Design Patterns

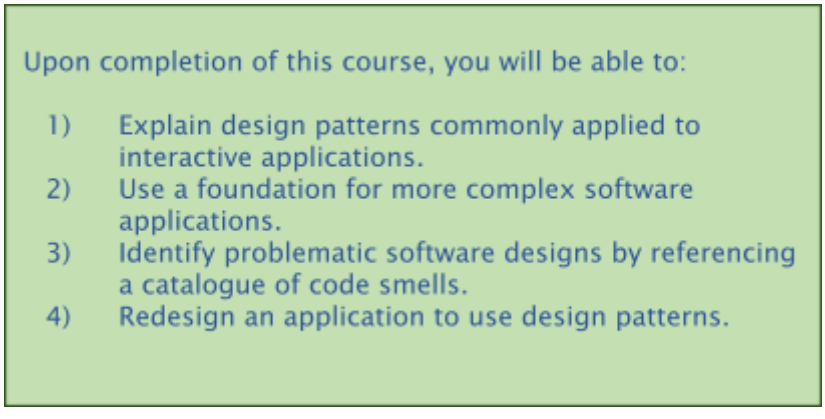
COURSE NOTES
Updated July 2021

UNIVERSITY OF ALBERTA

Table of Contents

Course Overview

Welcome to the second course in the Software Design and Architecture specialization, brought to you in partnership by the University of Alberta and Coursera. This course focuses on **design patterns**. Design issues in applications can be resolved through design patterns commonly applied by experts. This second course extends your knowledge of object-oriented analysis and design by covering design patterns used in interactive applications. Through a survey of established design patterns, you will gain a foundation for more complex software applications. This course will also explain several design principles that you can use to make your software more reusable, flexible, and maintainable. Finally, you will learn how to identify problematic software designs by referencing what is known as a catalog of code smells.



Module 1: Introduction to Design Patterns: Creational and Structural Patterns

Upon completion of this module, you will be able to:

1) Explain what a design pattern is.
2) Describe the Gang of Four pattern catalog.
3) Describe the Singleton pattern.
4) Describe the Factory Method pattern.
5) Describe the Façade pattern.
6) Describe the Adapter pattern.
7) Describe the Composite pattern.
8) Describe the Proxy pattern.
9) Describe the Decorator pattern.

Introduction to Design Patterns

The first course of this specialization focused on object-oriented analysis and design, including focusing on the major design principles of abstraction, encapsulation, decomposition, and generalization. This second course will extend this knowledge by teaching you how to apply **design patterns** to address design issues.

Over the course of your career in software engineering, you will find that the same design problems reoccur. There are many ways to deal with these problems, but more flexible or reusable solutions are preferred in the industry. One of these preferred methods is that of design patterns.

A design pattern is a practical proven solution to a recurring design problem. It allows you to use previously outlined solutions that expert developers have often used to solve a software problem, so you do not need to build a solution from the basics of object-oriented programming principles every time. These solutions are not just theoretical – they are actual solutions used in the industry. Design patterns may also be used to help fix tangled, structureless software code, also known as "spaghetti code."

A good way to think of design patterns is like recipes in a cookbook, which have been experimented upon and tested by many people over time. A particular recipe may become preferred because it creates the best-tasting dish. Similarly, design patterns have been experimented upon and tested by developers over many years. Design patterns outline solutions that often create the best outcome.

It is not always obvious which design pattern to use to solve a software design problem, especially as many design patterns exist. Deciding which one to use is similar to a game of chess – there are many possible moves and strategies that can be used to win in chess. Through experience and examining the

situation on the board, you will become better at judging which strategy to use to win a game or solve a problem. Like chess, many people start out only knowing basics, such as elements of language syntax. But, practice and experience will make you better at selecting which design patterns to use for different problems. Some people even become design experts who know the design patterns to use in solving particular software design problems!

It is important to understand that design patterns are not just a concrete set of source code that you memorize and put into your software, like a Java library or framework. Instead, design patterns are more conceptual. They are knowledge that you can apply within your software design to guide its structure and make it more flexible and reusable. In other words, design patterns help software developers so that they have a guide to help them solve design problems the way an expert might, so not everything needs to be built from scratch. Design patterns serve almost like a coach to help developers reach their full potential!

A strong advantage of design patterns is that they have already been proven by experts. This means that you do not need to go through the trials they have, and you go straight to creating better-written software.

Another advantage of design patterns is that they help create a design vocabulary. This means that design patterns provide a simplified means of discussing design solutions, so they do not need to be explained over and over. For example, design patterns might give patterns names, making it easier to discuss them. This saves time and ensures that everyone is referring to the same pattern. This leaves less room for misunderstanding.

One of the most famous books on design patterns is *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. This course will draw upon a selection of the patterns presented in this book.

Gang of Four's Pattern Catalogue

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, the authors of *Design Patterns: Elements of Reusable Object-Oriented Software*, have been collectively nicknamed the **Gang of Four**. These authors wrote their book based on their

own experiences as developers. When each had developed programs and graphical applications, they had discovered patterns emerging in their design solutions. As a group, they formalized those patterns into a reference book.

The patterns developed by the Gang of Four were organized to be readable and were often named after their purpose. The grouping of patterns together formed **Gang of Four's design pattern catalog.**

The Gang of Four's particular design pattern catalog is not actually like a recipe book, as described in the last lesson, but is closer to a list of tropes. You cannot pull universal patterns from the catalog to use in your code, like a recipe. Instead, design patterns in software reoccur in the same way tropes reoccur in storytelling.

A trope is a storytelling device or convention often found in fiction. Films and TV shows use a pattern of storytelling. A well-known example of a trope is the "hero's journey." This trope outlines a common pattern where a hero goes on an adventure and faces trials and wins a victory before coming home changed or transformed. This trope, or storytelling device, can be found in thousands of pieces of fiction. Similarly, occurrences of the same pattern can be found in thousands of programs.

An example of a simple pattern would be defining and calling methods.

Learning to look for specific design patterns will help you better recognize object-oriented design elsewhere.

Pattern Languages

The patterns and solutions of the Gang of Four's catalog serve a variety of different purposes. Depending on the context of the problem, you would select a different **pattern language** to use. A pattern language is a collection of patterns that are related to a certain problem space. For example, the pattern language you select for designing accounting software would be different from those you select for designing gaming software. A pattern language for accounting software would include double-entry bookkeeping, while a pattern language for gaming software would include design words such as encounters, quests, and players.

Based on context, you must decide which ones will suit your problem or design issue the best. However, sometimes you must consider trade-offs in design – some patterns may be more resource-intensive.

Categories of Patterns

The Gang of Four's pattern catalog contains 23 patterns. These patterns can be sorted into three different categories: **creational patterns, structural patterns,** and **behavioral patterns**. Some patterns might have elements that allow them to span all of the categories – it is not always clear cut which categories a pattern falls under. These categories were used to simply organize and characterize the patterns in their book.

Creational Patterns

Creational patterns tackle how you handle creating or cloning new objects. **Cloning** an object occurs when you are creating an object that is similar to an existing one, and instead of instantiating a new object, you clone existing objects instead of instantiating them.

Creational patterns depend on the programming language being used. Languages without the notion of classes, such as Javascript, would encourage you to clone an object and expand it to meet the purposes of those particular instances, called prototypes. Javascript allows for changes to these prototypes at runtime. Languages that rely on classes, however, such as Java and C#, instantiate objects using specific classes defined at compile time.

The different ways of creating objects will greatly influence how a problem is solved. Therefore, different languages therefore impact what patterns are possible to use.

Structural Patterns

Structural patterns describe how objects are connected to each other. These patterns relate to the design principles of decomposition and generalization, as discussed in the first course in this specialization.

There are many different ways that you can structure objects depending on the relationship you'd like between them. Not only do structural patterns describe how different objects have relationships, but they also describe how subclasses and

classes interact through inheritance. Structural patterns use these relationships and describe how they should work to achieve a particular design goal. Each structural pattern determines the various suitable relationships among the objects.

A good metaphor for considering structural patterns is that of pairing different kinds of foods together: flavor determines what ingredients can be mixed together to form a suitable relationship. Some relationships combine ingredients together, such as chickpeas and garlic in hummus. Some relationships still combine ingredients, but those ingredients may maintain some independence, like a salad of mixed vegetables. Some relationships allow a pairing of ingredients, without a physical combination, like wine and cheese.

Behavioral Patterns

Behavioral patterns focus on how objects distribute work and describe how each object does a single cohesive function. Behavioral patterns also focus on how independent objects work towards a common goal.

A good metaphor for considering behavioral patterns is that of a racing car pit crew at a track. Every member of the crew has a role, but together they work as a team to achieve a common goal. Similarly, a behavioral pattern lays out the overall goal and purpose for each object.

Singleton Pattern

A **singleton** is a creational pattern, which describes a way to create an object. It is a powerful technique, but it is also one of the simplest examples of a design pattern.

A singleton design pattern only has one object of a class. This might be desirable in order to circumvent conflicts or inconsistencies, by keeping clear which object the program should draw from. For example, the preferences of an app, the print queue of your printer, or the software driver for a device are all objects where it is preferable to only have one. If there are multiple instances, it can be confusing for the program output.

Another goal of the singleton design pattern is that the single object is globally accessible within the program.

In order to implement a singleton design pattern, the best practice is to build the "one and only one" goal into the class itself so that creating another instance of a Singleton class is not even possible. This "codifies" the design intent within the software. This is necessary if working on a large project or on projects with multiple developers, but it is helpful even on smaller or individual projects.

Let us examine this in code.

If a class has a public constructor, an object of this class can be instantiated at any time.

```
public class notSingleton {
   //Public Constructor
   public notSingleton() {
       ..
   }
}
```

Instead, if the class has a private constructor, then the constructor cannot be called from outside the class. Although this seems to prevent creating an object of the class, there are two key components for getting around this.

First, declare a class variable. In this case, it is called "unique Instance." This class variable will refer to the one instance of your Singleton class. As the variable is private, it can only be modified within the class.

Second, create a public method in the class that will create an instance of this class, but only if an instance does not exist already. In this case, the method is called "**getInstance**." It will check if the "unique Instance" variable is null. If it is null, then it will instantiate the class and set this variable to reference the object. On the other hand, if the "unique Instance" class variable currently references an object, meaning that there is already an object of this class, then the method will simply return that object. As the **getInstance()** method is public, it can be called globally and used to create one instance of the class. In a sense, it replaces the normal constructor.

```
public class ExampleSingleton { // lazy construction
   // the class variable is null if no instance is
   // instantiated
   private static ExampleSingleton uniqueInstance = null;

   private ExampleSingleton() {
       ...
   }

   // lazy construction of the instance
   public static ExampleSingleton getInstance() {
       if (uniqueInstance == null) {
           uniqueInstance = new ExampleSingleton();
       }

   return uniqueInstance;

   }

   ...
}
```

In the example above, the regular constructor is hidden. Other classes are forced to call the public "**getInstance**" method. This puts in place basic gatekeeping and ensures that only one object of this class is created. The same method can be used to globally reference a single object if it has already been created.

An advantage of this version of a Singleton class is **lazy creation**. Lazy creation means that the object is not created until it is truly needed. This is helpful, especially if the object is large. As the object is not created until the "**getInstance**" method is called, the program is more efficient.

There are trade-offs to the Singleton design principle. If there are multiple computing threads running, there could be issues caused by the threads trying to access the shared single object.

In real use, there may be variations of how Singleton is realized because design patterns are defined by purpose and not exact code. The intent of a Singleton pattern is to provide global access to a class that is restricted to one instance. In general, this is achieved by having a private constructor with a public

method that instantiates the class "if" it is not already instantiated.

Factory Method Pattern

The **Factory Method Pattern** is a creational pattern. In order to understand the Factory method pattern, we must first understand **factory objects**.

Factory Objects

A factory object operates like a factory in the real world, and creates objects. Factory objects make software easier to maintain, change, test, and reuse because it deals with the problem of creating objects without having to specify the class. The methods that use these **factories** can then focus on other behaviors.

Imagine a situation where you have software that implements an online store that sells knives. Perhaps when the store first opens, you only produce steak knives and chef's knives. You would have a superclass with those two subclasses. In your system, first, a knife object is created. Conditionals then determine which subclass of **Knife** is actually instantiated. This act of instantiating a class to create an object of a specific type is known as **concrete instantiation**. In Java, concrete instantiation is indicated with the operator "new."

However, imagine that the store is successful and adds more knife types to sell. New subclasses will need to be added, such as **BreadKnife**, **ParingKnife**, or **FilletKnife**. The list of conditionals would need to grow and grow as new knife types are added. However, the methods of the knife, such as sharpening, polishing, and packaging, would likely stay the same, no matter the type of knife. This creates a complicated situation. Instead of making **Knive** in the store, it may be better to create them in a Factory object.

A factory object is an object whose role is to create "product" objects of particular types. In this example, the methods of sharpening, polishing, and packaging would remain in the **orderKnife** method. However, the responsibility of creating the product will be delegated to another object: a "Knife Factory." The code for deciding which knife to create and the code for deciding which subclass of **Knife** to instantiate, are moved into the class for this factory.

Below is an example of the code that might be used for this example:

```
public class KnifeFactory {
  public Knife createKnife(String knifeType) {
      Knife knife = null;
      // create Knife object
      if (knifeType.equals("steak")) {
          knife = new SteakKnife();
      } else if (knifeType.equals("chefs")) {
          knife = new ChefsKnife();
      }
      return knife;
  }
}
```

The code to create a **Knife** object has been moved into a method of a new class called **KnifeFactory**. This allows the **KnifeStore** class to be a client for the **KnifeFactory**. The **KnifeFactory** object to use is passed into the constructor for the **KnifeStore** class. Instead of performing concrete instantiation itself, the orderKnife method delegates the task to the factory object.

```
public class KnifeStore {

    private KnifeFactory factory;

    // require a KnifeFactory object to be passed
    // to this constructor:
    public KnifeStore(KnifeFactory factory) {
        this.factory = factory;
    }

    public Knife orderKnife(String knifeType) {

         Knife knife;

         //use the create method in the factory
         knife = factory.createKnife(knifeType);
```

```
            //prepare the Knife
            knife.sharpen();
            knife.polish();
            knife.package();

            return knife;
        }
}
```

Concrete instantiation is the primary purpose of Factories. In general, a factory object is an instance of a factory class, which has a method to create product objects.

Benefits of Factory Objects

There are numerous benefits to using factory objects. One of these benefits is that it is much simpler to add new types of an object to the object factory without modifying the client code. Instead of hunting down multiple snippets of similar instantiation code, subclasses can simply be added or removed in the Factory. This only requires changing code in the Factory, or to the concrete instantiation, and not the client method.

Factories allow client code to operate on generalizations. This is known as **coding to an interface, not an implementation**: the client method does not need to name concrete knife classes and now deals with a Knife "generalization." As long as the client code receives the object it expects, it can satisfy its responsibilities without worrying about the details of object creation.

The use of the word "factory" serves as a good metaphor here. Using Factory objects is similar to using a factory in real life to create knives – the stores do not usually make the knives themselves, but get them from a factory. In fact, a factory may have multiple clients that they make knives for! If the factory changes how knives are made, the stores themselves will not care as long as they still receive knives of the right types.

Essentially, using factory objects means that you have cut out redundant code and made the software easier to modify, particularly if there are multiple clients that want to instantiate the same set of classes. In other words, if many parts of the software want to create the same objects, factory objects are

useful tools. Subclasses of the factory class can even become their own specialized factories!

Factory Method Pattern

The factory object is not actually a design pattern onto itself. The Factory method pattern does not use a factory object to create the objects; instead, the Factory method uses a separate "method" in the same class to create objects. The power of the Factory method comes in particular from how they create specialized product objects.

