    public QueryTimeAction(Target target) {
        this.target = target;
    }

    @Override
    public void execute() {
        target.displayTime(LocalDateTime.now());
    }
}
```

Command: Bringing it together

Using a `main` method or a dependency injection container:

```java
Map<String, Action> commandMap = new HashMap<>();
Target target = new Target();
Action action = new QueryTimeAction(target);
commandMap.put("showTime", action);
commandMap.get("showTime").execute();
```

Command: Bringing it together with Java 8 lambdas

Using a `main` method or a dependency injection container:

```java
Map<String, Action> commandMap = new HashMap<>();
Target target = new Target();
Action action = () -> {
   target.displayTime(LocalDateTime.now());
};
commandMap.put("showTime", action);
commandMap.get("showTime").execute();
```

Iterator Pattern

Iterator Pattern Properties

Type: Behavioral, Object
Level: Component

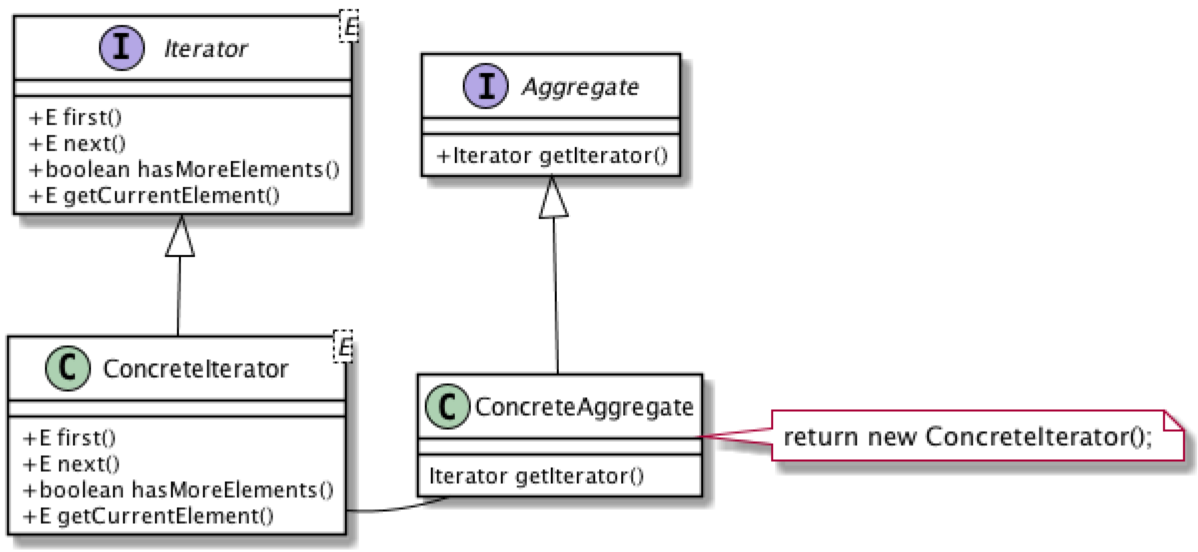
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

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

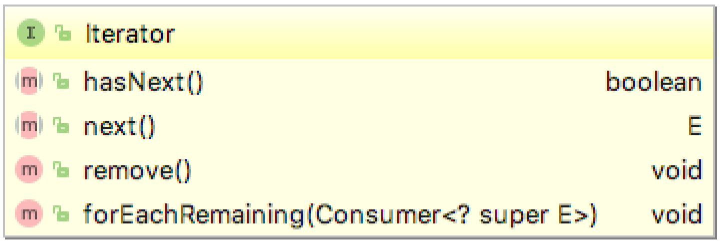
- A uniform interface for traversing a collection
- Not to tied to the implementation of the collection
- Enabling several clients to simultaneously navigate within the same underlying collection.
- You can think of an Iterator as a cursor or pointer into the collection

Iterator Disadvantages

- They give the illusion of order to unordered structures

Iterator Demo Diagram

JDK 8+ UML Diagram



Iterator: Java’s implementation (Java 8+)

```
public interface Iterator<E> {
    boolean hasNext();

    E next();

    default void remove() {
        throw new UnsupportedOperationException("remove");
    }

    default void forEachRemaining(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        while (hasNext())
            action.accept(next());
    }
}
```

The above is the Java 8+ signature

Use of Java's Iterator

```
var stringList = List.of("Foo", "Bar", "Baz", "Qux", "Quux");
Iterator<String> iterator = stringList.iterator();

String value1 = iterator.next();
String value2 = iterator.next();

assertEquals(value1, "Foo"); //true
assertEquals(value2, "Bar"); //true
```

Java Iterator Trick Question

What is `value1` and `value2`?

```
var stringList = List.of("Foo", "Bar", "Baz", "Qux", "Quux");
String value1 = stringList.iterator().next();
String value2 = stringList.iterator().next();
```

Using Java Iterator with `while`

```
var iterator = List.of("Foo", "Bar", "Baz", "Qux", "Quux").iterator();
var result = new ArrayList<String>();
while(iterator.hasNext()) {
    result.add(iterator.next());
}
assertEquals("[Foo, Bar, Baz, Qux, Quux]",result.toString());
```

Using Java Iterator with Java 5 for loop

If a collection extends from `Iterable` it can be placed in a loop

```
var list = List.of("Foo", "Bar", "Baz", "Qux", "Quux");
var result = new ArrayList<String>();
for (String s : list) {
    result.add(s);
}
assertEquals("[Foo, Bar, Baz, Qux, Quux]",result.toString());
```

Using Java's `ListIterator`

- An iterator for lists that traverses the list in either direction
- Modify the list during iteration
- Obtain the iterator's current position in the list

```
var list = List.of("Foo", "Bar", "Baz", "Qux", "Quux");
var listIterator = list.listIterator();
listIterator.next();
listIterator.next();
listIterator.next();
listIterator.previous();
listIterator.previous();
listIterator.next();
assertEquals(listIterator.next(), "Baz");
```

Using Java's `Spliterator`

- Iterator that "splits" perhaps for parallel purposes
- Traverse elements individually using `tryAdvance`
- Traverse elements sequentially in bulk using `forEachRemaining`
- `Spliterator` can be queried with characteristics about the underlying collection

Using `forEachRemaining` with a `Spliterator`

`forEachRemaining` iterates all elements using given `Consumer`

```
var list = List.of("Foo", "Bar", "Baz", "Qux", "Quux");
var split1 = list.spliterator();
var split2 = split1.trySplit();
split1.forEachRemaining(x -> System.out.println("S1 " + x));
split2.forEachRemaining(x -> System.out.println("S2 " + x));
```

Will result in:

```
S1 Baz
S1 Qux
S1 Quux
S2 Foo
S2 Bar
```

Using `tryAdvance` with a `Spliterator`

```
var list = List.of("Foo", "Bar", "Baz", "Qux", "Quux");
var split1 = list.spliterator();
var split2 = split1.trySplit();
split1.tryAdvance(x -> System.out.println("S1 " + x));
split2.tryAdvance(x -> System.out.println("S2 " + x));
```

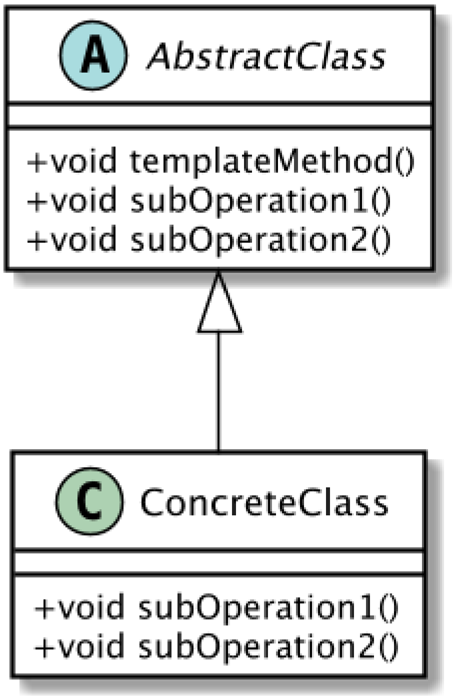
Will result in:

```
S1 Baz
S2 Foo
```

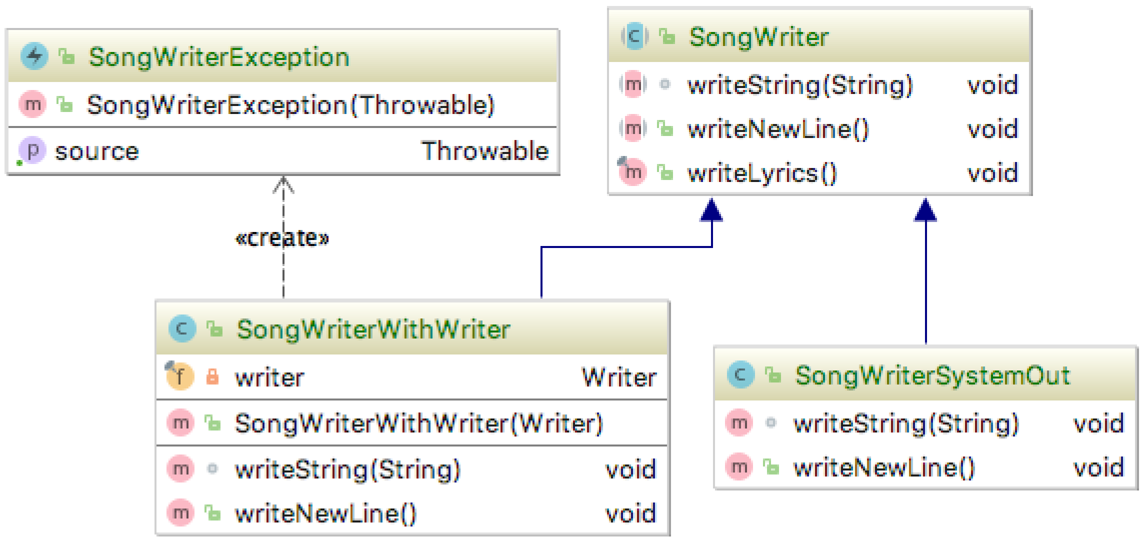
Template Method Pattern

Template Method Properties

Template Method Canonical Diagram



Template Method Diagram



Template Method 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.

Template Method: The `abstract` class