Generally, in order to create a "specialized" product object, a Factory Object approach would subclass the factory class. For example, a subclass of the **KnifeFactory** called **BudgetKnifeFactory** would make **BudgetChefsKnife** and **BudgetSteakKnife** product objects.

A Factory Method approach, however, would use a **BudgetKnifeStore** subclass of **KnifeStore**. The **BudgetKnifeStore** has a "Factory Method" that is responsible for creating **BudgetChefsKnife** and **BudgetSteakKnife** product objects instead. This design pattern's intent is to define an interface for creating objects but lets the subclasses decide which class to instantiate. So, instead of working with a factory object, we specialize or subclass the class that uses the Factory Method. Each subclass must define its own Factory Method. This is known as **letting the subclasses decide** how objects are made.

Let us examine this as code.

```
public abstract class KnifeStore {
    public Knife orderKnife(String knifeType) {
        Knife knife;
        // now creating a knife is a method in the class
        knife = createKnife(knifeType);
        knife.sharpen();
        knife.polish();
        knife.package();
        return knife;
    }
    abstract Knife createKnife(String type);
}
```

In this example, a **KnifeStore** is abstract, and cannot be instantiated. Instead, subclasses such as **BudgetKnifeStore** inherit the **orderKnife** method. The **orderKnife** method uses a Factory, but it is a factory method, **createKnife()**. It is declared in the superclass, but it is abstract and empty. It is abstract because we want the Factory method to be defined by the subclasses. When a **KnifeStore** subclass is defined, it "must" define this **createKnife** method.

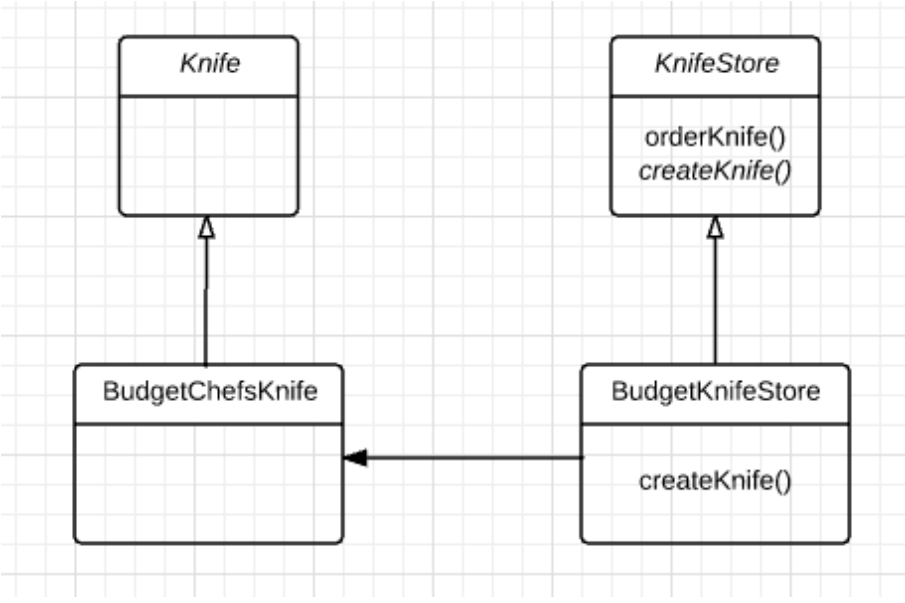
Let us examine this subclass as code.

```
public BudgetKnifeStore extends KnifeStore {

   //up to any subclass of KnifeStore to define this method
   Knife createKnife(String knifeTYpe) {
       if (knifeType.equals("steak")) {
           return new BudgetSteakKnife();
       } else if (knifeType.equals("chefs")) {
           return new BudgetChefsKnife();
       }
       //.. more types
       else return null;
   }
}
```

Because the **BudgetKnifeStore** is a subclass of **KnifeStore**, it inherits the **orderKnife** method. The **orderKnife** method can be run from any subclass of **KnifeStore**.

UML Diagrams

A class's subclasses will all inherit from the same method. However, each subclass must define its own methods as well. This can be represented using UML diagrams. Let us examine a UML diagram based on the Knife Store example used throughout this lesson.



In this diagram, the "Knife" and "KnifeStore" classes are italicized to indicate that they are abstract classes that cannot be instantiated. Instead, their subclasses must be defined. In this case, there is only one subclass for **Knife**, but you could have several more. Similarly, there is only one **KnifeStore** in this example, but theoretically, several more could exist, which would make other types of knives. Note that the **createKnife()** method is also abstract under the **KnifeStore** class. This indicates that any subclass of Knife Store must define this method.

In summary, this diagram indicates to us that **BudgetKnifeStore**, and any other **KnifeStore** subclass defined must have its own **createKnife()** method.

This structure is the core of the Factory Method Design Pattern. We can abstract out this example to demonstrate a more general structure.

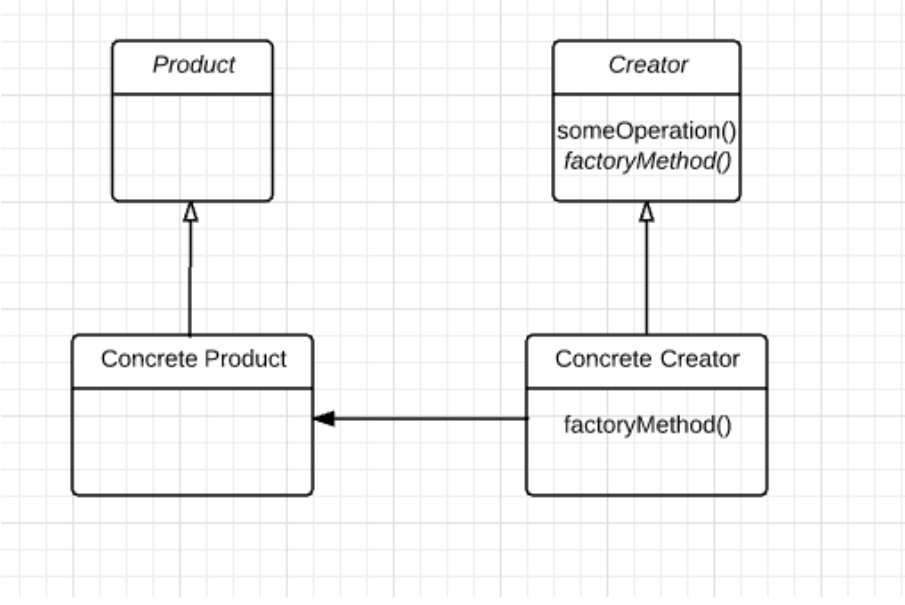
There should be an abstract **Creator** class that contains methods that only operate on generalizations. The Factory Method is declared by the Creator abstractly, so each Concrete Creator class is obliged to provide a Factory Method.

There should also be a subclass of the abstract Creator class, a Concrete Creator that is responsible for concrete instantiation. The Concrete Creator inherits methods from the abstract Creator.

There should be a **factoryMethod()** in the concrete creator subclass. Every time a Concrete **Creator** subclass is added to the design, the **factoryMethod()** must be defined to make the right products. This is how the subclass "decides" to create objects.

Finally, the Product superclass generalizes the **Concrete Products**.

The general structure of a Factory Method pattern is illustrated in the UML diagram below.



The methods of the Concrete Creator class only operate on the general Product, never the Concrete Products. The Concrete Products are made by the Concrete Creator. The type of product made is decided by which Concrete Creator is made.

If object creation is separate from other behavior, the code becomes cleaner to read and easier to maintain or change. The client code is simplified. The code becomes more extensible, and inheritance allows for the specialization of object creation.

Façade Pattern

As systems or parts of systems become larger, they also become more complex. This is not necessarily a bad thing – if the scope of a problem is large, it may require a complex

solution. Client classes function better with a simpler interaction, however. The **façade design pattern** attempts to resolve this issue by providing a single, simplified interface for client classes to interact with a subsystem. It is a structural design pattern.

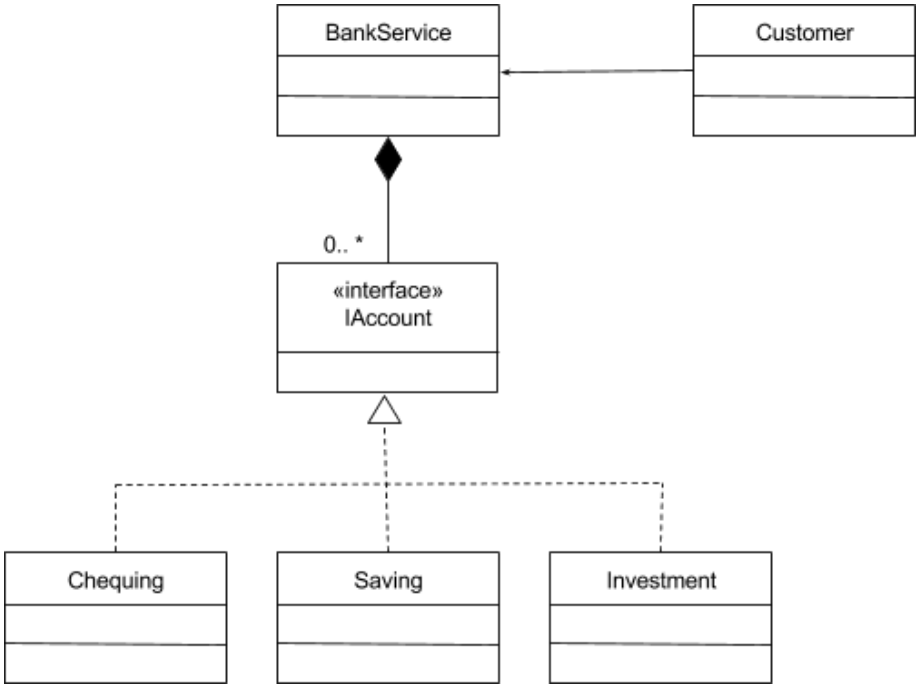
A **façade** is a wrapper class that encapsulates a subsystem in order to hide the subsystem's complexity; it acts as a point of entry into a subsystem without adding more functionality in itself. The wrapper class allows a client class to interact with the subsystem through the façade. A façade might be compared metaphorically to a waiter or salesperson, who hides all the extra work to be done in order to purchase a good or service.

A façade design pattern should therefore be used if there is a need to simplify the interaction with a subsystem for client classes and if there is a need for a class to instantiate other classes within your system and to provide these instances to another class. Often façade design patterns combine interface implementation by one or more classes, which then gets wrapped by the façade class. This can be explained through a number of steps.

1. Design the interface.
2. Implement the interface with one or more classes.
3. Create the façade class and wrap the classes that implement the interface.
4. Use the façade class to access the subsystem.

Let us examine each of these steps with an example for a bank system.

In the UML diagram below, we can see that a **BankService** class acts as a façade for **Chequing**, **Saving**, and **Investment** classes. As all three accounts implement the **iAccount** interface, the **BankService** class wraps the account interface and classes and presents a simpler "front" for the Customer client class.



Let us look at how to do this with code for each step outlined above.

Step 1: Design the Interface

First, create an interface that will be implemented by the different account classes, but will not be known to the Customer class.

```
public interface IAccount {
	public void deposit(BigDecimal amount);
	public void withdraw(BigDecimal amount);
	public void transfer(BigDecimal amount);
	public int getAccountNumber();
}
```

Step 2: Implement the Interface with one or more classes

Implement the interface with classes that will be wrapped with the façade class. Note that in this simple example, only one interface is being implemented and hidden, but in practice, a

façade class can be used to wrap all the interfaces and classes for a subsystem.

> Instructor's Note: Remember that interfaces allow the creation of subtypes! This means that in this example, Chequing, Saving, and Investment are subtypes of IAccount, and they are expected to behave like an account type.

```
public class Chequing implements IAccount { … }
public class Saving implements IAccount { … }
public class Investment implements IAccount { … }
```

Step 3: Create the façade class and wrap the classes that implement the interface

The **BankService** class is the façade. Its public methods are simple to use and show no hint of the underlying interface and implementing classes. The **information hiding** principle is used here to prevent client classes from "seeing" the account objects, and how these accounts behave – note that access modifiers for each **Account** have been set to private.

```
public class BankService {
   private Hashtable<int, IAccount> bankAccounts;

   public BankService() {
       this.bankAccounts = new Hashtable<int, IAccount>
   }

   public int createNewAccount(String type, BigDecimal
initAmount) {
       IAccount newAccount = null;
       switch (type) {
           case "chequing":
               newAccount = new Chequing(initAmount);
               break;
           case "saving":
               newAccount = new Saving(initAmount);
```

```
                break;
            case "investment":
                newAccount = new Investment(initAmount);
                break;
            default:
                System.out.println("Invalid account type");
                break;
        }
        if (newAccount != null) {
            this.bankAccounts.put(newAccount.getAccountNumber(),
newAccount);
            return newAccount.getAccountNumber();
        }
        return -1;
    }

    public void transferMoney(int to, int from, BigDecimal
amount) {
        IAccount toAccount = this.bankAccounts.get(to);
        IAccount fromAccount = this.bankAccounts.get(from);
        fromAccount.transfer(toAccount, amount);
    }
}
```

Step 4: Use the façade class to access the subsystem

With the façade class in place, the client class can access accounts through the methods of the **BankService** class. The **BankService** class will tell the client what type of actions it will allow the client to call upon and then it will delegate that action to the appropriate **Account** object.

```
public class Customer {
    public static void main(String args[]) {
        BankService myBankService = new BankService();
        int mySaving = myBankService.createNewAccount("saving",
new BigDecimal(500.00));
```

```
        int myInvestment =
myBankService.createNewAccount("investment", new
BigDecimal(1000.00));
        myBankService.transferMoney(mySaving, myInvestment, new
BigDecimal(300.00));
    }
}
```

Façade design patterns draw on a number of different design principles. Subsystem classes are encapsulated into a façade class. Encapsulation is also demonstrated through information hiding subsystem classes from client classes. This also represents a separation of concerns.

In summary, the façade design pattern:

- Hides the complexity of a subsystem by encapsulating it behind a unifying wrapper called a façade class.
- Removes the need for client classes to manage a subsystem on their own, which results in less coupling between the subsystem and the client classes.
- Handles instantiation and redirection of tasks to the appropriate class within the subsystem.
- Provides client classes with a simplified interface for the subsystem.
- Acts simply as a point of entry to a subsystem and does not add more functional subsystems.

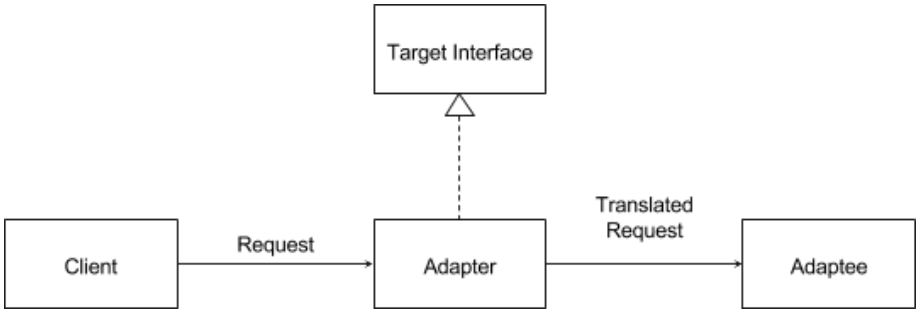
Adapter Pattern

Physically, an adapter is a device that is used to connect pieces of equipment that cannot be connected directly. Software systems may also face similar issues: not all systems have compatible software interfaces. In other words, the output of one system may not conform to the expected input of another system. This frequently happens when a pre-existing system needs to incorporate third-party libraries or needs to connect to other systems. The **adapter design pattern** facilitates communication between two existing systems by providing a compatible interface. It is a structural design pattern.

The adapter design pattern consists of several parts.

1. A client class. This class is the part of your system that wants to use a third-party library or external system.
2. An adaptee class. This class is the third-party library or external system that is to be used.
3. An adapter class. This class sits between the client and the adaptee. The adapter conforms to what the client is expecting to see, by implementing a target interface. The adapter also translates the client request into a message that the adaptee will understand, and returns the translated request to the adaptee. The adapter is a kind of wrapper class.
4. A target interface. This is used by the client to send a request to the adapter.

Translated into a diagram, the pattern looks as below:



Implementation of an adapter design pattern can also be broken down into steps.

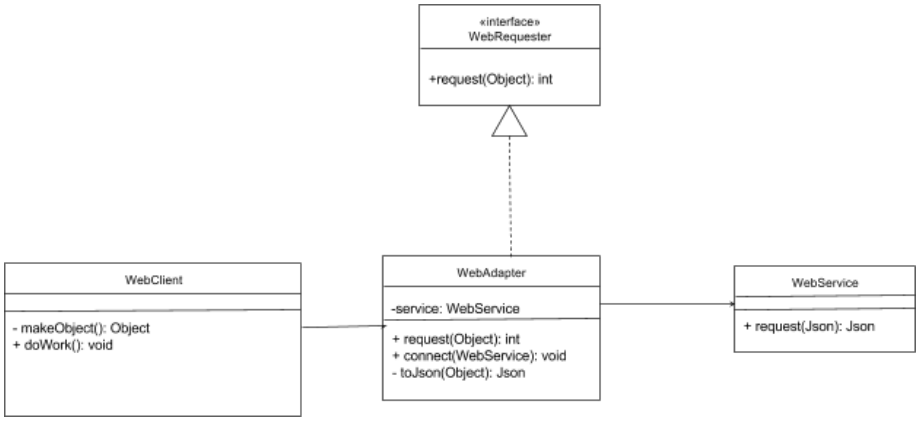
1. Design the target interface.
2. Implement the target interface with the adapter class.
3. Send the request from the client to the adapter using the target interface.

Let us examine each of these steps using a specific example.

Step 1: Design the target interface

Imagine an example where there is a pre-existing web client that we would like to interact with another web service. However, the service only supports JSON objects, and an adapter is needed to convert our Object request into a JSON object.

Here is the UML diagram for this situation:



First, create the target interface that your adapter class will be implementing for your client class to use:

```
public interface WebRequester {
    public int request(Object);
}
```

Step 2: Implement the target interface with the adapter class

The adapter class provides the methods that will take the client class's object and convert it into a JSON object. The adapter should convert any instance of a class that the client can create and send that in a request. The adapter class also transfers the translated request to the adaptee. The client class therefore only needs to know about the target interface of the adapter.

```
public class WebAdapter implements WebRequester {
    private WebService service;

    public void connect(WebService currentService) {
        this.service = currentService;
    /* Connect to the web service */
    }

    public int request(Object request) {
        Json result = this.toJson(request);
```

```
        Json response = service.request(result);
        if (response != null)
            return 200; // OK status code
        return 500; // Server error status code
    }

    private Json toJson(Object input) { ... }
}
```

Step 3: Send the request from the client to the adapter using the target interface

The web client normally returns an object back to the client. In light of this, the **doWork()** method should not be modified as it may disrupt other parts of the system. Instead, the **Web Client** should perform this behavior as normal and add in a send message method, where you can pass in the adapter, the web service, and any message you want to send.

```
public class WebClient {
    private WebRequester webRequester;

    public WebClient(WebRequester webRequester) {
        this.webRequester = webRequester;
    }

    private Object makeObject() { … } // Make an Object

    public void doWork() {
        Object object = makeObject();
        int status = webRequester.request(object);

        if (status == 200) {
            System.out.println("OK");
        } else {
            System.out.println("Not OK");
        }
        return;
    }
}
```

In the main program, the **Web Adapter**, the **Web Service**, and the **Web Client** need to be instantiated. The **Web Client** deals with the adapter through the **Web Requester** interface to send a request. The **Web Client** should not need to know anything about the **Web Service**, such as its need for JSON objects. The adaptee is hidden from the client by the wrapping adapter class.

```
public class Program {
   public static void main(String args[]) {
        String webHost = "Host: https://google.com\n\r";
        WebService service = new WebService(webHost);
        WebAdapter adapter = new WebAdapter();
        adapter.connect(service);
        WebClient client = new WebClient(adapter);
        client.doWork();
   }
}
```

Although it might be tempting to think that a solution is to simply change an interface so that it is compatible with another one, this is not always feasible, especially if the other interface is from a third-party library or external system. Changing your system to match the other system is not always a solution either, because an update by the vendors to the outside systems may break part of our system.

An adapter is an effective solution. In summary, an adapter is meant to:

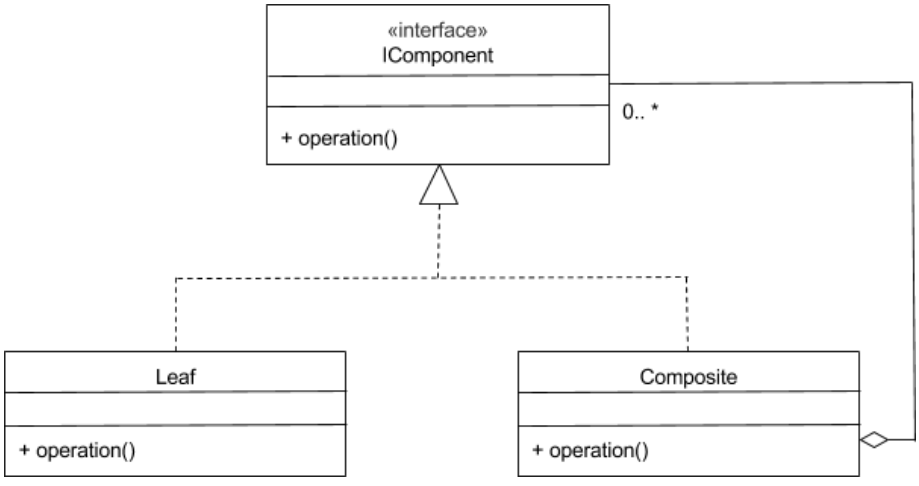
- Wrap the adaptee and exposes a target interface to the client.
- Indirectly change the adaptee's interface into one that the client is expecting by implementing a target interface.
- Indirectly translate the client's request into one that the adaptee is expecting.
- Reuse an existing adaptee with an incompatible interface.

Composite Pattern

A **composite design pattern** is meant to achieve two goals:

- To compose nested structures of objects
- To deal with the classes for these objects uniformly

It is a structural design pattern. The following basic design is used by composite design patterns:



In this design, a component interface serves as the supertype for a set of classes. Using **polymorphism**, all implementing classes conform to the same interface, which allows them to be dealt with uniformly.

> Instructor's Note: An abstract superclass can also be used in place of an interface, as both allow for polymorphism. However, this lesson will focus on interfaces.

A **composite class** is present as well. This class is used to aggregate any class that implements the component interface. The composite class allows you to "traverse through" and "potentially manipulate" the component objects that the composite object contains.

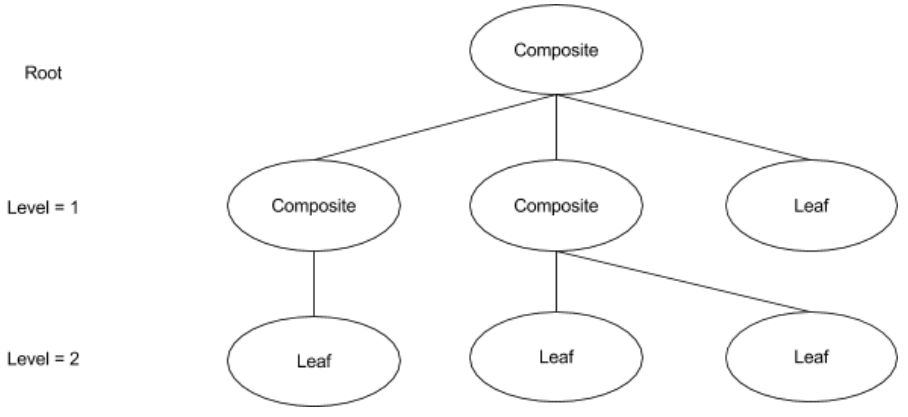
A **leaf class** represents a non-composite type. It is not composed of other components.

The **leaf** class and the **composite** class implement the **Component** interface, unifying them with a single type. This allows us to deal with non-composite and composite objects uniformly – the **Leaf** class and the **Composite** class are now considered subtypes of **Component**.

You may have other composite or leaf classes in practice, but there will only be one overall component interface or abstract superclass.

Another important concept to note is that a composite object can contain other composite objects since the composite class is a subtype of the component. This is known as **recursive composition**. This term is also a synonym for composite design patterns.

This design pattern has a composite class with a cyclical nature, which may make it difficult to visualize. Instead, it is easier to think of composite design patterns as trees:

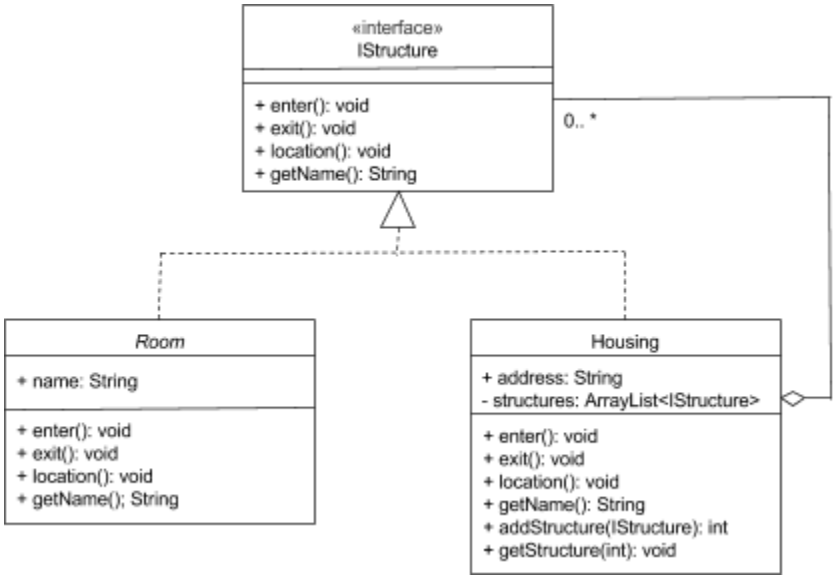


The main composite object, which is made up of other component objects, is at the root level of the tree. At each level, more components can be added below each composite object, like another composite or a leaf. Leaf objects cannot have components added to them. Composites, therefore, have the ability to "grow" a tree, while a leaf ends the tree where it is.

At the beginning of this lesson, we mentioned two issues that composite design patterns are meant to address. This tree helps us understand how those goals are met. Individual types of objects, in particular, composite class objects, can aggregate component classes, which create a tree-like structure. As well, each individual class is a subtype of an interface or superclass that will be able to conform to a set of shared behaviors.

Let us examine the implementation of a composite design pattern by using an example with a building composed of generic housing structures.

Below is a UML diagram of this example:



Expressing this in Java can be broken down into steps.

1. Design the interface that defines the overall type.
2. Implement the composite class.
3. Implement the leaf class

Step 1: Design the interface that defines the overall type

Begin by defining the interface that the composite and leaf classes will implement. This supports polymorphism for your component and leaf classes.

```
public interface IStructure {
   public void enter();
   public void exit();
   public void location();
   public String getName();
}
```

Step 2: Implement the composite class

Implement the interface in the composite class. This will give the **Housing** class its own behavior when the client code uses it.

As a **Housing** object can be composed of other structures, you will need to represent this containment with a suitable collection, and you will need a method for adding more components to your composite object. In the code below, we have a private **ArrayList**, which can be changed through the public **addStructure** method. This allows you to traverse through the components of a composite object, as you can retrieve the components when asked. Note that additional methods can also be included to help with the management of component objects. It is up to you to decide what else to include in your composite class.

```
public class Housing implements IStructure {
   private ArrayList < IStructure > structures;
   private String address;

   public Housing(String address) {
       this.structures = new ArrayList < IStructure > ();
       this.address = address;
   }

   public String getName() {
       return this.address;
   }

   public int addStructure(IStructure component) {
       this.structures.add(component);
       return this.structures.size() - 1;
   }

   public IStructure getStructure(int componentNumber) {
       return this.structures.get(componentNumber);
   }

   public void location() {
       System.out.println("You are currently in " +
this.getName() + ". It has ");
       for (IStructure struct: this.structures)
           System.out.println(struct.getName());
```

```
    }

    public void enter() {
        System.out.println("You have entered the " +
this.getName());
    }
    public void exit() {
        System.out.println("You have left the " +
this.getName());
    }
}
```

Step 3: Implement the leaf class

The final step is to implement the leaf class. The leaf class does not contain any components and does not need to have a collection of components added to it, or any methods to manage such a collection. Instead, the methods of **IStructure** interface simply need to be implemented.

```
public abstract class Room implements IStructure {
    public String name;

    public Room(String name) {
        this.name = name;
    }
    public void enter() {
        System.out.println("You have entered the " +
this.getName());
    }
    public void exit() {
        System.out.println("You have left the " +
this.getName());
    }
    public void location() {
        System.out.println("You are currently in the" +
this.getName());
    }
    public String getName() {
        return this.name;
```

```
    }
}
```

In order to implement the building with the composite design pattern, we construct our building structure by structure.

In this example, the building has one floor with a common room and two washrooms. See the code below.

```
public class Main {
    public static void main(String[] args) {
        Housing building = new Housing("123 Street");
        Housing floor1 = new Housing("123 Street - First
Floor");
        int firstFloor = building.addStructure(floor1);

        Room washroom1m = new Room("1F Men's Washroom");
        Room washroom1w = new Room("1F Women's Washroom");
        Room common1 = new Room("1F Common Area");

        int firstMens = floor1.addStructure(washroom1m);
        int firstWomans = floor1.addStructure(washroom1w);
        int firstCommon = floor1.addStructure(common1);

        building.enter();
        Housing currentFloor = (Housing)
building.getStructure(firstFloor);
        currentFloor.enter(); // Walk into the first floor
        Room currentRoom = (Room)
currentFloor.getStructure(firstMens);
        currentRoom.enter(); // Walk into the men's room
        currentRoom = (Room)
currentFloor.getStructure(firstCommon);
        currentRoom.enter(); // Walk into the common area
        building.exit();
    }
}
```

Composite objects can be built quickly and easily. Each component can be expected to have the same set of behaviors without needing to do any type checking.

In summary, a composite design pattern allows you to build a tree-like structure of objects and to treat individual types of those objects uniformly. This is achieved by:

- Enforcing polymorphism across each class through implementing an interface (or inheriting from a superclass).
- Using a technique called recursive composition that allows objects to be composed of other objects that are of a common type.

Composite design patterns apply the design principles of decomposition and generalization. They break a whole into parts but the whole and parts both conform to a common type. Complex structures can be built using composite objects and leaf objects that belong to a unified component type. This makes it easier to understand and manipulate this structure.

Proxy Pattern

A **proxy design pattern** allows a proxy class to represent a real "subject" class. It is a structural design pattern. A **proxy** is something that acts as a simplified, or lightweight version, of the original object. A proxy object can perform the same tasks as an original object but may delegate requests to the original object to achieve them.