```java
public abstract class SongWriter {
    abstract void writeString(String str) throws SongWriterException;
    public abstract void writeNewLine() throws SongWriterException;

    public final void writeLyrics() throws SongWriterException {
        writeString("I see trees of green");
        writeNewLine();
        writeString("red roses too, I see them bloom");
        writeNewLine();
        writeString("for me and you");
        writeNewLine();
        writeString("and I think to myself");
        writeNewLine();
        writeString("what a wonderful world");
        writeNewLine();
    }
}
```

Template Method: One possible implementation

```java
import java.io.IOException;
import java.io.Writer;

public class SongWriterWithWriter extends SongWriter {
    private final Writer writer;

    public SongWriterWithWriter(Writer writer) {
        this.writer = writer;
    }

    @Override
    void writeString(String str) throws SongWriterException {
        try {
            writer.write(str);
        } catch (IOException e) {
            throw new SongWriterException(e);
        }
    }

    @Override
    public void writeNewLine() throws SongWriterException {
        try {
            writer.write("\n");
        } catch (IOException e) {
            throw new SongWriterException(e);
        }
    }
}
```

Template Method: Another possible implementation

```java
public class SongWriterSystemOut extends SongWriter {
    @Override
    void writeString(String str) throws SongWriterException {
        System.out.println(str);
    }

    @Override
    public void writeNewLine() throws SongWriterException {
        System.out.println();
    }
}
```

Template Method can be done with `default` methods

```
public interface LineItem {
    public float getCost();
    public float getQuantity();
    public float getDiscount();
    public default float getSubtotal() {
        return (getCost() * getQuantity()) * getDiscount();
    }
}
```

Lab: The 'combine' template method in the `Function` interface

Step 1: In design_patterns_training, go to the *src/test/java* folder exists and a package called `com.xyzcorp.templatemethod`, if they don't exist, create them.

Step 2: In the `com.xyzcorp.templatemethod` package, create a test class called `TemplateMethodTest`

Step 3: In the `TemplateMethodTest`, create a test called `testFunctionCompose`, and in the test create two functions

1. A `Function` that takes an `Object` and returns the result of `toString`
2. A `Function` that takes an `String` and returns the size of the `String`

Step 4: Using the API of `Function` as a guide, combine these two functions using `combine` which will be a `Function` that takes an `Object` and returns `String`

Step 5: Test your results

Lab: The `andThen` template method in the `Function` interface

Step 6: In the `TemplateMethodTest`, create a test called `testFunctionAndThen`, and in the test, create or copy the two functions from the previous test:

1. A `Function` that takes an `Object` and returns the result of `toString`
2. A `Function` that takes an `String` and returns the size of the `String`

Step 7: Using the API of `Function` as a guide, combine these two functions using `andThen` which will be a `Function` that takes an `Object` and returns `String`

Step 8: Test your results

Mediator Pattern

Mediator Pattern Properties

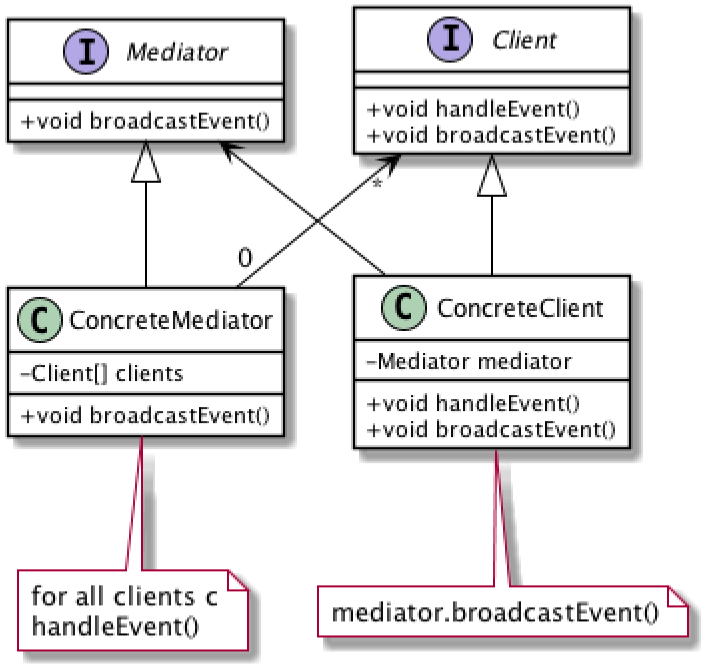
Type: Behavioral
Level: Component

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

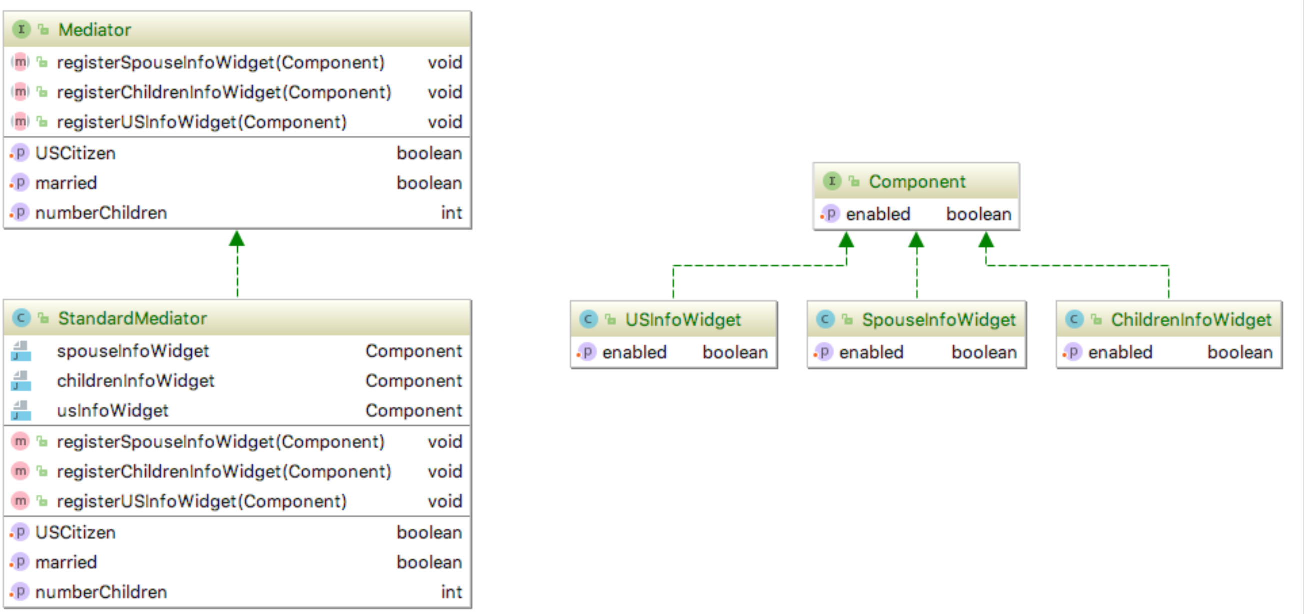
Mediator Advantages

- 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

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

Mediator Demo Diagram



Mediator: The Mediator Interface

```
public interface Mediator {
    void setUSCitizen(boolean isUSCitizen);
    void setNumberChildren(int children);
    void setMarried(boolean isMarried);
    void registerSpouseInfoWidget(Component component);
    void registerChildrenInfoWidget(Component component);
    void registerUSInfoWidget(Component component);
}
```

Mediator: The Mediator Implementation Registration

```
public class StandardMediator implements Mediator {
    private Component spouseInfoWidget;
    private Component childrenInfoWidget;
    private Component usInfoWidget;

    @Override
    public void registerSpouseInfoWidget(Component component) {
        this.spouseInfoWidget = component;
    }

    @Override
    public void registerChildrenInfoWidget(Component component) {
        this.childrenInfoWidget = component;
    }

    @Override
    public void registerUSInfoWidget(Component component) {
        this.usInfoWidget = component;
    }
}
```

Mediator: The Mediator Implementation Activation

```java
public class StandardMediator implements Mediator {
    private Component spouseInfoWidget;
    private Component childrenInfoWidget;
    private Component usInfoWidget;

    //Switches

    @Override
    public void setUSCitizen(boolean isUSCitizen) {
        if (usInfoWidget != null) {
            if (isUSCitizen) {
                usInfoWidget.setEnabled(true);
            } else {
                usInfoWidget.setEnabled(false);
            }
        }
    }

    @Override
    public void setNumberChildren(int children) {
        if (childrenInfoWidget != null) {
            if (children > 0) {
                childrenInfoWidget.setEnabled(true);
            } else {
                childrenInfoWidget.setEnabled(false);
            }
        }
    }

    @Override
    public void setMarried(boolean isMarried) {
        if (spouseInfoWidget != null) {
            if (isMarried) {
                spouseInfoWidget.setEnabled(true);
            } else {
                spouseInfoWidget.setEnabled(false);
            }
        }
    }
}
```

Memento Pattern

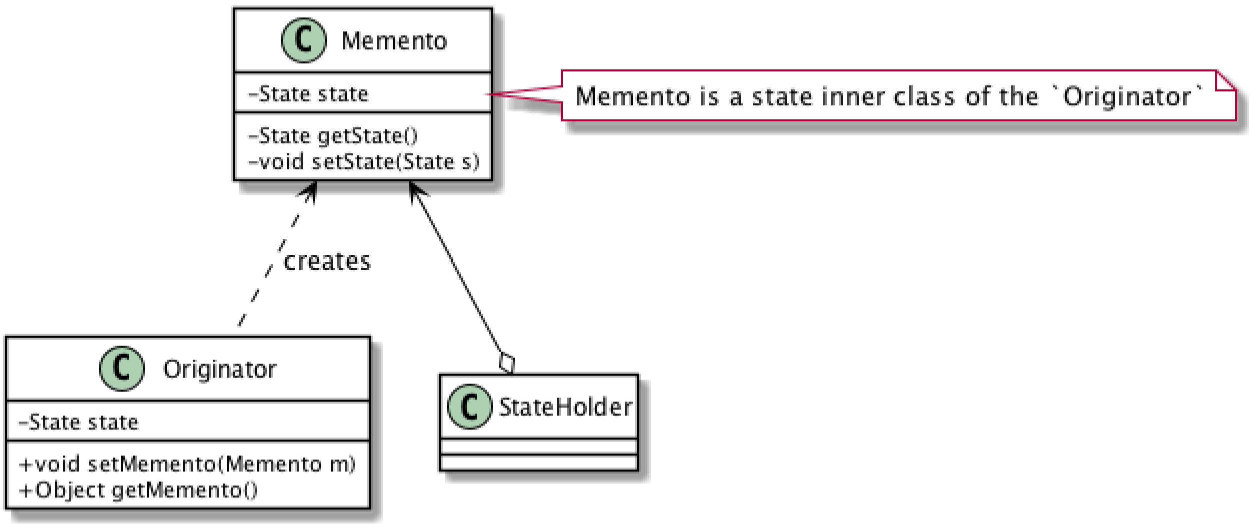
Memento Pattern Properties

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



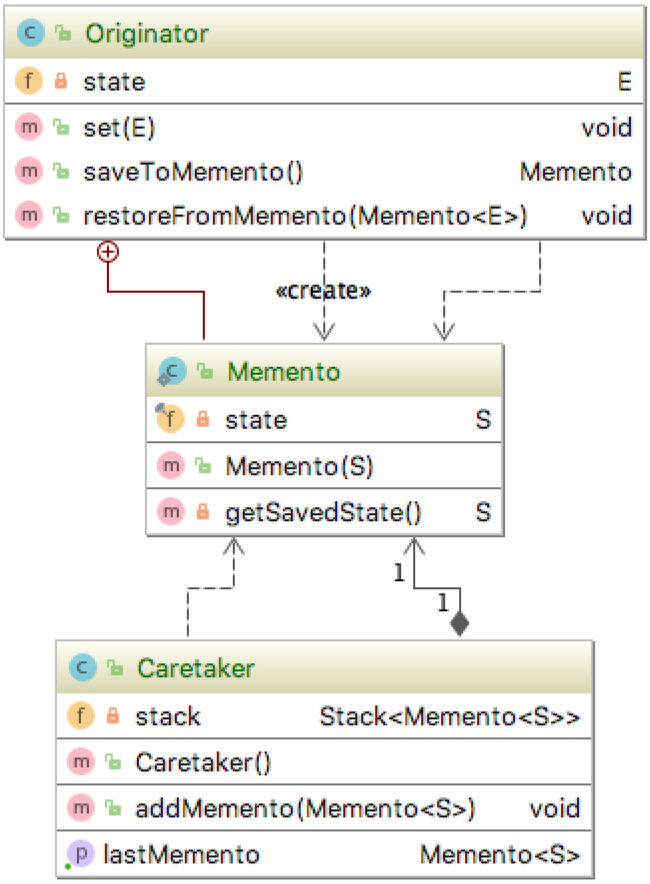
Memento Advantages

- 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

Memento Disadvantages

- Expensive Storage Overtime
- Requires thought with large graphs

Memento Demo Diagram



Memento: Originator

```java
public class Originator<E> {
    private E state;
    // The class could also contain
    // additional data that is not part of the
    // state saved in the memento..

    public void set(E state) {
        this.state = state;
        System.out.println("Originator: Setting" +
                           " state to " + state);
    }

    public Memento saveToMemento() {
        System.out.println("Originator: Saving to Memento.");
        return new Memento<>(this.state);
    }

    public void restoreFromMemento(Memento<E> memento) {
        this.state = memento.getSavedState();
        System.out.println("Originator: State after" +
                           "restoring from Memento: " +
                           state);
    }

    //Memento Declaration...
}
```

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

Memento: Memento inside the Originator

```java
public class Originator<E> {

    //Memento declaration and methods in previous slide

    public static class Memento<S> {
        private final S state;

        public Memento(S stateToSave) {
            state = stateToSave;
        }

        // accessible by outer class only
        private S getSavedState() {
            return state;
        }
    }
}
```

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

Memento: StateHolder

- `Caretaker` is a state holder will hold the different `Memento`
- You can use whatever collection to recall the previous state

```java
import java.util.Stack;

public class Caretaker<S> {
    private Stack<Originator.Memento<S>> stack;

    public Caretaker() {
        this.stack = new Stack<>();
    }

    public void addMemento(Originator.Memento<S> memento) {
        this.stack.push(memento);
    }

    public Originator.Memento<S> getLastMemento() {
        return this.stack.pop();
    }
}
```

Observer Pattern

Observer Pattern Properties

Type: Behavioral
Level: Component

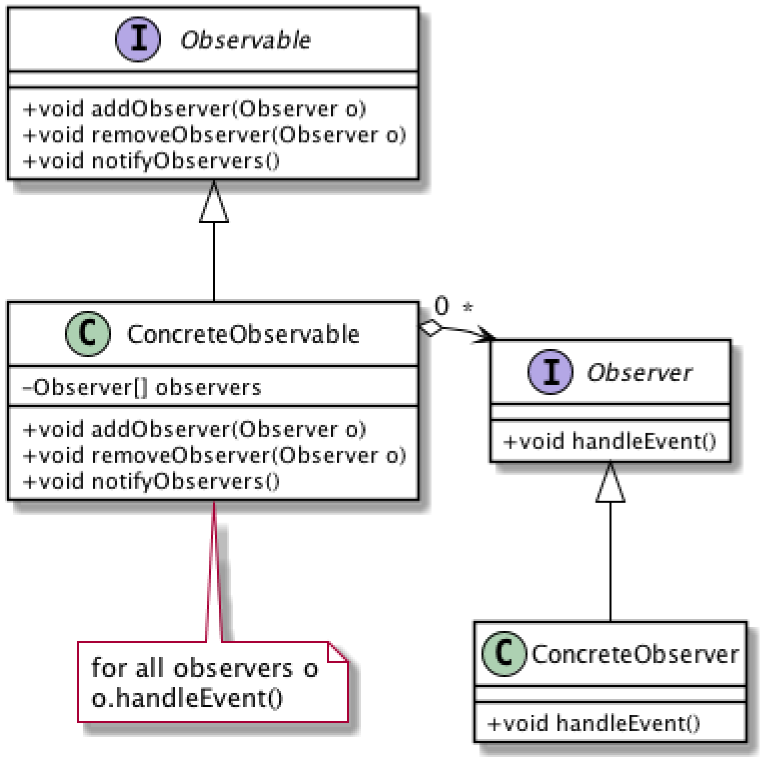
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Observer Pattern Purpose

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

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

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.

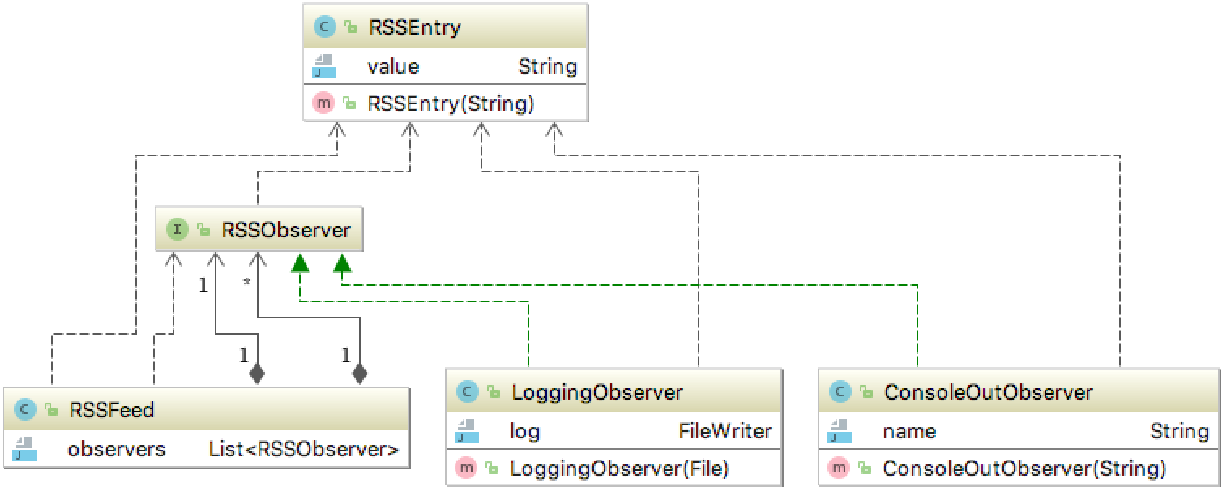
The notification methods copy (or clone) the `Observer` list and iterate through the list, and call the specific listener methods on each `Observer`.

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: Applied Java Patterns By Stephen Stelting, Olav Massen

Observer Demo Diagram



Observer: Object