In this design pattern, the proxy class wraps the real subject class. This means that a reference to an instance of the real subject class is hidden in the proxy class. The real object is usually a part of the software system that contains sensitive information, or that would be resource-intensive to instantiate. As the proxy class is a wrapper, client classes interact with it instead of the real subject class.

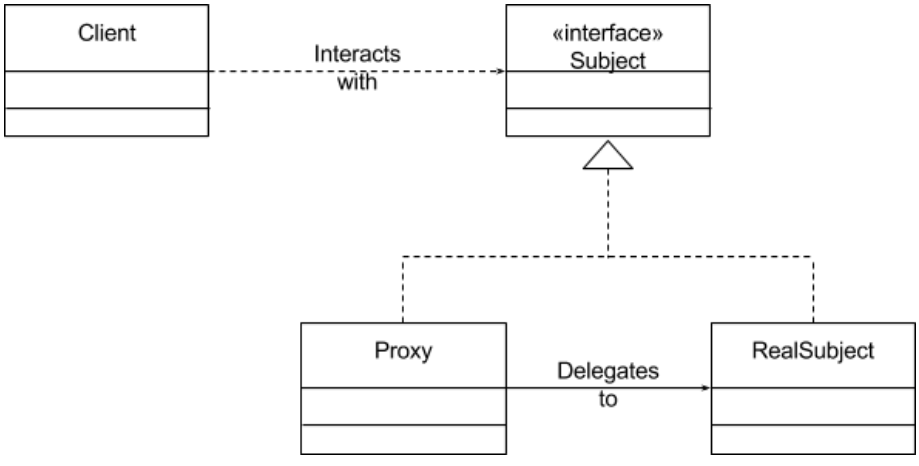
The three most common scenarios where proxy classes are used are:

- To act as a **virtual proxy**. This is when a proxy class is used in place of a real subject class that is resource-intensive to instantiate. This is commonly used on images in web pages or graphic editors, as a high-definition image may be extremely large to load.
- To act as a **protection proxy**. This is when a proxy class is used to control access to the real subject class.

For example, a system that is used by both students and instructors might limit access based on roles.

- To act as a **remote proxy**. This is when a proxy class is local, and the real subject class exists remotely. Google docs make use of this, where web browsers have all the objects it needs locally, which also exist on a Google server somewhere else.

The UML diagram below demonstrates the proxy design pattern.



The proxy class wraps and may delegate, or redirect, calls upon it to the real subject class. However, not all classes are delegated, as the proxy class can handle some of its lighter responsibilities. Only substantive requests are sent to the real subject class. Because of this, the proxy class must offer the same methods. Having both these classes implement a common subject interface allows for polymorphism.

Note that the proxy and real subject classes are subtypes of the subject. A client class can therefore interact with the proxy, which will have the same expected interface as the real subject.

Below is an example of a UML diagram for an online retail store with global distribution and warehousing. In this scenario, you need to determine which warehouse to send orders to. A system that routes orders for fulfillment to an appropriate warehouse will prevent your warehouses from receiving orders that they cannot fulfill. A proxy may protect your real subject, the warehouses, from receiving orders if the warehouses do not have enough stock to fulfill an order.

The Order Fulfillment class is the proxy in this example. Clients interact with the system by using the interface.

Implementation of this pattern in Java can be broken down into steps.

1. Design the subject interface.
2. Implement the real subject class.
3. Implement the proxy class.
4.

Step 1: Design the subject interface

First, create the interface that client software will use to interact with your system.

```
public interface IOrder {
    public void fulfillOrder(Order);
}
```

Step 2: Implement the real subject class

Next, implement the real subject class. In this case, the Warehouse class knows how to process an order for fulfillment, and it knows how to report the current stock of items. It does not need to check if it has enough stock to fulfill an order, as an order should only be sent to a warehouse if it can be fulfilled.

```
public class Warehouse implements IOrder {
   private Hashtable<String, Integer> stock;
```

```
    private String address;

    /* Constructors and other attributes would go here */
    ...

    public void fulfillOrder(Order order) {
        for (Item item : order.itemList)
            this.stock.replace(item.sku, stock.get(item)-1);

        /* Process the order for shipment and delivery */
        ...
    }

    public int currentInventory(Item item) {
        if (stock.containsKey(item.sku))
            return stock.get(item.sku).intValue();
        return 0;
    }
}
```

Step 3: Implement the proxy class

The final step requires implementing the proxy class. In this case, the Order Fulfillment class checks warehouse inventory and ensures that an order can be completed before sending requests to the warehouse. To do this, it asks each warehouse if it has enough stock of a particular item. If a warehouse does, then the item gets added to a new Order object that will be sent to the Warehouse. The Order Fulfillment class also lets you separate order validation from order fulfillment by separate them into two pieces. This improves the overall rate of processing an order, as the warehouse does not have to worry about the validation process or about re-routing an order if it cannot be fulfilled.

The Order Fulfillment class can be improved with other functionalities, such as prioritizing sending orders to warehouses based on proximity to the customer.

```
public class OrderFulfillment implements IOrder {
    private List<Warehouse> warehouses;

    /* Constructors and other attributes would go here */
```

```
    public void fulfillOrder(Order order) {
        /* For each item in a customer order, check each
warehouse to see if it is in stock.
        If it is then create a new Order for that warehouse.
Else check the next warehouse.
        Send all the Orders to the warehouse(s) after you finish
iterating over all the items in the original Order. */

        for (Item item: order.itemList) {
            for (Warehouse warehouse: warehouses) {
                ...
            }
        }
        return;
    }
}
```

Key responsibilities of the proxy class there include protecting the real subject class by checking the client's request and controlling access to the real subject class and acting as a wrapper class for the real subject class.

In summary, a proxy design pattern allows you to defer creating resource-intensive objects until needed, control access to specific objects, or when you need something to act as a local representation of a remote system. They made systems more secure, and less resource-intensive. The main features of a proxy design pattern are:

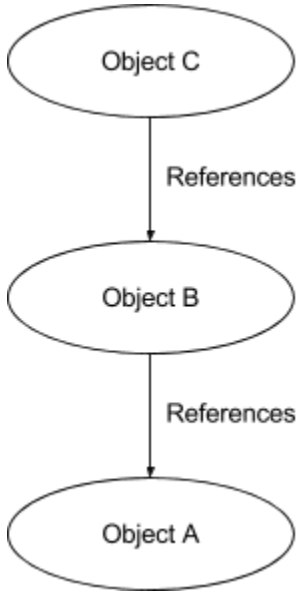
- To use the proxy class to wrap the real subject class.
- To have a polymorphic design so that the client class can expect the same interface for the proxy and real subject classes.
- To use a lightweight proxy in place of a resource-intensive object until it is actually needed.
- To implement some form of intelligent verification of requests from client code in order to determine if, how, and to whom the requests should be forwarded to.
- To present a local representation of a system that is not in the same physical or virtual space.

Decorator Pattern

It is beneficial for software to have flexible combinations of overall behaviors. However, changes to classes cannot be made while a program is running, as the behavior of an object is defined by its class, and only occurs at compile time. A new class would need to be created in order to achieve a new combination of behaviors while a program is running. Lots of new combinations, however, would lead to lots of classes, which is undesirable.

A **decorator design pattern** allows additional behaviors or responsibilities to be dynamically attached to an object, through the use of aggregation to combine behaviors at run time. It is a structural design pattern.

As explained in the first course of this specialization, aggregation is a design principle used to represent a "has-a" or "weak containment" relationship between two objects. This can be used to build a stack of objects, also known as an "aggregation stack", where each level of the stack contains an object that knows about its own behavior and augments the one underneath it.



In this visual representation, object A is a **base object**, because it does not contain another object, and has its own set of behaviors. **Object B** aggregates **Object A**, allowing **Object B** to "augment" the behaviors of **Object A**. Objects can

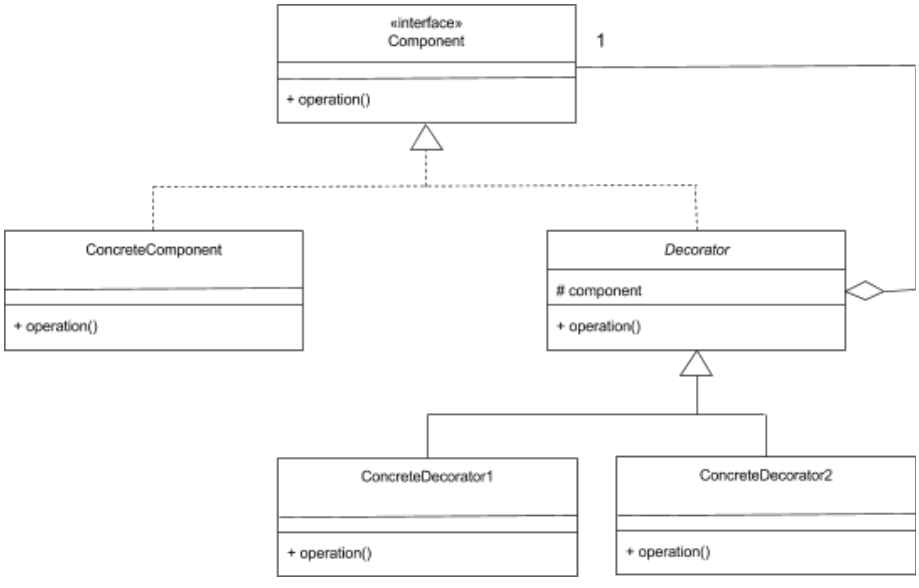
continue to be added to the stack. So, in the above diagram, **Object C** aggregates to **Object B** and augments the behaviors of **Object B**.

In an aggregation stack, the aggregation relationship is always one-to-one in the decorator design pattern. This allows the stack to build up, so one object is on top of another.

An overall combination of behaviors for stacked objects can be achieved if you call upon the top element, which is **Object C** in this example. This would start a series of calls: **Object C** would call upon **Object B**, **Object B** would call upon **Object A**. **Object A** would then respond with its behaviour, then **Object B** would add its increment of behavior, followed by **Object C**.

This design pattern makes use of interfaces and inheritance so that the classes conform to a common type whose instances can be stacked in a compatible way. This builds a coherent combination of behavior overall.

Below is a UML diagram for a decorator design pattern.



It consists of a **Component** interface that defines the common type for all the classes. A client class expects the same interface across all the component classes. A concrete component class implements the component interface and can be instantiated. An instance of this class can be used as a base object in the stack.

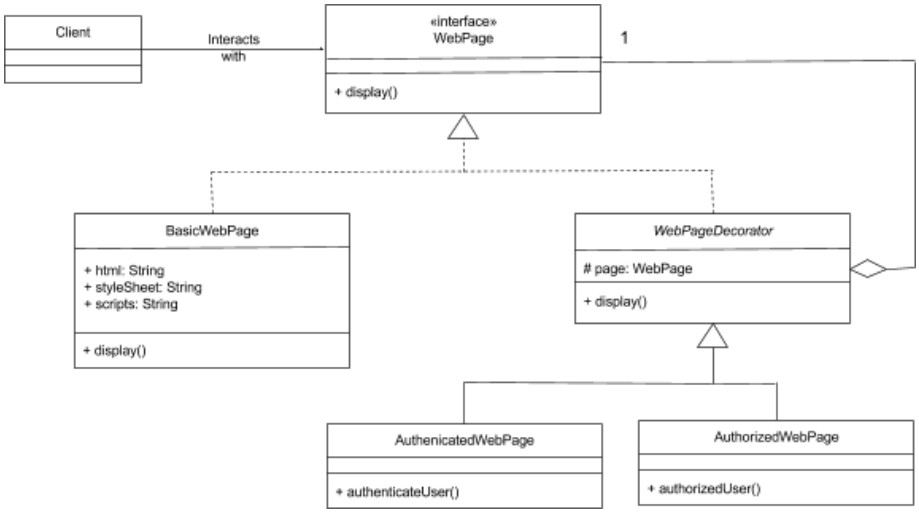
As indicated by the italics, ***Decorator*** is an abstract class. Similar to the concrete component class, it implements the component interface. However, the Decorator class aggregates other types of components, which will allow us to "stack" components on top of each other. It also serves as the abstract superclass of concrete decorator classes that will each provide an increment of behavior.

A stack of components can be built starting with an instance of the concrete component class and continuing with instances of subclasses of the decorator abstract class.

Decorator design patterns allow objects to dynamically add behaviors to others, and it reduces the number of classes needed to offer a combination of behaviors.

Let us examine the decorator design pattern using an example of a web page. A web page might display complex behavior, such as only allowing access to authorized users, or splitting search results across multiple pages. Every possible combination of web page permission, pagination, or caching behaviors does not need to be written as different web page types. Decorator design patterns can be used to create one class for each type of behavior as well as build specific combinations for a web page that you want at run time.

When expressed as a UML diagram, the example outlined above looks as below:



Note that for simplicity, the **BasicWebPage's** HTML, stylesheet, and scripts are represented as **Strings**. As well, any

number of additional behaviors can be used to augment the basic web page, but this example is limited to the behaviors of adding user authorization and authentication to make sure the user is who they claim to be.

Implementation of this design pattern with Java can be broken down into steps.

1. Design the component interface.
2. Implement the interface with your base concrete component class.
3. Implement the interface with your abstract decorator class.
4. Inherit from the abstract decorator and implement the component interface with concrete decorator classes.

Step 1: Design the component interface

First, define the interface that the rest of the classes in the design pattern will be subtypes of. This interface will define the common behaviors that your basic web page and decorators will have.

```
public interface WebPage {
     public void display();
}
```

Step 2: Implement the interface with your base concrete component class

Next, implement the base concrete component class with the interface. The concrete component class will be the base building block for all web pages objects during run time.

The concrete component class will implement how it displays itself by using standard HTML markup and page element styling defined in the cascading style sheet. This example web page will also run some basic JavaScript.

```
public class BasicWebPage implements WebPage {
     public String html = ...;
     public String styleSheet = ...;
     public String script = ...;
```

```
     public void display() {
    /* Renders the HTML to the stylesheet, and run any embedded
scripts */
 }
```

Step 3: Implement the interface with your abstract decorator class

Next, the abstract decorator class should be implemented with the interface. This implementation is important.

```
public abstract class WebPageDecorator implements WebPage {
   protected WebPage page;

   public WebPageDecorator(WebPage webpage) {
       this.page = webpage;
   }

   public void display() {
       this.page.display();
   }
}
```

The web page decorator only has one instance of the web page. This allows decorators to be stacked on top of the basic web page, and on top of each other. Each type of web page is responsible for its own behavior and will recursively invoke the next web page on the stack to execute its behavior.

The constructor allows different subtypes of web pages to be linked together in a stack. All that must be done is to indicate what instance of web page subtype should be stacked upon.

In stacking, the order in which you build the stack matters. However, the basic web page must be the first one in the stack. The rest of the ordering will depend on the design of your system and which augmenting behaviors you want to be executed first.

The abstract decorator simply delegates the display behavior to the web page object that it aggregates. This allows you to combine the display behavior down the stack of web pages.

Step 4: Inherit from the abstract decorator and implement the component interface with concrete decorator classes

The final step is to inherit from the abstract decorator and implement the component interface with the concrete decorator classes.

```
public class AuthorizedWebPage extends WebPageDecorator {
   public AuthorizedWebPage(WebPage decoratedPage) {
super(decoratedPage); }

   public void authorizedUser() {
       System.out.println("Authorizing user");
   }

   public display() {
       super.display();
       this.authorizedUser();
   }
}

public class AuthenticatedWebPage extends WebPageDecorator {
   public AuthenticatedWebPage(WebPage decoratedPage) {
super(decoratedPage); }

   public void authenticateUser() {
       System.out.println("Authenticating user");
   }

   public display() {
       super.display();
       this.authenticateUser();
   }
}
```

In this example, the constructors use the abstract superclass's constructor, as it allows decorators to be stacked together. The abstract web page decorator class handles the aggregation of the concrete decorator classes. Further, each decorator has its

own responsibilities. These are implemented within the appropriate classes so that they can be invoked.

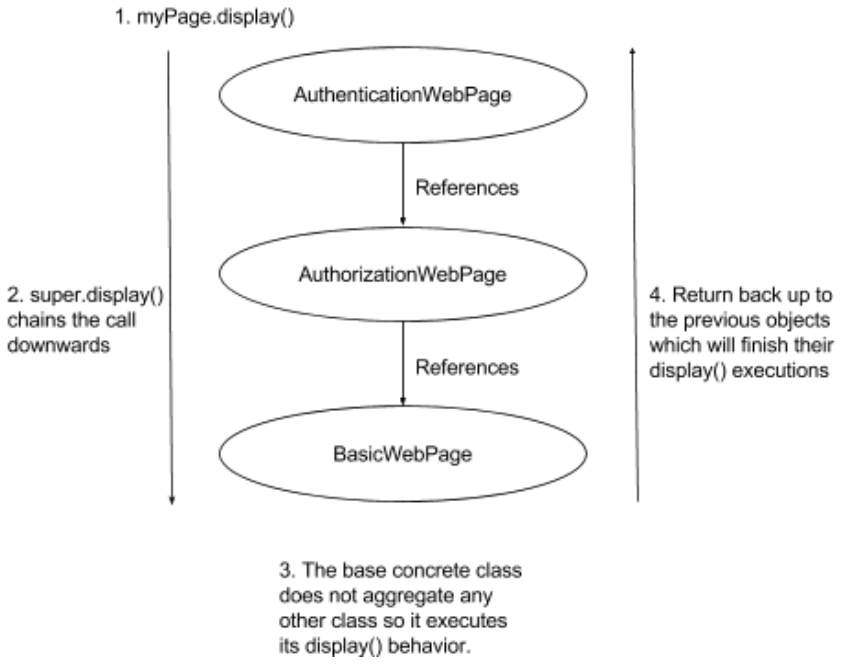
Notice that to recursively call the display behavior, the concrete decorators invoke the superclass's display method. Since the abstract decorator superclass facilitates the aggregation of various types of web design, the call to **super.display()** will cause the next web page in the stack to execute its version of the display until you get to the basic web page. The recursive call will end here because the basic web page is the concrete component, which does not aggregate any other types of web pages. This links the calls of display all the way to the bottom and bubbles the execution back up. The basic web page must be built before behaviors can be added to it.

Let us now examine the decorator pattern in action.

```
public class Program {
	public static void main(String args[]) {
		WebPage myPage = new BasicWebPage();
		 myPage = new AuthorizedWebPage(myPage);
		 myPage = new AuthenticatedWebPage(myPage);
		 myPage.display();
	}
}
```

As explained above, the basic web page is built first. The authorization behavior is added next. The web page is completed by adding the authentication behavior last.

When the display method is called, it will link the method calls down to the basic web page. The basic web page will display itself and then the display call will move back up the links to the authorized behavior first and the authentication behavior last. As an aggregation stack diagram, this would look as follows:



Any decorator could be added to the basic web page to create a different combined behavior. The basic web page's behavior can be dynamically built up in this way.

In summary, the main features of a decorate design pattern are:

- We can add, in effect, any number of behaviors dynamically to an object at runtime by using aggregation as a substitute for pure inheritance
- Polymorphism is achieved by implementing a single interface.
- Aggregation lets us create a stack of objects.
- Each decorator object in the stack is aggregated in a one-to-one relationship with the object below it in the stack.
- By combining aggregation and polymorphism, we can recursively invoke the same behavior down the stack and have the behavior execute upwards from the concrete component object.

The use of design patterns like the decorator pattern helps you to create complex software, without the complex overhead.

Module 2: Behavioural Design Pattern

Upon completion of this module, you will be able to:

1. Explain what a behavioural design pattern is.
2. Describe the template method pattern.
3. Describe the chain of responsibility pattern.
4. Describe the state pattern.
5. Describe the command pattern.
6. Describe the observer pattern.

This module will describe certain **behavioral design patterns**. These are patterns that focus on ways that individual objects collaborate to achieve a common goal. This is an important aspect to consider in software design – objects are only pieces of a larger solution. For each to work effectively, it must have a set purpose. Let us examine some behavioral design patterns.

Template Method Pattern

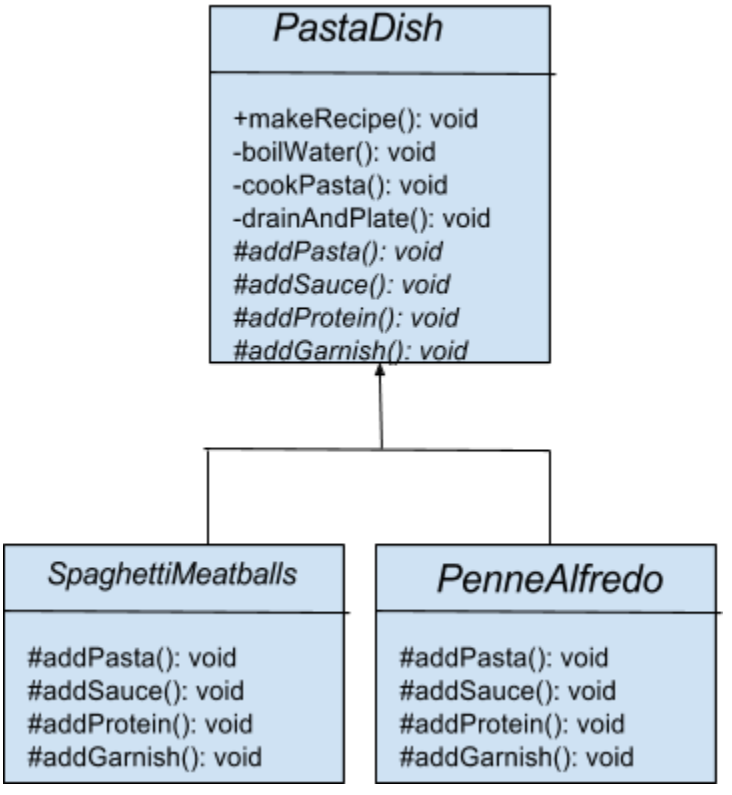
The **template method pattern** defines an algorithm's steps generally, by deferring the implementation of some steps to subclasses. In other words, it is concerned with the assignment of responsibilities.

The template method is best used when you can generalize between two classes into a new superclass. Think of it as another technique to use when you notice you have two separate classes with very similar functionality and order of operations. You can choose to use a template method, so changes to these algorithms only need to be applied in one place instead of two. The template method would be within the superclass, and would therefore be inherited by the subclasses. Differences in algorithms would be done through calls to abstract methods whose implementations are provided by the subclass. After using generalization, objects can be more effectively reused. Inheritance allows functionality to be shared between classes and enables clearer and more self-explanatory code.

Let us examine a UML diagram for an example of a template method pattern. In this example, imagine you are an executive chef who oversees a large chain of restaurants that serves pasta. In order to ensure that dishes at all the restaurant locations are consistent, instructions are provided for making popular dishes. The two most popular dishes are spaghetti with tomato sauce and meatballs, and penne noodles with alfredo sauce and chicken.

Both dishes require that you boil water, cook the pasta, add the sauce, add the protein, and garnish the plate. Some of these steps have different implementations depending on which dish is being prepared – each dish has a different sauce, protein, and garnish. Other steps, however, would be the same, such as boiling the water.

This situation could be modeled with a **PastaDish** class that has a method that makes the recipe for each subclass, spaghetti with meatballs or penne alfredo. The method knows the general set of steps to make either dish - from boiling water to adding the garnish. Steps that are special to a dish, like the sauce to add, are implemented in the subclass. The UML diagram below illustrates this example.



The UML for the template method pattern is centered upon the relationship between the superclass and its subclasses. The superclass must contain abstract methods that are implemented by the subclasses.

Let us now examine how this might be implemented as Java code.

First, let us examine a look at the code for the **PastaDish** superclass.

```
public abstract class PastaDish {
	final void makeRecipe() {
	boilWater();
```

```
        addPasta();
                cookPasta();
        drainAndPlate();
        addSauce();
                addProtein();
                addGarnish();
        }
        abstract void addPasta();
        abstract void addSauce();
        abstract void addProtein();
        abstract void addGarnish();

        private void boilWater() {
        System.out.println("Boiling water");
        }
        ...
}
```

The **PastaDish** is an abstract class – a generic pasta dish cannot be made. The **makeRecipe()** method is our template method. This method is marked as final; in Java, this keyword means that the method declared cannot be overridden by subclasses. In this example, it means that neither specific dish subclass can have its own version of **makeRecipe**, and ensures consistency in the steps of making the dishes while also reducing redundant code.

From within the **makeRecipe**, the other methods are called. Methods that are the same for any subclass are consolidated in the superclass. Specialized methods, however, are left up to subclasses to provide them.

Let us now examine the **SpaghettiMeatballs** and **PenneAlfredo** subclasses. Both subclasses extend the **PastaDish** class and implement the abstract methods **addPasta**, **addProtein**, **addSauce**, and **addGarnish**. These methods are called in the **makeRecipe** template method found in the **PastaDish** superclass. The template method is inherited in a subclass and behaves as expected to make a dish with the right ingredients.

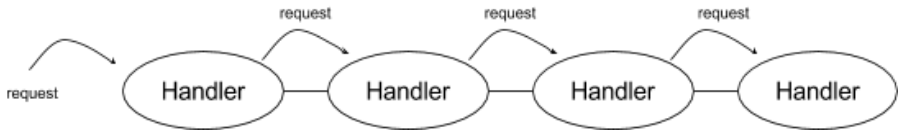
```
public class SpaghettiMeatballs extends PastaDish {
  public void addPasta() {
      System.out.println("Add spaghetti");
  }
  public void addProtein() {
      System.out.println("Add meatballs");
  }
  public void addSauce() {
      System.out.println("Add tomato sauce");
  }
  public void addGarnish() {
      System.out.println("Add Parmesan cheese");
  }
}

public class PenneAlfredo extends PastaDish {
  public void addPasta() {
      System.out.println("Add penne");
  }
  public void addProtein() {
      System.out.println("Add chicken");
  }
  public void addSauce() {
      System.out.println("Add Alfredo sauce");
  }
  public void addGarnish() {
      System.out.println("Add parsley");
  }
}
```

Chain of Responsibility Pattern

The **chain of responsibility design pattern** is a chain of objects that are responsible for handling requests. In software design, these objects are handler objects that are linked together.

When a client object sends a request, the first handler in the chain will try to process it. If the handler can process the request, then the request ends with this handler. However, if the handler cannot handle the request, then the request is sent

to the next handler in the chain. This process will continue until a handler can process the request.



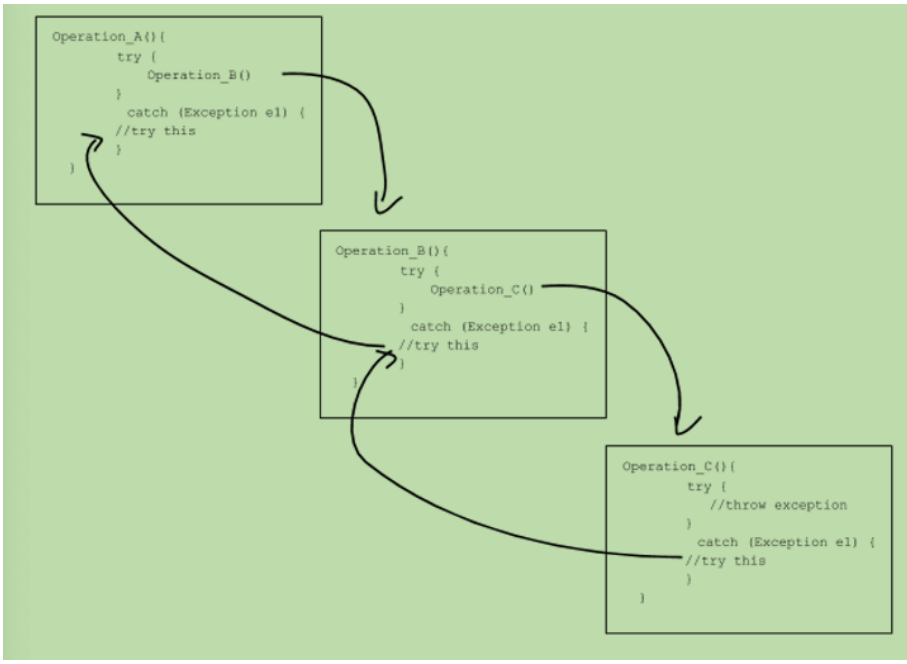
If the entire chain is unable to handle the request, then the request is not satisfied.

The chain of responsibility is an important pattern in software. This pattern avoids coupling the sender to the receiver by giving more than one object a chance to handle the request. Whoever sends the request does not need to to worry about who will handle the request, so the sender and the receiver are decoupled from each other. The chain of responsibility also helps deal with situations in which streams of different requests need to be handled.

A good real-life metaphor for a chain of responsibility pattern might be considering fixing a chair. Imagine you come across a screw, and you need a tool to tighten it. If you are not sure which type of tool to use, you might try each tool and try it one at a time on the screw until you could determine which one of them works.

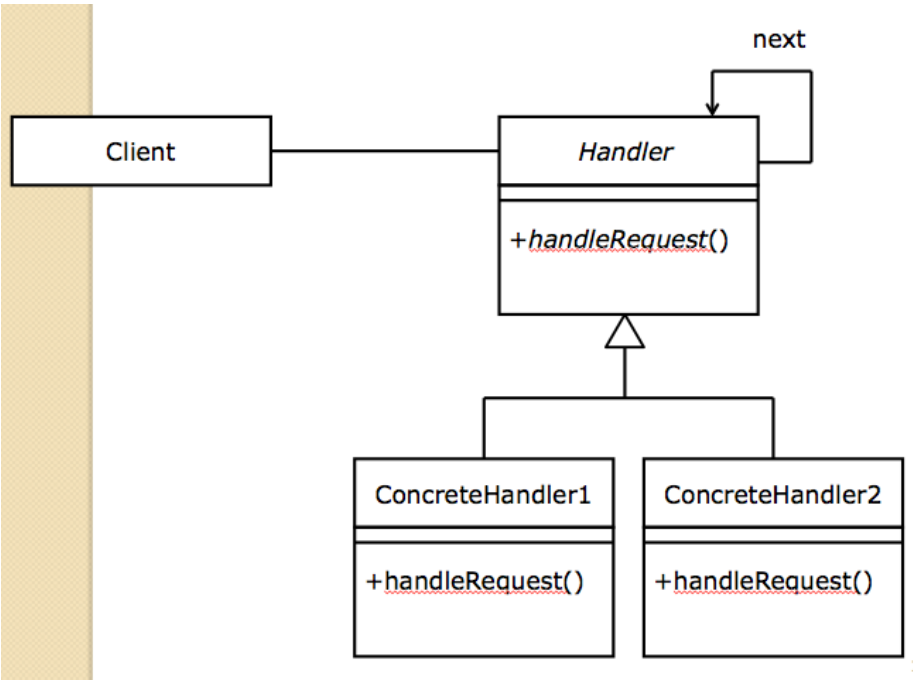
DID YOU KNOW?