```
public class RSSEntry {

    private String value;

    public RSSEntry(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }
}
```

Observer: Concrete Observable

```
public class RSSFeed {

    private List<RSSObserver> observers = new ArrayList<RSSObserver>();

    public void broadcast(RSSEntry entry) {
        for (RSSObserver observer : observers) {
            observer.update(entry);
        }
    }

    public RSSObserver addObserver(RSSObserver observer) {
        observers.add(observer);
        return observer;
    }

    public void removeObserver(RSSObserver observer) {
        observers.remove(observer);
    }
}
```

Observer: Observer

```
public interface RSSObserver {

    void update(RSSEntry entry);
}
```

Observer: Concrete Observer

```java
public class LoggingObserver implements RSSObserver {

    private FileWriter log;

    public LoggingObserver(File log) {
        try {
            this.log = new FileWriter(log);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void update(RSSEntry entry) {

        try {
            log.write(entry.getValue());
            log.write('\n');
            log.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Observer: Another Concrete Observer

```java
public class ConsoleOutObserver implements RSSObserver {

    private String name;

    public ConsoleOutObserver(String name) {
        this.name = name;
    }

    public void update(RSSEntry entry) {
        System.out.println(name + " : " + entry.getValue());
    }
}
```

Lab: Using Guava's Observer

Guava has an interesting flavor of the Observer pattern called `EventBus` https://github.com/google/guava/wiki/EventBusExplained

Step 1: In design_patterns_training, go to the *src/test/java* folder and a package called `com.xyzcorp.observer`, if it does not exist create it.

Step 2: In the `com.xyzcorp.observer` package, create a test called `BroadcastTest`

Step 3: Create a class called `BroadcastEvent` inside `BroadcastTest`, this will be a plain old java object. `BroadcastEvent` should have a single property called `message` and should be a type of `String`. Be sure to include `equals`, `hashCode`, `toString`

Step 4: Create a class called `Broadcaster` inside of `BroadcastTest` that has a single property called `eventBus` of type `com.google.common.eventbus.EventBus` be sure that it is assignable via a constructor or a setter. Create a method called `broadcastToAll` that returns `void`. Inside of the `broadcastToAll`, use the `eventBus` to `post` a new `BroadcastEvent` with whatever message you would like

Step 5: Create a class called `Subscriber` inside of `BroadcastTest` that contains a property of type `List<String>` called `messages`

Step 6: In `Subscriber`, create a method called `eventOccured` that includes the annotation `com.google.common.eventbus.Subscribe` and the following signature:

```
@Subscribe
public void eventOccured(BroadcastEvent event) {
    messages.add(event.getMessage());
}
```

Step 7: In `Subscriber`, create a method called `getCount` that will return the number of message it received. Next, create a method called `getMessages` that will return the `List<String> messages` property or a copy of the property

Lab: Using Guava's Observer

Step 8: In the `BroadcastTest` you created, create a test method called `testBasicUse()` with the following:

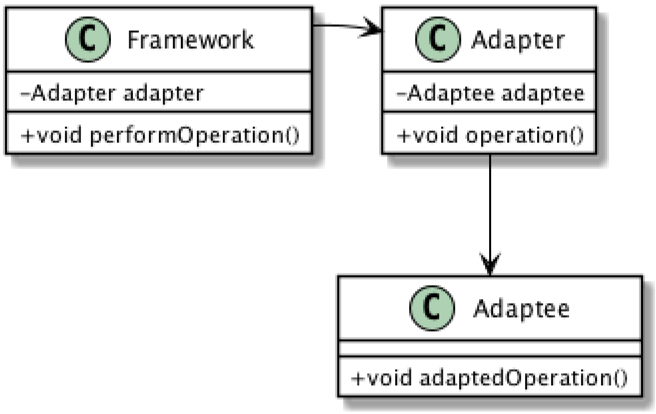
```java
EventBus eventBus = new EventBus();
Subscriber subscriber = new Subscriber();
eventBus.register(subscriber);

Broadcaster broadcaster = new Broadcaster();
broadcaster.setEventBus(eventBus);

broadcaster.broadcastToAll();
broadcaster.broadcastToAll();
broadcaster.broadcastToAll();
```

Step 9: In the `testBasicUse()` method, use `subscriber's `count` and `getMessages` to ensure that the messages were recieved, and run the tests.

Structural Patterns

Adapter Pattern

Adapter Pattern Canonical Diagram



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

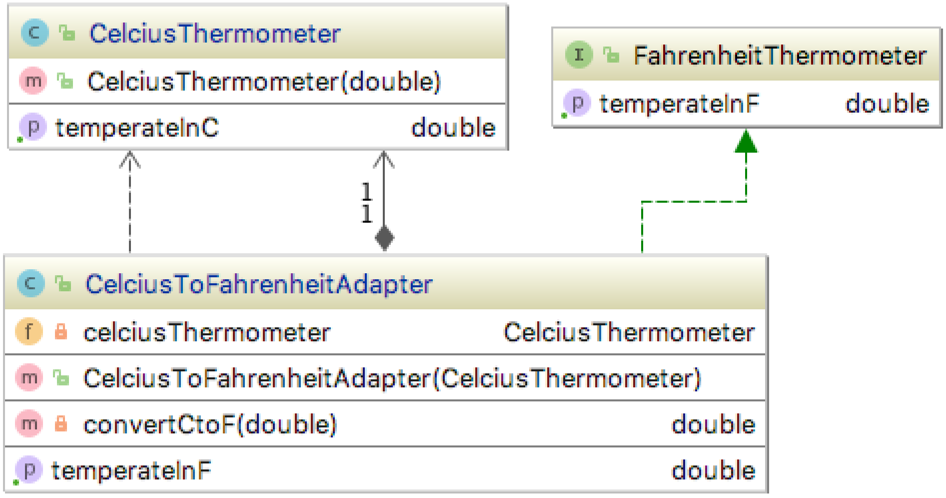
Adapter Pattern Advantages

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

Adapter Pattern Disadvantages

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

Adapter Pattern Demo Diagram



Adapter Pattern: The Target (Adaptee)

```
public class CelciusThermometer {

    private double temp;

    public CelciusThermometer(double temp) {
        this.temp = temp;
    }

    public double getTemperateInC() {
        return temp;
    }
}
```

Adapter Pattern: The Adapter

```
public class CelciusToFahrenheitAdapter
   implements FahrenheitThermometer {

    private CelciusThermometer celciusThermometer;

    public CelciusToFahrenheitAdapter
              (CelciusThermometer celciusThermometer) {
        this.celciusThermometer = celciusThermometer;
    }

    public double getTemperateInF() {
        return convertCtoF(celciusThermometer.getTemperateInC());
    }

    private double convertCtoF(double c) {
        return (c * 9 / 5) + 32;
    }

}
```

Adapter Pattern: The Adapter's Interface

```
public interface FahrenheitThermometer {
    double getTemperateInF();
}
```

Lab: Making an Adapter

Step 1: In design_patterns_training, go to *src/test/java* and a package called `com.xyzcorp.adapter`, if they do not exist, create them.

Step 2: In the `com.xyzcorp.adapter` package, create a test class called `AdapterTest`

Step 3: There is no `isOdd` nor `isEven` in the `java.lang.Integer` class, but we don't have access to the source. In the `AdapterTest`, create an adapter with an `isOdd` method that returns a `boolean` if the target is odd, call it `OddEvenAdapter`. Create an `isEven` method that returns `boolean` if the target is even.