This pattern is similar to exception handling in Java. In exception handling, a series of try/catch blocks are written in your software to ensure that exceptions are dealt with properly. When an exception occurs, one of the catch blocks is expected to handle it.



The chain of responsibility can be used for many different purposes. For example, it is commonly used for filtering objects, such as putting certain emails into a spam folder.

The UML diagram for the chain of responsibility typically looks as below. Objects on the chain are handlers that implement a common method **handleRequest()**, which is declared in the abstract superclass Handler, as indicated by the italics. Handle objects are connected from one to the next in the chain. Subclasses of Handler handle requests in their own way.



The chain of responsibility pattern does not come without potential issues. For example, what if there's a mistake in the second filter and its rule doesn't match but forgets to pass the request on to the next filter. In this case, the handling ends prematurely. To circumvent this problem, there needs to be an algorithm that ensures each filter class handles requests in a similar fashion.

Each filter needs to go through the following steps:

1. Check if the rule matches.
2. If it matches, do something specific.
3. If it doesn't match, call the next filter in the list.

It may be helpful to use the template pattern learned in a previous lesson to ensure that each class will handle a request in a similar way, following the above steps.

DID YOU KNOW?

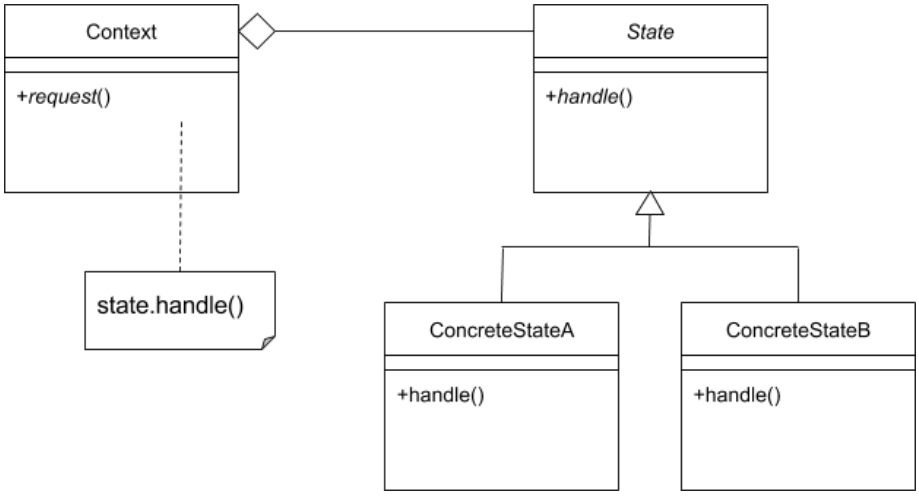
Design patterns are not without problems! In fact, design patterns often have their own internal problems. It is very common in the industry to combine design patterns to fix those problems.

State Pattern

Objects in your code are aware of their current state. They can choose an appropriate behavior based on their current state. When their current state changes, this behavior can be altered - this is the **state design pattern**.

This pattern should be primarily used when you need to change the behavior of an object based upon changes to its internal state or the state it is in at run-time. This pattern can also be used to simplify methods with long conditionals that depend on the object's state.

The general structure of a state pattern can be represented as below in a UML diagram:



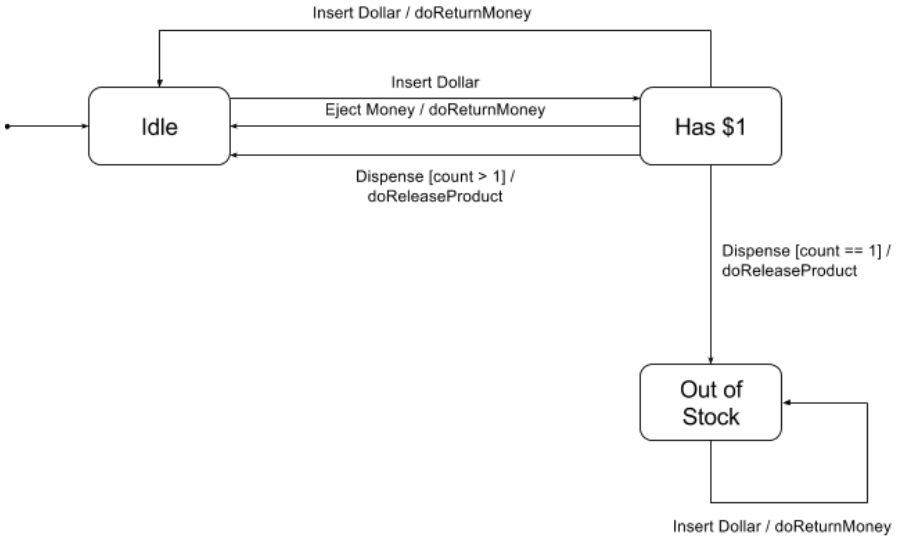
The state design pattern might be best explained through an example. A vending machine has several states, and specific actions based on those states. For example, imagine you want to purchase a chocolate bar from a vending machine that costs one dollar.

If you insert the dollar into the machine, several things might happen. You could make your selection, the machine would dispense a chocolate bar, and you would collect the bar and leave. You could decide you no longer want the chocolate bar, press the eject money button, and the machine would return the dollar. You could make your selection, the machine could be out of chocolate bars, and could notify you of this.

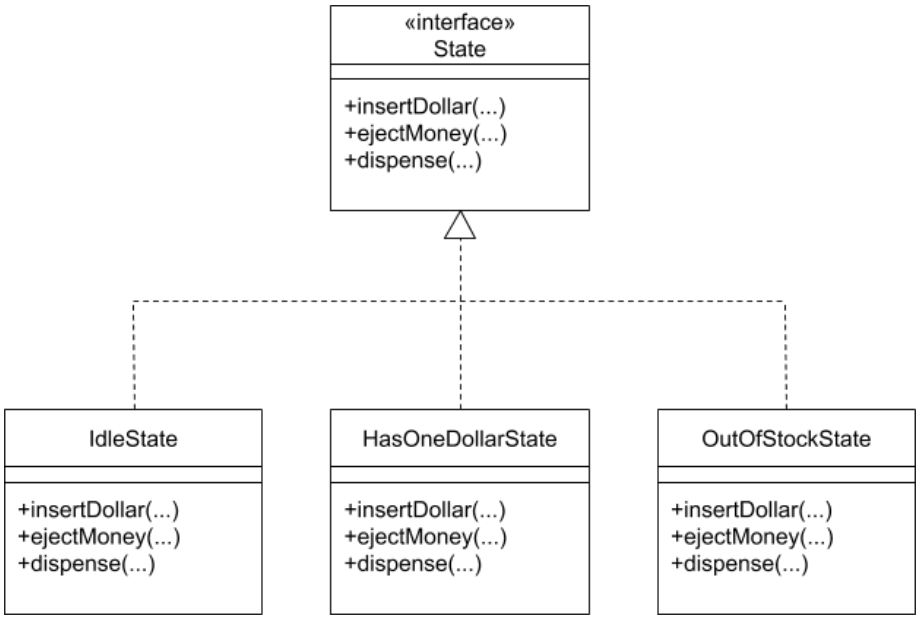
There are three different states that the vending machine can be in within this scenario. Before being approached, the machine is in an idle state. When the dollar is inserted, the state of the machine changes and could either dispense a product or return the money upon receiving an eject money request. The third state the machine could be in is if the machine is out of stock.

Each of these states could be represented as state objects in the code. However, in this scenario, state objects are passive and don't have much responsibility themselves. Using the state design pattern rather than a state diagram, this can be represented in a different way.

Below is the UML state diagram:



But, applying the state design pattern as a UML diagram presented at the beginning of this lesson, the following diagram can be generated:



Let us examine the state interface as code. State classes must implement the methods in this interface to respond to each trigger.

```
public interface State {
       public void insertDollar( VendingMachine vendingMachine
);
       public void ejectMoney( VendingMachine vendingMachine );
       public void dispense( VendingMachine vendingMachine );
}
```

Now, let us examine the Idle State class:

```
public class IdleState implements State {

   public void insertDollar( VendingMachine vendingMachine ) {
       System.out.println( "dollar inserted" );

       vendingMachine.setState(
           vendingMachine.getHasOneDollarState()
       );
   }

   public void ejectMoney( VendingMachine vendingMachine ) {
       System.out.println( "no money to return" );
```

```
    }

    public void dispense( VendingMachine vendingMachine ) {
        System.out.println( "payment required" );
    }
}
```

The **IdleState** class now implements the **State** interface. When a dollar is inserted, the **insertDollar()** method is called, which then calls a **setState()** method upon the **vendingMachine** object. This changes the current state of the machine to the **HasOneDollar** state.

```
public class HasOneDollarState implements State {

    public void insertDollar( VendingMachine vendingMachine ) {
        System.out.println( "already have one dollar" );
    }

    public void ejectMoney( VendingMachine vendingMachine ) {
        System.out.println( "returning money" );

        vendingMachine.doReturnMoney();
        vendingMachine.setState(
            vendingMachine.getIdleState()
        );
    }

        // class HasOneDollarState continued below
        ...
```

In the **HasOneDollarState** class, when the **ejectMoney** method is called, the money is returned and **setState()** is called on the vending machine to change the state to the Idle state.

```
...
// class HasOneDollarState continued

        public void dispense( VendingMachine vendingMachine ) {
        System.out.println( "releasing product" );
```

```
        if (vendingMachine.getCount() > 1) {
                        vendingMachine.doReleaseProduct();
                        vendingMachine.setState(
                vendingMachine.getIdleState());
                } else {
                        vendingMachine.doReleaseProduct();
                        vendingMachine.setState(
                vendingMachine.getOutOfStockState());
                }
        }
}
```

If the dispense method is called in the **HasOneDollarState** class, then the product is released. If the stock remains after this, the vending machine's state sets to either the **Idle** state or the **OutOfStock** state.

```
public class PopMachine {

        private State idleState;
        private State hasOneDollarState;
        private State outOfStockState;

        private State currentState;
        private int count;

        public PopMachine( int count ) {
        // make the needed states
        idleState = new IdleState();
        hasOneDollarState = new HasOneDollarState();
        outOfStockState = new OutOfStockState();

        if (count > 0) {
                currentState = idleState;
                this.count = count;
        } else {
                currentState = outOfStockState;
                this.count = 0;
        }
 }
```

The **VendingMachine** class constructor will instantiate each of the state classes, and the current state will refer to one of these state objects.

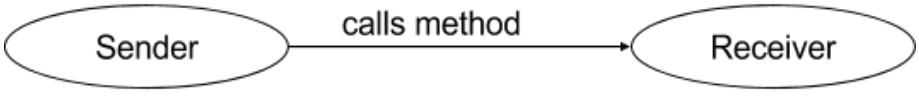
```
public void insertDollar() {
	currentState.insertDollar( this );
}

public void ejectMoney() {
	currentState.ejectMoney( this );
}

public void dispense() {
	currentState.dispense( this );
}
```

The **VendingMachine** class would have methods to handle the triggers, but it delegates the handling to whatever is the current state object.

Using the State design pattern, long conditionals are not necessary for the methods and creates much cleaner code.

Command Pattern

The **command pattern** encapsulates a request as an object of its own. In general, when an object makes a request for a second object to do an action, the first object would call a method of the second object and the second object would complete the task. There is direct communication between the sender and receiver object.



The command pattern creates a command object between the sender and receiver. This way, the sender does not have to know about the receiver and the methods to call.

In a command pattern, a sender object can create a command object. However, an invoker is required to make the command object do what it's supposed to do and get the specific receiver object to complete the task. An **invoker** is therefore an object that invokes the command objects to complete whatever task it is supposed to do. A command manager can also be used that basically keeps track of the commands, manipulates them, and invokes them.

Purposes of the Command Pattern

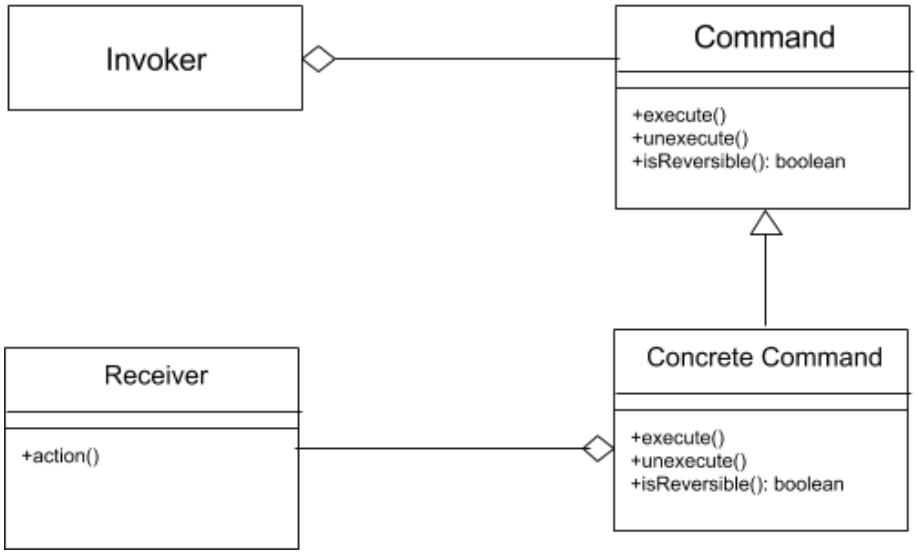
There are many different purposes for using the command pattern.

One is to store and schedule different requests. If requests are turned into command objects in your software, then they can be stored into lists and manipulated before they are completed. They can also be placed onto a queue so that different commands can be scheduled to be completed at different times. For example, the command pattern can be used to have an alarm ring in calendar software. A command object could be created to ring the alarm, and this command could be placed into a queue so that it is completed when the event is scheduled to occur.

Another purpose for the command pattern is to allow commands to be undone or redone. For example, edits can be undone or redone in a document. Imagine that the software has two lists: a history list, which holds all the commands that have been executed, and a redo list, which will be used to put commands that have been undone. Each time a command is requested, a command object is created and executed. When the command is completed, it goes to the history list. If you undo a command, then the software would go to the history list and ask the most recent command executed to undo itself, and put it on the redo list. Alternately, if a user needs to redo, the software would take the most recent command undone in the redo list, and move it onto the history list again. The redo list will be emptied every time a command is executed because usually, you can't redo a previous edit after a new edit has been made.

The command pattern lets you do things to requests that you wouldn't be able to do if they were simple method calls from one object to the other. Commands can also be stored in a log list, so if the software crashes unexpectedly, users can redo all the recent commands.

Below is a UML diagram of the basic structure of a command pattern:



In this diagram, there is a command superclass, and all commands are instances of subclasses of this command superclass. The superclass defines the common behaviors of your commands. Each command will have the methods **execute()**, **unexecute()**, and **isReversible()**.

- The **execute()** method will do the work the command is supposed to do.
- The **unexecute()** method will do the work of undoing the command.
- The **isReversible()** method will determine if the command is reversible, returning true if the command can be undone.

Some commands may not be able to be undone, such as a save command.

The concrete command classes call on specific receiver classes to deal with the work of completing the command.

Let us examine how a command object should be written in Java code, with the example of “pasting text”.

```
public class PasteCommand extends Command {
       private Document document;
       private int position;
       private String text;
 ...
       public PasteCommand( Document document,
int position, String text ){
              this.document = document;
              this.position = position;
              this.text = =text;
 }
       public void execute() {
              document.insertText ( position, text );
       }
       public void unexecute() {
              document.deleteText ( position, text.length());
       }
       public boolean isReversible() {
              return true;
       }
}
```

The paste command extends the **Command** superclass and keeps track of where the text will be inserted as well as what will be inserted. Command objects must be able to keep track of a lot of details on the current state of the document in order for commands to be reversible.

When the **execute()** and **unexecute()** methods are called, this is where the command object actually calls on the receiver to complete the work.

Let us examine the code for the invoker.

```
CommandManager commandManager = CommandManager.getInstance();
Command command = new PasteCommand(aDocument, aPosition,
AText);
commandManager.invokeCommand (command);
```

The invoker references the command manager, which is the object that manages the history and redo lists. Then, the

invoker creates the command object with the information needed to complete the command. Finally, it calls the command manager to execute the command.

Benefits of the Command Pattern

The command pattern allows commands to be manipulated as objects. Functionalities can be added to the command objects, such as putting them into queues and adding undo/redo functions.

Command patterns also decouple the objects of your software program, as classes do not need to know about other objects in the software system – the command object deals with the work by invoking receiver objects, and the original object does not need to know what other objects are involved in the request.

The command pattern also allows logic to be pulled from user interfaces. User interface classes should only be dealing with issues like getting information to and from the user, and application logic should not be in user interface classes. The command pattern creates a layer where command objects go, so that every time a button is clicked on the interface, a command object is created. This is where application logic will sit instead. The command objects are independent of the user interface so that adding changes like new buttons to the interface is easier and faster.

Each and every service in a system can be an object of its own, allowing for more flexible functionality. This pattern can be a great asset to making versatile and easy-to-maintain software programs.

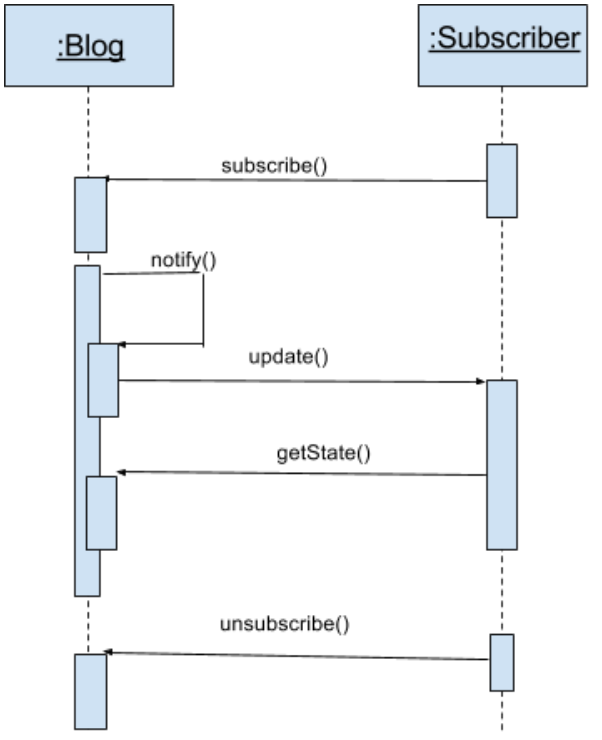
Observer Pattern

The **observer design pattern** is a pattern where a **subject** keeps a list of **observers**. Observers rely on the subject to inform them of changes to the state of the subject.

In an observer design pattern, there is generally a **Subject** superclass, which would have an attribute to keep track of all the observers. There is also an **Observer** interface with a method so that an observer can be notified of state changes to the subject. The Subject superclass may also have subclasses that implement the Observer interface. These elements create the relationship between the subject and observer.

Let us explore this pattern through an example. Imagine you have subscribed to a blog and would like to receive notifications of any changes made to the blog.

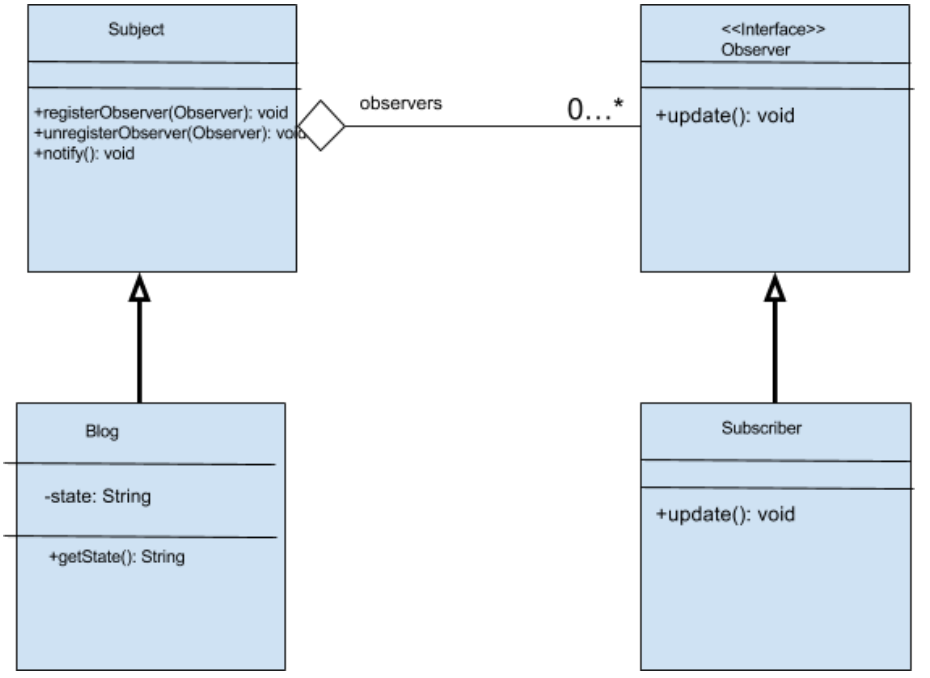
The sequence diagram for this example might look as below:



A sequence diagram for observe patterns will have two major roles: the subject (the blog) and the observer (a subscriber). In order to form the subject and observer relationship, a subscriber must subscribe to the blog. The blog then needs to be able to notify subscribers of a change. The notify function keeps subscribers consistent and is only called when a change has been made to the blog. If a change is made, the blog will make an update call to update subscribers. Subscribers can get the state of the blog through a **getState()** call. It is up to the blog to ensure its subscribers get the latest information.

To unsubscribe from the blog, subscribers could use the last call in the sequence diagram: **unsubscribe()** originates from the subscriber and lets the blog know the subscriber would like to be removed from the list of observers.

Let us now examine this example as a UML diagram:



The **Subject** superclass has three methods: register observer, unregister observer, and notify. These are essential for a subject to relate to its observers. A subject may have zero or more observers registered at any given time. The Blog subclass would inherit these methods.

The **Observer** interface only has the update method. An observer must have some way to update itself. The **Subscriber** class implements the Observer interface, providing the body of an update method, so a subscriber can get what changed in the blog.

Let us now examine this example as Java code.

```
public class Subject {
	private ArrayList<Observer> observers = new
ArrayList<Observer>();

	public void registerObserver(Observer observer) {
		observers.add(observer);
	}
	public void unregisterObserver(Observer observer) {
		observers.remove(observer);
	}
```

```
        public void notify() {
                for (Observer o : observers) {
                        o.update();
                }
        }
}
```

In the **Subject** superclass, the **registerOberserver()** method adds an observer to the list of observers. The **unregisterObserver()** method removes an observer from the list. The notify method calls update upon each observer on the list.

The **Blog** class is a subclass of **Subject**, which will inherit the **registerObserver()**, **unregisterObserver()**, and **notify()** methods. Also, the **Blog** class has the other responsibilities of managing a blog and posting methods.

The Observer interface would be as below:

```
public interface Observer {
        public void update();
}
```

The **Observer** interface makes sure all observer objects behave the same way. There is only a single method to implement, **update()**, which is called by the subject. The subject makes sure when a change happens, all its observers are notified to update themselves. In this example, there is a class **Subscriber** that implements the **Observer** interface.

```
class Subscriber implements Observer {
        public void update() {
        // get the blog change
        ...
        }
}
```

This update method is called when the blog notifies the subscriber of a change.

Observer design patterns save time when implementing a system. If many objects rely on the state of one, the observer design pattern has even more value. Instead of managing all

observer objects individually, the subject manages them and ensures observers are updating themselves as needed.

This behavior pattern is typically used to make it easy to distribute and handle notifications of changes across systems in a manageable and controlled way.

Module 3: Working with Design Patterns and Anti-patterns

Upon completion of this module, you will be able to:

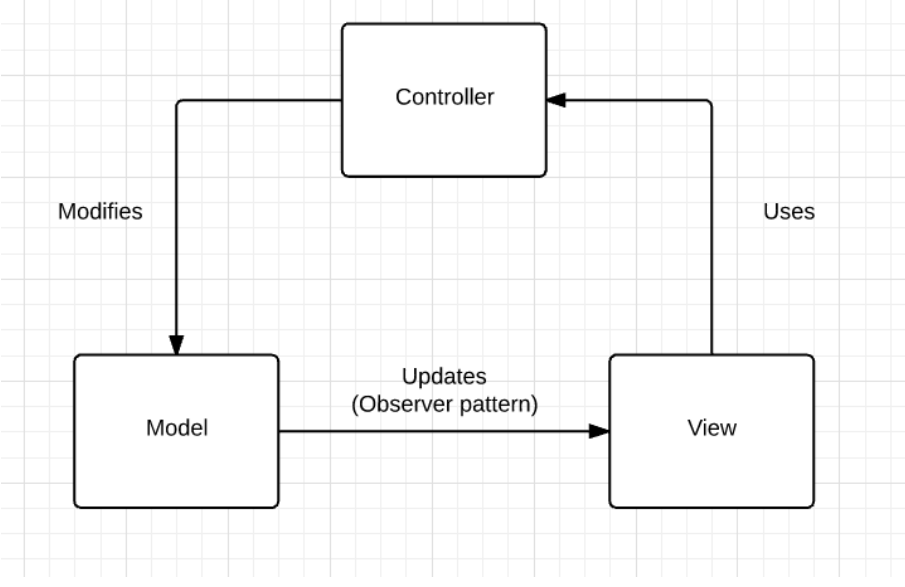
1. Describe the Model View Controller (MVC) design pattern.
2. Explain several general design principles that underlie design patterns, such as
    - the open/close principle
    - the dependency inversion principle
    - the composing object principle
    - the interface segregation principle
    - the principle of least knowledge
3. Analyze and critique code for bad design.
4. Explain antipatterns (also known as code smells), that include:
    - duplicate code
    - long method
    - large class
    - divergent class
    - shotgun surgery
    - long parameter list
    - feature envy
    - data class
    - data clumps
    - primitive obsession
    - switch statements
    - speculative generality
    - message chains
    - inappropriate intimacy
    - refused request
    - comments

This module will cover the Model-View-Controller (MVC) design pattern. This pattern builds upon design patterns presented in the first two modules of this course. This module will also explore several design principles that underlie design patterns. These principles help ensure the software is reusable, flexible, and maintainable. Then, you will learn ways to analyze and critique your code for bad design, by exploring common malpractices. These malpractices are commonly referred to as antipatterns or code smells in the software industry.

MVC Pattern

Model, View, Controller (MVC) pattern is a pattern that should be considered for use with user interfaces. MVC patterns divide the responsibilities of a system that offers a user interface into three parts: model, view, and controller.

Below is a diagram of a simple MVC pattern.



The **Model** is going to contain the underlying data, state, and logic that users want to see and manipulate through an interface. It is the "back end", or the underlying software. A key aspect of the MVC pattern is that the model is self-contained. It has all of the state, methods, and other data needed to exist on its own. The **View** gives users the way to see the model in the way they expect and allows them to interact with it or at least parts of it. It is the "front end" or the presentation layer of the software. A model could have several views that present

different parts of the model, or present the model in different ways.

When a value changes in the model, the view needs to be notified, so it can update itself accordingly. The observer design pattern allows this to happen. In an observer pattern, observers are notified when the state of the subject changes. In this case, the view is an observer. When the model changes, it notifies all of the views that are subscribed to it.

The view may also present users with ways to make changes to the data in the underlying model. It does not directly send requests to the model, however. Instead, information about the user interaction is passed to a **Controller**. The controller is responsible for interpreting requests and interacts with elements in the view, and changing the model. The view is therefore only responsible for the visual appearance of the system. The model focuses only on managing the information for the system.

The MVC pattern uses **the separation of concerns** design principle to divide up the main responsibilities in an interactive system. The controller ensures that the views and the model are loosely coupled. The model corresponds to **entity objects**, which are derived from analyzing the problem space for the system. The view corresponds to a **boundary object**, which is at the edge of your system that deals with users. The controller corresponds to a **control object**, which receives events and coordinates actions.

In order to examine the implementation of this pattern in Java code, let us consider an example. Imagine you are creating an interface for a grocery store, where cashiers can enter orders, which are displayed. Customers and cashiers should be able to see the list of items entered into the order with a barcode scanner and see the total bill amount. Cashiers should also be able to make corrections if necessary.

Model

Let us begin with the model, which is the most essential part of the pattern. The model should be able to exist independently, without views or controllers.

In this example, since the view is going to be an observer, the model needs to be observable. The **java.util** package can be used to implement this behavior. This package contains an

observable class that can be extended. In this example, Observable can be extended, and the **StoreOrder** class will allow you to add your views as observers. This will allow them to update whenever the order is updated.

```
import java.util.*;

public class StoreOrder extends Observable {
   private ArrayList<String> itemList;
   private ArrayList<BigDecimal> priceList;

   public StoreOrder() {
       itemList = new ArrayList<String>();
       priceList = new ArrayList<BigDecimal>();
   }

   public String getItem( int itemNum ) {
       return itemList.get(itemNum);
   }

   public String getPrice( int itemNum ) {
       return priceList.get(itemNum);
   }

   public ListIterator<String> getItemList() {
       ListIterator<String> itemItr = itemList.listIterator();
       return itemItr;
   }

   public ListIterator<BigDecimal> getPriceList() {
       ListIterator<String> priceItr =
priceList.listIterator();
       return priceItr;
   }

   public void deleteItem( int itemNum ) {
       itemList.remove(itemNum);
       priceList.remove(itemNum);
       setChanged();
       notifyObservers();
   }
```

```
    public void addItem( int barcode ) {
        // code to add item (probably used with a scanner)
        // prices are looked up from a database

        ...

        setChanged();
        notifyObservers();
    }

    public void changePrice( int itemNum, BigDecimal newPrice )
{
        priceList.set(itemNum,newPrice);
        setChanged();
        notifyObservers();
    }
}
```

This simple example is a class that has several methods that modify itself. Items can be added, deleted, or have their prices change. When changes are made, the **setChange()** method flags the change, the notify **Observers** method notifies the views so that they can update themselves.

Note that the model does not contain any user interface elements, and it is not aware of any views.

View

Let us now examine one of the views in Java code.

As the **OrderView** class implements the Observer interface, an update method must be provided. Notice that the **storeOrder()** had to be downcast in order to call the **getItemList()** and **getPriceList()** methods. To make the code more type-safe, you could alternately develop your own Observer pattern with Generics.

The view cannot call the method of the Model, but instead calls the methods of the Controller. This is what happens when the view tries to modify the model.

```
import java.util.*;
import javax.swing.JFrame;
// ..etc.

public class OrderView extends JPanel implements Observer,
ActionListener {

   // Controller
   private OrderController controller;

   // User-Interface Elements
   private JFrame frame;
   private JButton changePriceButton;
   private JButton deleteItemButton;
   private JTextField newPriceField;
   private JLabel totalLabel;
   private JTable groceryList;

   private void createUI() {

   // Initialize UI elements. e.g.:
       deleteItemButton = new JButton("Delete Item");
       add(deleteItemButton);

   ...

       // Add listeners. e.g.:
       deleteItemButton.addActionListener(this);

   ...
   }

   public void update ( Observable s, Object arg ) {

       display(((StoreOrder) s).getItemList(), ((StoreOrder)
s).getPriceList());
   }
```

```
    public OrderView(OrderController controller) {
        this.controller = controller;
        createUI();
    }

    public void display ( ArrayList itemList, ArrayList
priceList ) {
        // code to display order
        ...
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == deleteItemButton) {
            controller.deleteItem(groceryList.getSelectedRow());
        }
        else if (event.getSource() == changePriceButton) {
            BigDecimal newPrice = new
BigDecimal(newPriceField.getText());
            controller.changePrice(groceryList.getSelectedRow(),
newPrice);
        }
    }
}
```

Controller

Let us now examine the controller as Java code.