Step 4: Test the results in a test method called `testAdapter`

Bridge Pattern

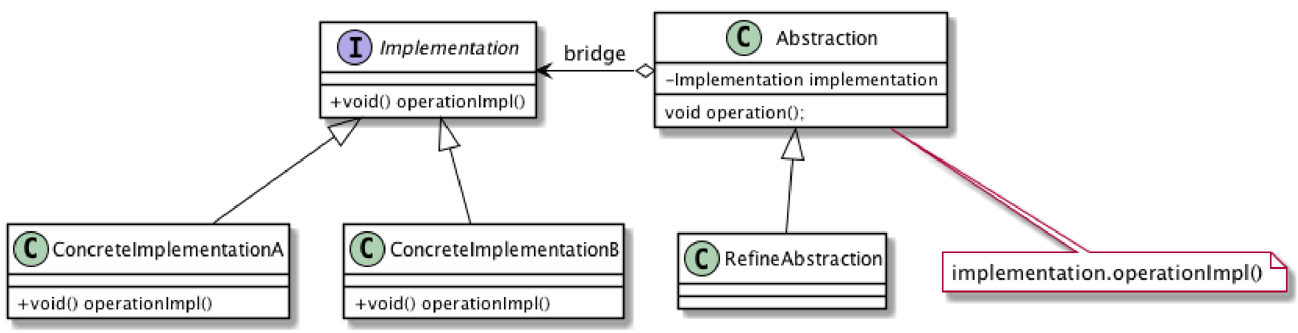
Bridge Pattern Properties

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

Bridge Pattern Canonical Diagram



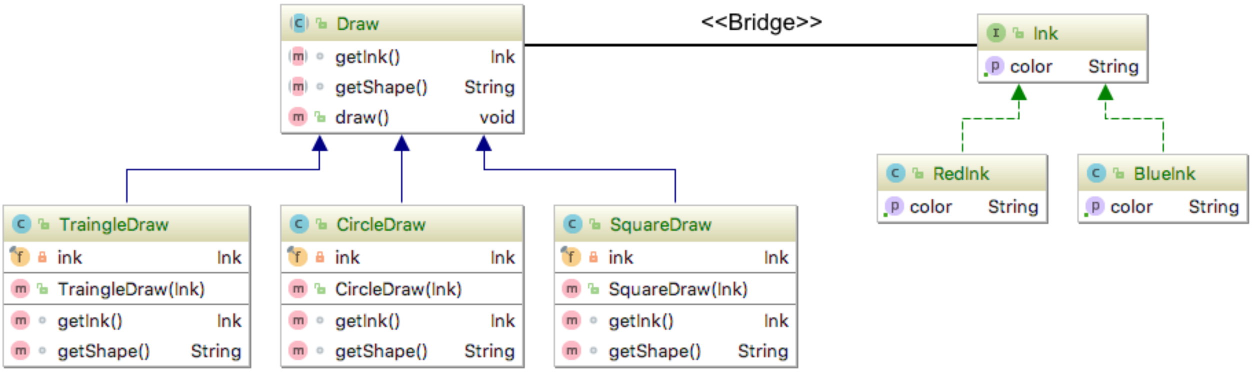
Bridge Pattern Advantages

- 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.

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

Bridge Demo Diagram



Bridge: Abstraction

```
public abstract class Draw {

    abstract Ink getInk();
    abstract String getShape();

    public void draw() {
        System.out.println("Drawing a " + getShape() + " with " +
getInk().getColor() + " ink");
    }
}
```

Bridge: Refine Abstraction

```
public class SquareDraw extends Draw {

    private final Ink ink;

    public SquareDraw(Ink ink) {
        this.ink = ink;
    }

    @Override
    Ink getInk() {
        return ink;
    }

    @Override
    String getShape() {
        return "square";
    }
}
```

Bridge: Implementation

```java
public interface Ink {
    String getColor();
}
```

Bridge: Concrete Implementation

```java
public class RedInk implements Ink {
    @Override
    public String getColor() {
        return "red";
    }
}
```

Bridge: Another Concrete Implementation

```java
public class BlueInk implements Ink {
    @Override
    public String getColor() {
        return "blue";
    }
}
```

Composite Pattern

Composite Properties

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

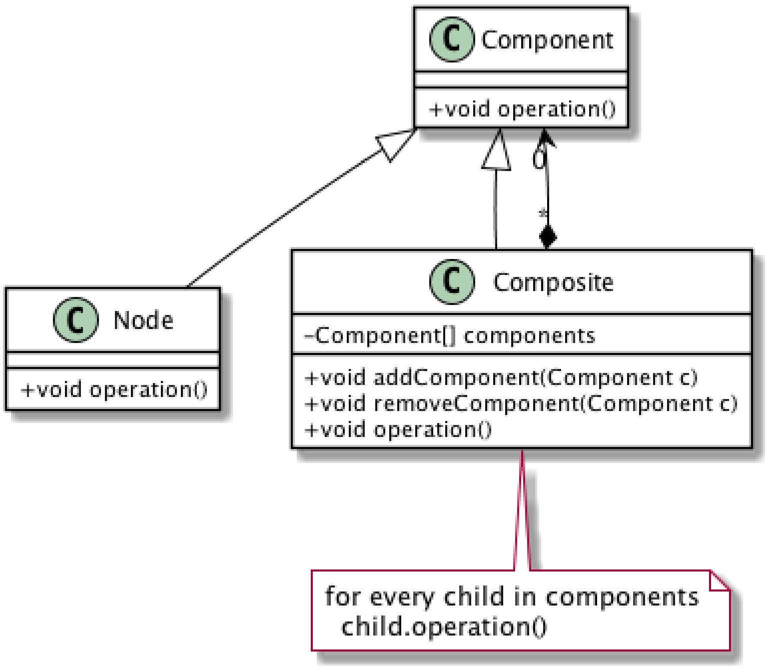
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

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.

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

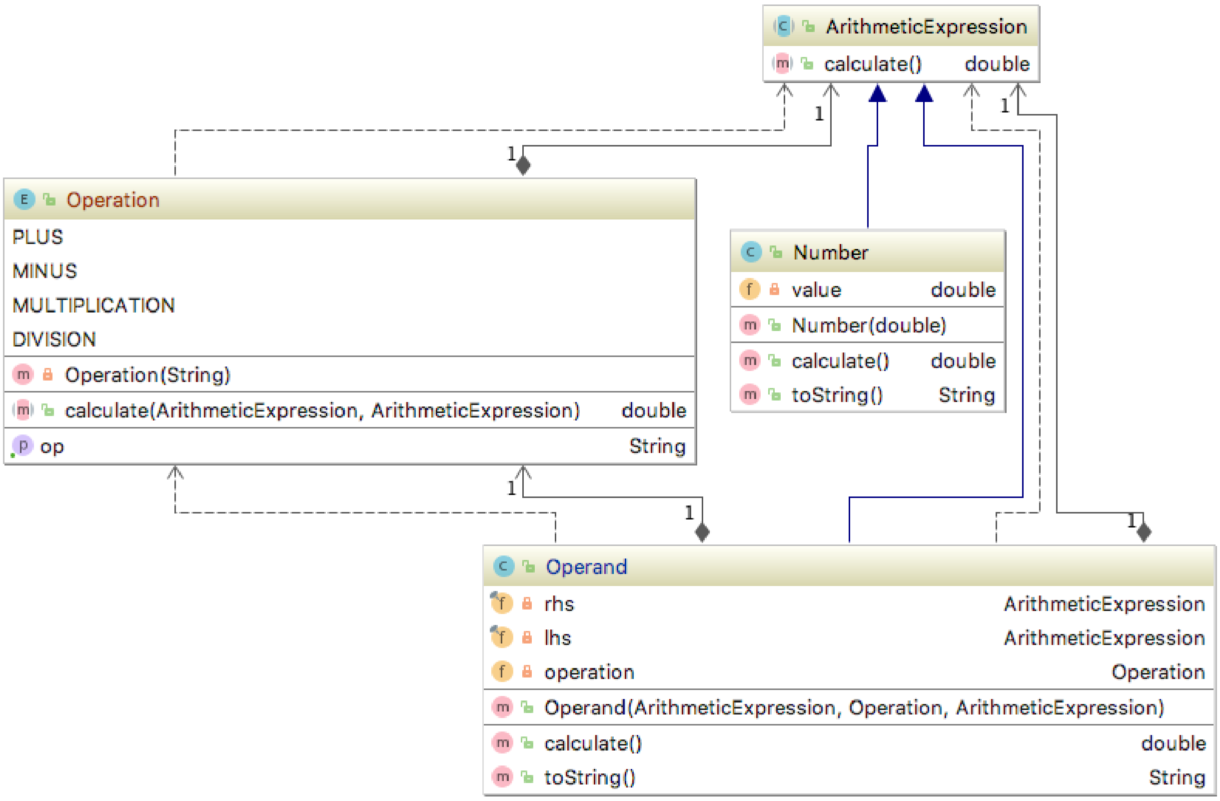
Composite Advantages

- 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 Demo Diagram



Composite: Component

```java
public abstract class ArithmeticExpression {
    public abstract double calculate();
}
```

Composite: Leaf

```
public class Number extends ArithmeticExpression {

    private double value;

    public Number(double value) {
        this.value = value;
    }

    @Override
    public double calculate() {
        return value;
    }

    @Override
    public String toString() {
        return Double.toString(value);
    }
}
```

Composite: Composite

```
public class Operand extends ArithmeticExpression {
    private final ArithmeticExpression rhs;
    private final ArithmeticExpression lhs;
    private Operation operation;

    public Operand(ArithmeticExpression rhs,
                   Operation operation,
                   ArithmeticExpression lhs) {
        this.rhs = rhs;
        this.lhs = lhs;
        this.operation = operation;
    }

    @Override
    public double calculate() {
        return operation.calculate(rhs, lhs);
    }

    @Override
    public String toString() {
        return "( " + rhs.toString() + " " +
                      operation.getOp() + " " +
                      lhs.toString() + " )";
    }
}
```

Composite: Utility Class

```
public enum Operation {
    PLUS("+") {
        @Override
        public double calculate(ArithmeticExpression rhs,
                                ArithmeticExpression lhs) {
            return rhs.calculate() + lhs.calculate();
        }
    },
    MINUS("-") {
        @Override
        public double calculate(ArithmeticExpression rhs,
                                ArithmeticExpression lhs) {
            return rhs.calculate() - lhs.calculate();
        }
    },
    MULTIPLICATION("*") {
        @Override
        public double calculate(ArithmeticExpression rhs,
                                ArithmeticExpression lhs) {
            return rhs.calculate() * lhs.calculate();
        }
    },
    DIVISION("/") {
        @Override
        public double calculate(ArithmeticExpression rhs,
                                ArithmeticExpression lhs) {
            return rhs.calculate() / lhs.calculate();
        }
    };
}
```

Composite: Utility Class Continued

```
public enum Operation {

    //...

    private String op;

    private Operation(String op) {
        this.op = op;
    }

    public abstract double calculate(ArithmeticExpression rhs,
ArithmeticExpression lhs);

    public String getOp() {
        return op;
    }
}
```

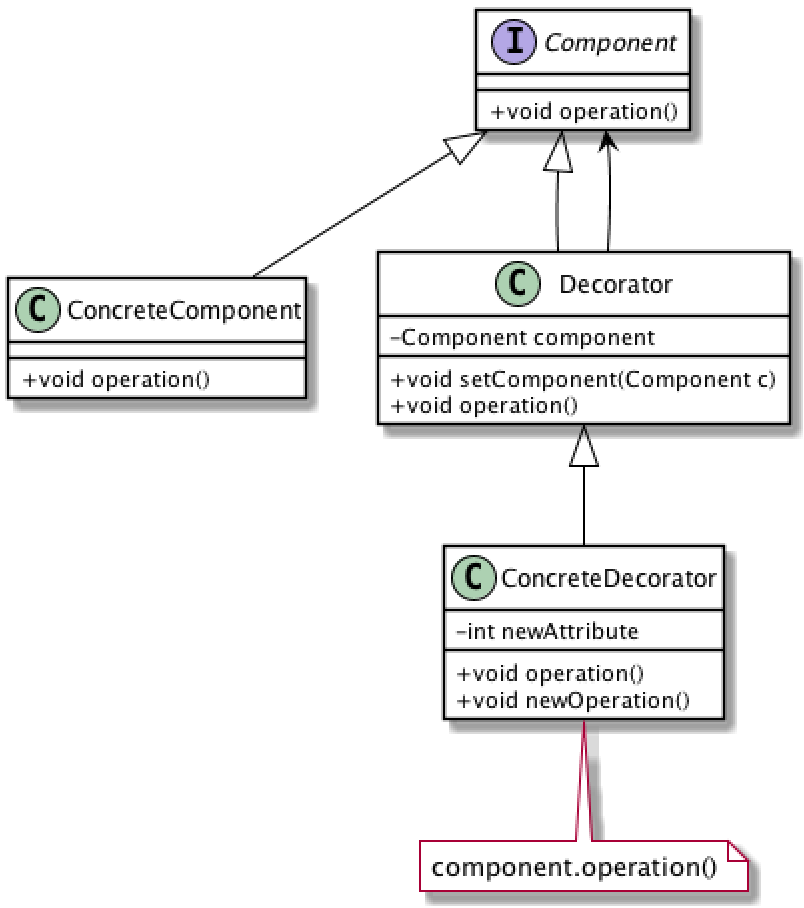
Decorator Pattern

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



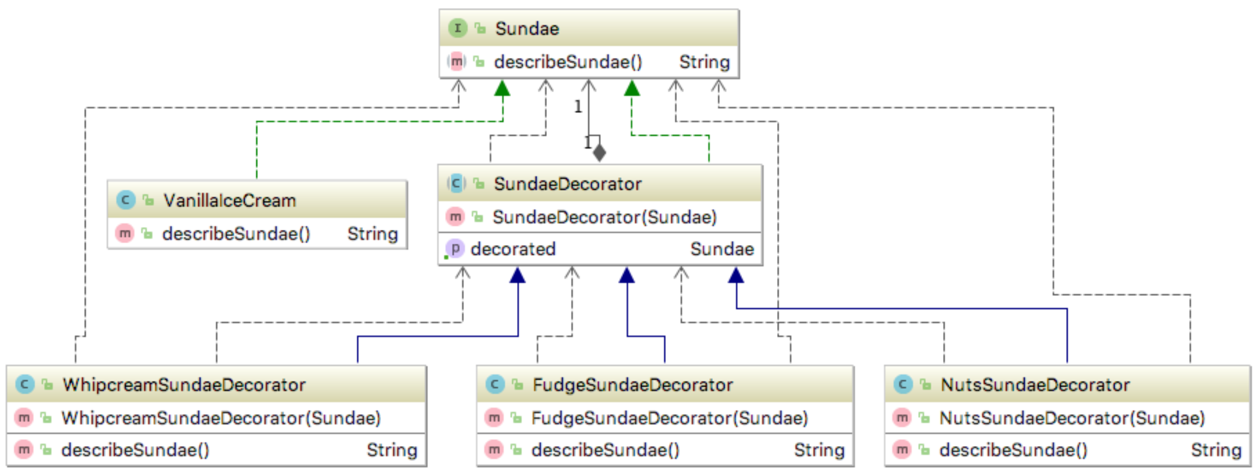
Decorator Pattern Advantages

- "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

Decorator Pattern Disadvantages

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

Decorator Demo Diagram



Base Interface

```
public interface Sundae {
    String describeSundae();
}
```

The Decorator

```
public abstract class SundaeDecorator implements Sundae {

    private Sundae decorated;

    public SundaeDecorator(Sundae decorated) {
        this.decorated = decorated;
    }

    public Sundae getDecorated() {
        return decorated;
    }
}
```

The Base Class

```
public class VanillaIceCream implements Sundae {
    public String describeSundae() {
        return "Vanilla Ice Cream";
    }
}
```

Sample Decorator Layer

```
public class WhipcreamSundaeDecorator extends SundaeDecorator {
    public WhipcreamSundaeDecorator(Sundae sundae) {
        super(sundae);
    }

    public String describeSundae() {
        return "Whipcream " + getDecorated().describeSundae();
    }
}
```

Another Sample Decorator Layer

```
public class NutsSundaeDecorator extends SundaeDecorator {
    public NutsSundaeDecorator(Sundae sundae) {
        super(sundae);
    }

    public String describeSundae() {
        return "Nuts " + getDecorated().describeSundae();
    }
}
```

And Yet, Another

Notice this one, this one changes the end result

```java
public class FudgeFilterDecorator extends SundaeDecorator {
    public FudgeFilterDecorator(Sundae cherryOnTopDecorator) {
        super(cherryOnTopDecorator);
    }

    @Override
    public String describeSundae() {
        return getDecorated().describeSundae().replaceAll("Fudge",
"xxxxx");
    }
}
```

Running a Decorator

```java
Sundae sundae = new NutsSundaeDecorator(
        new FudgeSundaeDecorator(
                new WhipcreamSundaeDecorator(
                        new VanillaIceCream()
                )
        )
);

System.out.println(sundae.describeSundae());

//add a cherry

Sundae cherryOnTopDecorator = new CherryOnTopDecorator(sundae);

System.out.println(cherryOnTopDecorator.describeSundae());

Sundae filteredFudge = new FudgeFilterDecorator(cherryOnTopDecorator);

System.out.println(filteredFudge.describeSundae());
```

Lab: Making an Functional Decorator

Step 1: In design_patterns_training, go to *src/test/java* and a package called `com.xyzcorp.decorator`, if they do not exist, create them.

Step 2: In the `com.xyzcorp.decorator` package, create a test class called `DecoratorTest`

Step 3: The decorator pattern can use an upgrade. We can probably use a `java.util.function.Function` in order to do decorating instead of a bunch of classes. First create a class called `Camera` that has a `color` property where `Color` is a `java.awt.Color`. Create an `equals`, `hashCode`, `toString`

Step 4: Create a method in `Camera` with either of the following signatures:

```java
public Camera decorate(Function<Color, Color>... filters) {
    ...
}
```

```java
public Camera decorate(List<Function<Color, Color>> filters) {
    ...
}
```

Step 5: This will not be easy. Use functional programming to join, or shall we say `compose`, these `Function` together. Look up or ask about `reduce` and return a **new** Camera that applies the current color to the combined function.

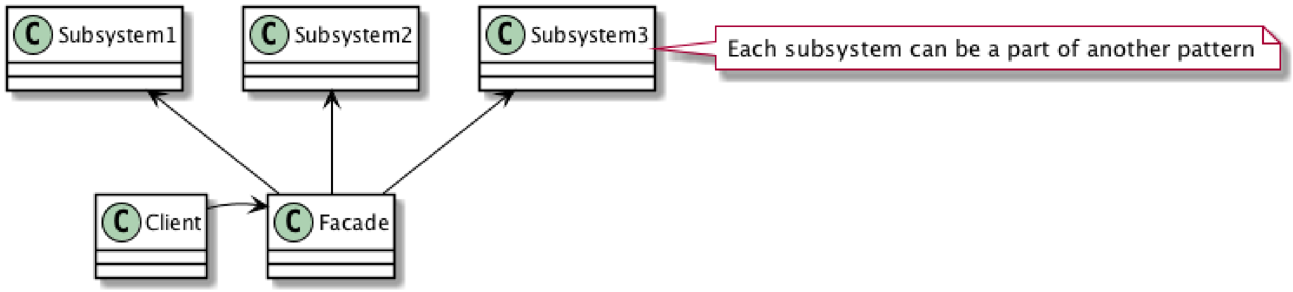
Step 6: Test the results by comparing `getColor` on the new `Camera` with the old `Camera`

Facade Pattern

Facade Pattern Properties

Type: Structural
Level: Component

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.

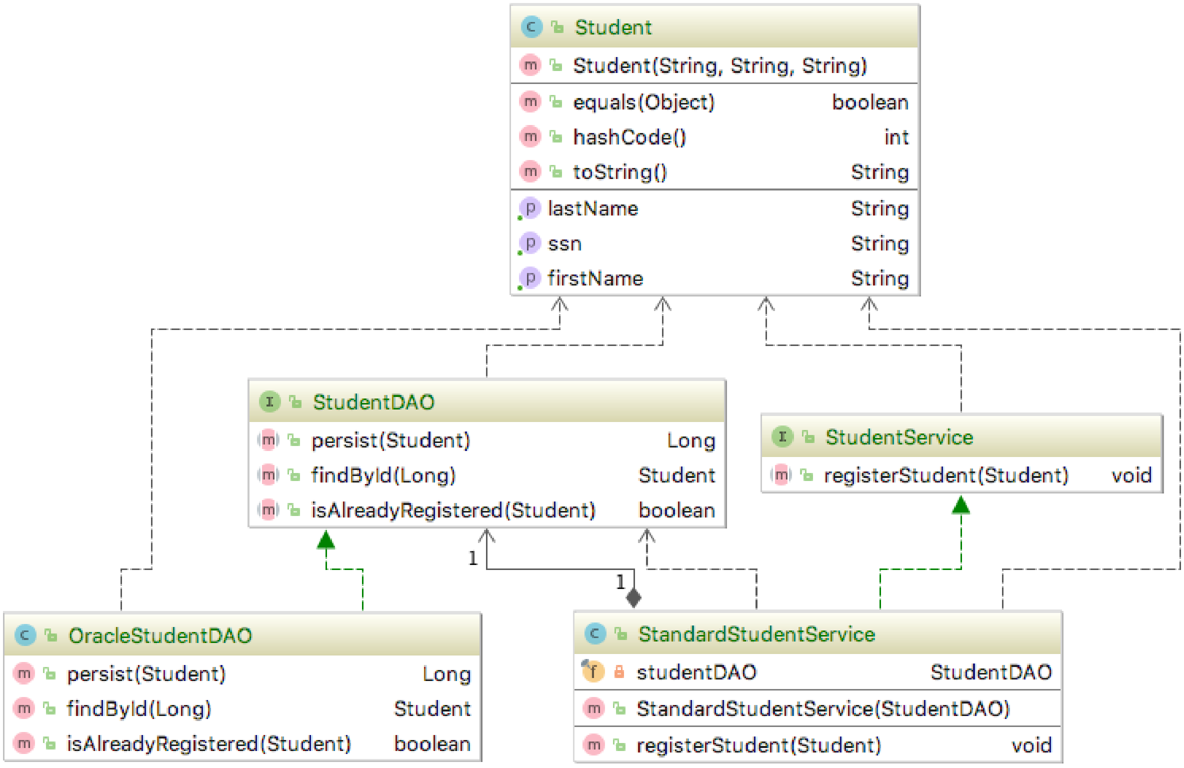
Facade Advantages

- 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

Facade Disadvantages

- Can be difficult to debug if subsystems gets too high

Facade Demo Diagram



Facade: The Facade's Interface

```
public interface StudentService {
    public void registerStudent(Student student);
}
```

Facade: The Facade's Implementation

Here, we create a "complication", albeit a small one.

```java
public class StandardStudentService implements StudentService {

    private final StudentDAO studentDAO;

    public StandardStudentService(StudentDAO studentDAO) {
        this.studentDAO = studentDAO;
    }

    @Override
    public void registerStudent(Student student) {
        if (!studentDAO.isAlreadyRegistered(student)) {
            studentDAO.persist(student);
        }
    }
}
```

Facade: The Facade's Dependency Interface

```java
public interface StudentDAO {
    public Long persist(Student student);
    public Student findById(Long id);
    public boolean isAlreadyRegistered(Student student);
}
```

Facade: The Facade's Dependency Implementation

```java
public class OracleStudentDAO implements StudentDAO {
    @Override
    public Long persist(Student student) {
      //insert lots of database code
    }

    @Override
    public Student findById(Long id) {
      //insert lots of database code
    }

    @Override
    public boolean isAlreadyRegistered(Student student) {
      //insert lots of database code
    }
}
```

Proxy Pattern

Proxy Pattern Properties

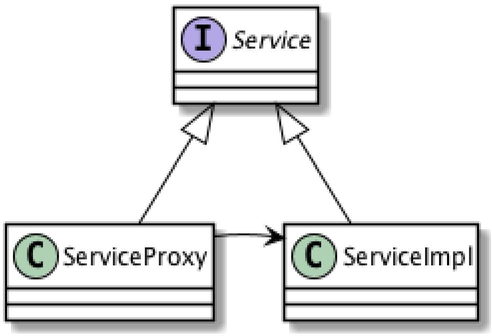
Type: Structural
Level: Component

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

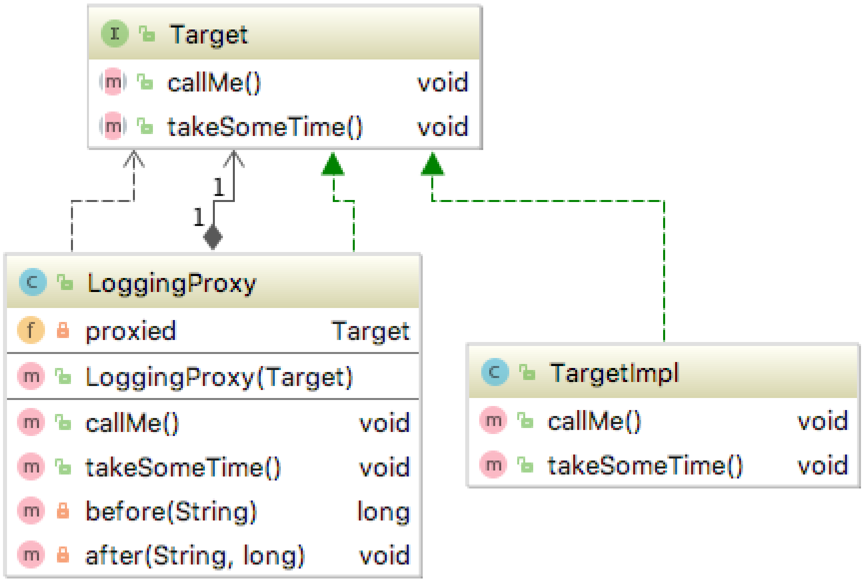
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Proxy Disadvantages

- Complicated setup
- Unnecessary for simple local reference

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Proxy Demo Diagram



Proxy: Target Interface

```
public interface Target {
    void callMe();
    void takeSomeTime();
}
```

Proxy: Target Implementation

```java
public class TargetImpl implements Target {

    public void callMe() {
        System.out.println("Called");
    }

    public void takeSomeTime() {
        try {
            Thread.sleep(1000);
            System.out.println("Took some time");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Proxy: The Proxy

The difference between a Proxy and an Adapter is that the Adapter can have a different interface, a Proxy must have the same interface and perhaps some supporting methods

```java
public class LoggingProxy implements Target {

    private Target proxied;

    public LoggingProxy(Target proxied) {
        this.proxied = proxied;
    }

    public void callMe() {
        long start = before("callMe()");
        proxied.callMe();
        after("callMe()", start);
    }

    public void takeSomeTime() {
        long start = before("takeSomeTime()");
        proxied.takeSomeTime();
        after("takeSomeTime()", start);
    }

    private long before(String name) {
        System.out.println("Before " + name);
        return System.currentTimeMillis();
    }

    private void after(String name, long start) {
        System.out.println("After: " + name + " took: " + (System
.currentTimeMillis() - start));
    }
}
```

Proxy: Use of a Proxy

```java
Target target = new TargetImpl();

target.callMe();
target.takeSomeTime();

System.out.println("Added Proxy");
Target targetProxy = new LoggingProxy(target);

targetProxy.callMe();
targetProxy.takeSomeTime();
```

Flyweight Pattern

Flyweight Pattern Properties

Type: Structural
Level: Component

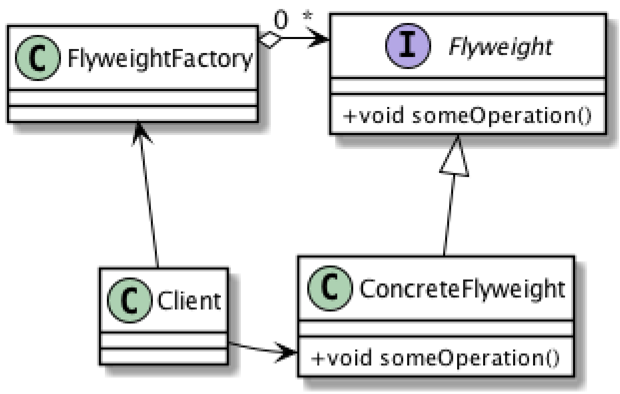
Source: Applied Java Patterns By Stephen Stelting, Olav Massen

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

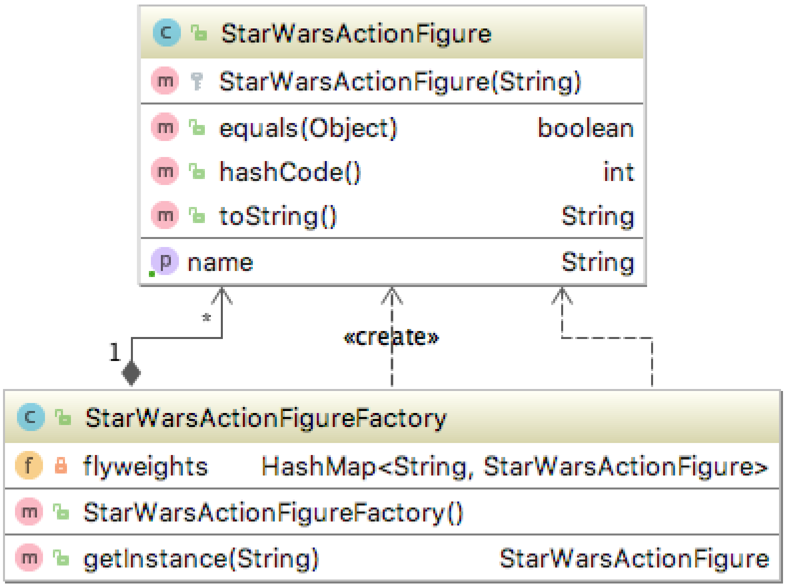
- Sharing an object among multiple clients occurs when you must manage thousands
- Provides for sharing an object between clients
- Save in memory
- Runtime can be efficient

Source: Applied Java Patterns By Stephen Stelting, Olav Massen

Flyweight Disadvantages

- None

Flyweight Demo Diagram



Flyweight Factory

```java
import java.util.HashMap;

public class StarWarsActionFigureFactory {
    private HashMap<String, StarWarsActionFigure> flyweights;

    public StarWarsActionFigureFactory() {
        this.flyweights = new HashMap<>();
    }

    public StarWarsActionFigure getInstance(String name) {
         if (flyweights.containsKey(name)) return flyweights.get(name);
         else {
             StarWarsActionFigure starWarsActionFigure =
                new StarWarsActionFigure(name);
             flyweights.put(name, starWarsActionFigure);
             return starWarsActionFigure;
         }
    }
}
```

Flyweight Objects

```java
public class StarWarsActionFigure {
    private String name;

    protected StarWarsActionFigure(String name) {
        this.name = name;
    }

    public String getName() {return name;}

    //toString, equals, hashCode
}
```

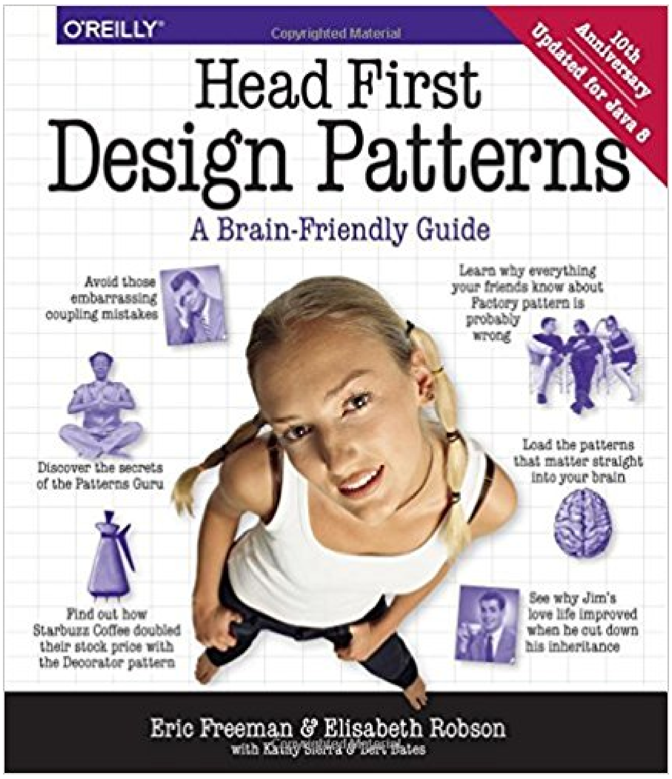
Discussion of Active Projects

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
A Brain-Friendly Guide
By Bert Bates, Kathy Sierra, Eric Freeman, Elisabeth Robson
Publisher: O'Reilly Media
Release Date: June 2009
Pages: 688

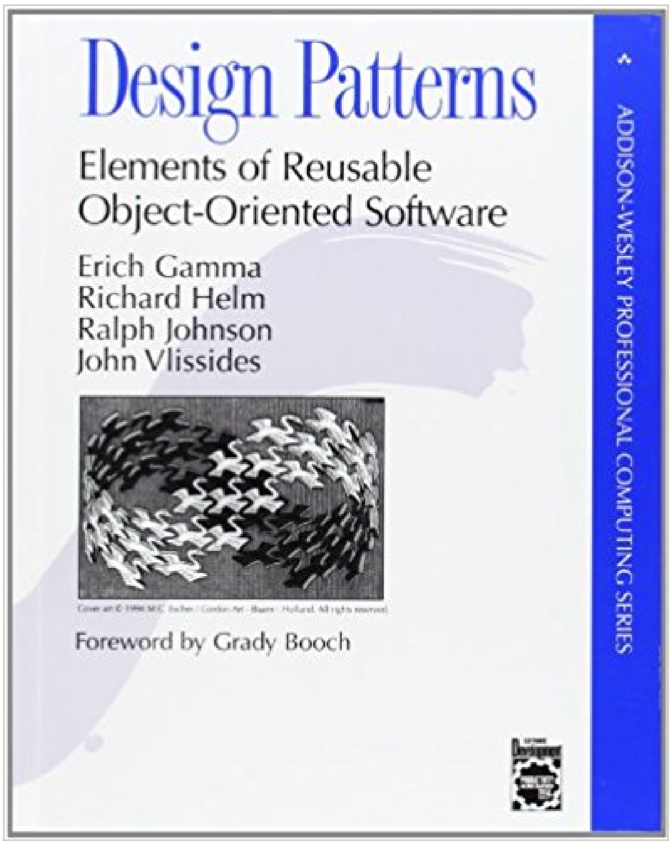
http://shop.oreilly.com/product/9780596007126.do
Available on O'Reilly Safari

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

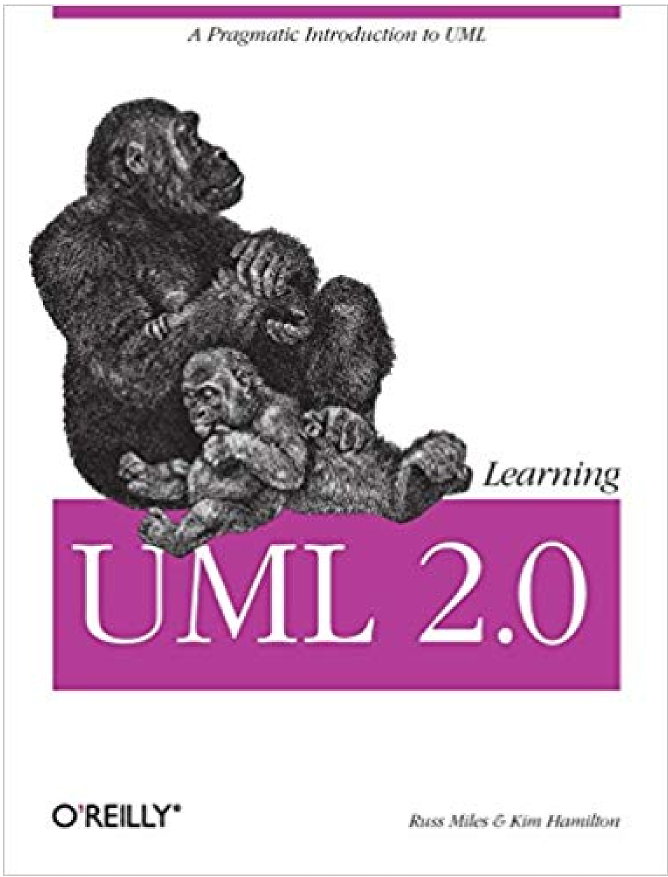
Design Patterns Elements Resusable Object Oriented Software



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
Available on O’Reilly Safari

Thank You

- Email: dhinojosa@evolutionnext.com
- Github: https://www.github.com/dhinojosa
- Twitter: http://twitter.com/dhinojosa
- Linked In: http://www.linkedin.com/in/dhevolutionnext