This example is very simple but still demonstrates that the controller must have references to both the view and model that it connects. Notice also that the controller does not make changes to the state of the model directly. Instead, it calls methods of the model to make changes.

```
public class OrderController {

    private StoreOrder storeOrder;
    private OrderView orderView;
```

```
    public OrderController(StoreOrder storeOrder, OrderView
orderView) {
        this.storeOrder = storeOrder;
        this.orderView = orderView;
    }

    public void deleteItem( int itemNum ) {
        storeOrder.deleteItem( itemNum );
    }

    public void changePrice(int itemNum, BigDecimal newPrice) {
        storeOrder.changePrice( itemNum, newPrice );
    }
}
```

Controllers make the code better in the following ways:

- The view can focus on its main purpose: presenting the user interface, as the controller takes the responsibility of interpreting the input from the user and working with the model based on that input.
- The view and the model are not "tightly coupled" when a controller is between them. Features can be added to the model and tested long before they are added to the view.

Controllers make the code cleaner and easier to modify. Note that in most software systems, there are generally multiple models, view, and controller classes.

MVC patterns may be used in many ways. The defining feature of the MVC pattern is the separation of concerns between the back-end, the front-end, and the coordination between the two.

DID YOU KNOW?

The MVC pattern is particularly important because of its use of the separation of concerns. It allows the program to be modular and loosely coupled. Loose coupling is key for large development teams – it allows different teams to work on different parts, so one team can work on the only the view, and another team can work on only the model. The view and the model can change, without needing knowledge of the other.

Design Principles Underlying Design Patterns

All design patterns follow a basic set of design principles. This next lesson will examine some of these underlying design principles. These principles address issues such as flexibility and reusability.

Open/Closed Principle

The **open/closed principle** states that classes should be open for extension but closed to change.

A class is considered "closed" to editing once it has:

- Been tested to be functioning properly. The class should behave as expected.
- All the attributes and behaviors are encapsulated,
- Been proven to be stable within your system. The class or any instance of the class should not stop your system from running or do it harm.

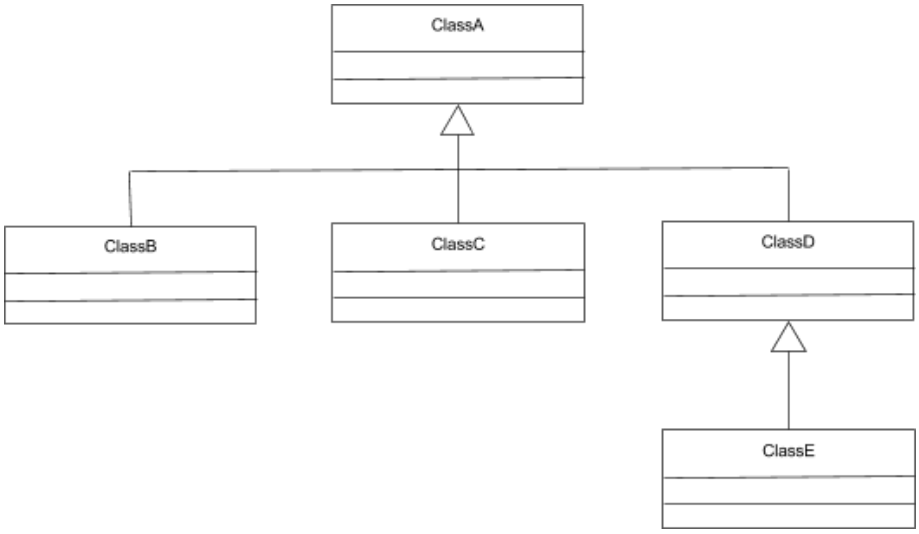
Although the principle is called "closed", it does not mean that changes cannot be made to a class during development. Things should change during the design and analysis phase of your development cycle. A "closed" class occurs when a point has been reached in development when most of the design decisions have been finalized and once you have implemented most of your system.

During the lifecycle of your software, certain classes should be closed to further changes to avoid introducing undesirable side effects. A "closed" class should still be fixed if any bugs or unexpected behaviors occur.

If a system needs to be extended or have more features added, then the "open" side of the principle comes into play. An "open" class is one that can still be built upon. There are two different ways to extend a system with the open principle.

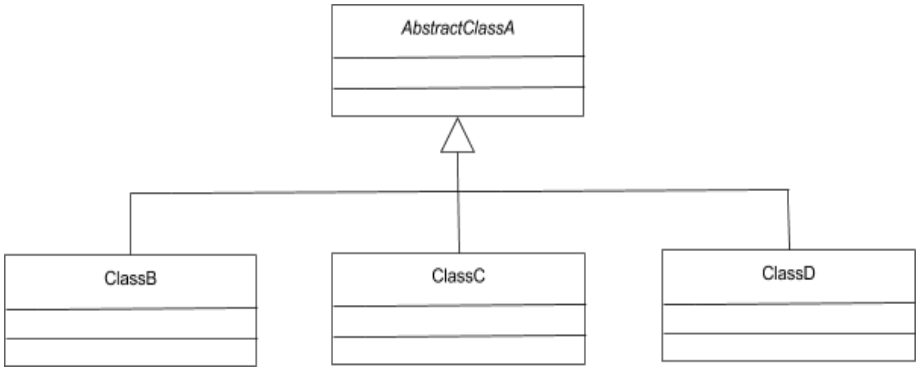
The first way is through the **inheritance** of a superclass. Inheritance can be used to simply extend a class that is considered closed when you want to add more attributes and behaviors. The subclasses will have the original functions of the superclass, but extra features can be added in the subclasses. This helps preserve the integrity of the superclass, so if the extra features of the subclasses are not needed, the original

class can still be used. Note that subclasses can also be extended, so this allows the open/closed principle to continually extend your system as much as desired.



If you want to limit a class so that it is no longer extendable, it can be declared as “final” to prevent further inheritance. This keyword can also be used for methods.

The second way a class can be open is when the class is abstract and enforces the open/closed principle through polymorphism. An abstract class can declare abstract methods with just the method signatures. Each concrete subclass must provide its own implementation of these methods. The methods in the abstract superclass are preserved, and the system can be extended by providing different implementations for each method. This is useful for behaviors like sorting or searching.



An interface can also enable polymorphism, but remember that it will not be able to define a common set of attributes.

The open/closed principle is used to keep stable parts of a system separate from varying parts. It allows the addition of new features to a system, but without the expense of disrupting working parts. Extension, unlike change, allows varying parts to be worked upon while avoiding unwanted side effects in the stable parts. Varying parts should be kept isolated from one another, as these extensions will eventually become stable, and there is no guarantee they will all be finished at the same time, as some features may be more complex, larger, or more ambitious than others.

All design patterns use the open/closed principle in some way. They all follow the idea that some part of your system should be able to extend and be built upon through some means like inheritance or implementing an interface. It may not always be possible to practice this principle, but it is something to strive towards.

The open/closed principle:

- Helps keep a system stable by "closing" classes to change.
- Allows a system to open for extension through inheritance or interfaces.

Dependency Inversion Principle

Software dependency, or coupling, is a common problem that needs to be addressed in system design. Coupling defines how much reliance there is between different components of your software. High coupling indicates a high degree of reliance. Low coupling indicates low dependency. Dependency determines how easily changes can be made to a system.

> Instructor's Note: For a review on coupling, check out the lesson on Cohesion and Coupling in the first course of this specialization!

If parts of a system are dependent on each other, then substitute a class or resource for another is not easily done, or even impossible.

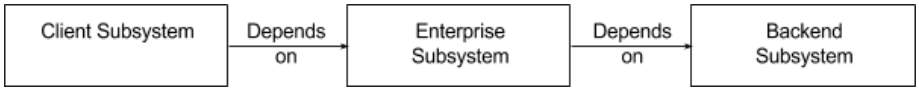
The **dependency inversion principle** addresses dependency, to help make systems more robust and flexible. The principle states that high-level modules should depend on high-level generalizations, and not on low-level details. This keeps client classes independent of low-level functionality.

This suggests that client classes should depend on an interface or abstract class, rather than a concrete resource. Further, concrete resources should have their behaviors generalized into an interface or abstract class. Interfaces and abstract classes are high-level resources. They define a general set of behaviors. Concrete classes are low-level resources. They provide the implementation for behaviors.

All the design patterns covered in this course are built on this dependency inversion principle.

Low Level Dependency

In order to better understand the dependency inversion principle, let us examine low-level dependency. In a **low-level dependency**, the client classes name and make direct references to a concrete class. Subsystems are directly dependent on each other, so the client class will directly reference some concrete class in the enterprise subsystem, which will directly reference some concrete class in your backend.



Let us examine this as code.

```
public class ClientSubsystem {
   public QuickSorting enterpriseSorting;
   /* Constructors and other attributes go here */

   public void sortInput(List customerList) {
       this.enterpriseSorting.quickSort(customerList);
   }
}
```

This has a low-level dependency. There is direct naming and referencing to the concrete class, **QuickSorting**, which is used in this example to sort a list sent to the system by a user. However, if a different sorting class called **MergeSort** were to be implemented, significant changes would need to be made to the **ClientSubsystem** class. The type of sorting class would have to change, as well as the sorting method call. This would require a great deal of work every time a change needed to be made to the sorting algorithm.

```
public class ClientSubsystem {
    public MergeSorting enterpriseSorting;
    /* Constructors and other attributes go here */

    public void sortInput(List customerList) {
        this.enterpriseSorting.mergeSort(customerList);
    }
}
```

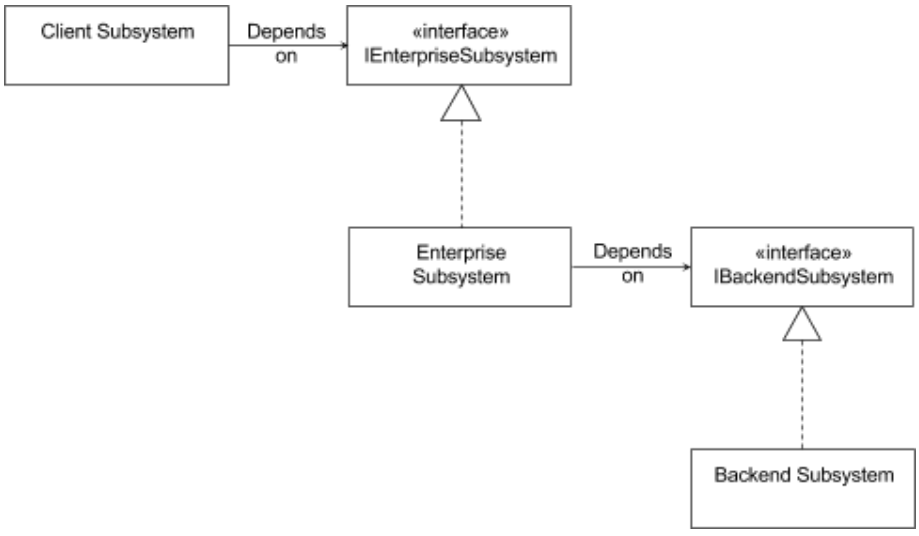
Such large changes are impractical and could entail unexpected side effects if old references are missed. The dependency inversion principle addresses this issue by generalizing low-level functionality into interfaces or abstract classes. This way, when alternative implementations are offered, they can be used with ease.

High-Level Dependency

In **high-level dependency**, the client class is dependent on a high-level generalization rather than a low-level concrete class, allowing changes to be more easily made to resources. The generalization gives a degree of indirection by allowing behaviors to be invoked in a concrete class through an interface.

High-level dependency prevents the client class from being directly coupled with a specific concrete class. A client class is dependant on expected behaviors, not on a specific implementation of behaviors.

When the dependency inversion principle and high-level dependency are used, the overall architecture of a system will look similar to design patterns explored earlier in this course. The diagram below illustrates this.



Behaviors are generalized across each subsystem into an interface. The concrete classes of each subsystem will then implement the interface. Client classes will make references to the interface instead of directly to the concrete classes.

Let us examine this as code.

```
public class ClientSubsystem {
    public Sorting enterpriseSorting;

    public ClientSubsystem(Sorting concreteSortingClass) {
        this.enterpriseSorting = concreteSortingClass;
    }

    public void sortInput(List customerList) {
        this.enterpriseSorting.sort(customerList);
    }
}
```

A **Sorting** interface is declared instead of a concrete class in this example. This allows you to determine which instance of **Sorting** class you want the **ClientSubsystem** to have during instantiation. The sorting behavior can also be generalized into a method called sort in the interface, and the method call does not need to change in the **ClientSubsystem**.

Object-oriented designs should use method calls through a level of indirection. The dependency inversion principle facilitates the use of indirections by writing dependencies to

generalizations, rather than to a concrete class. It helps insulate parts of a system from specific implementations of functionality. It is best practice to program to generalizations, like interfaces and abstract classes, rather than directly to a concrete class.

The dependency inversion principle is a means to:

- Change the referencing of concrete classes from being direct to indirect.
- Generalize the behaviors of your concrete classes into abstract classes and interfaces.
- Have client classes interact with your system through a generalization rather than directly with concrete resources.
- Put emphasis on high-level dependency over low-level concrete dependency.

Using the dependency inversion principle may seem like extra work, but the effort is worthwhile, particularly if you are working on a large system. If a system is too heavily coupled, changes will be difficult to make. This principle helps ensure a robust software solution.

Composing Object Principle

Tight coupling is a common problem in system design. Coupling can be managed using the design principles that underlie design patterns. The design patterns examined in this course use generalization, abstraction, and polymorphism - as a means of indirection and a means to reduce coupling.

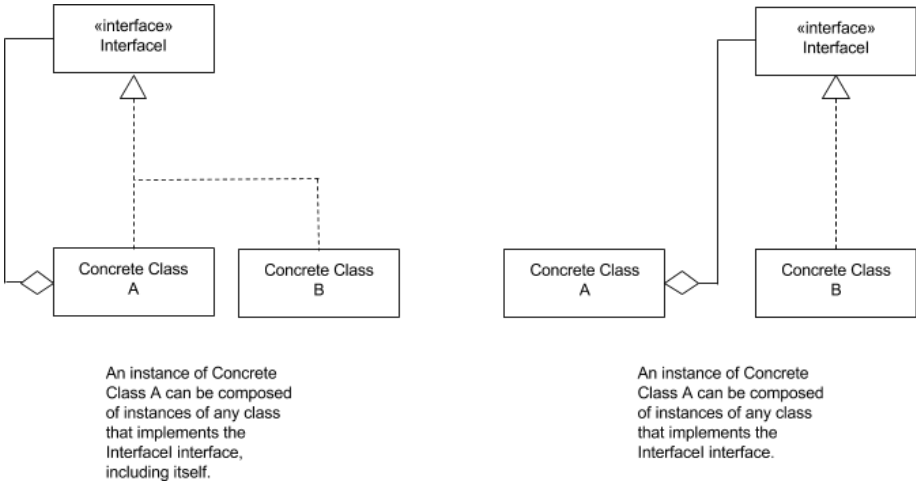
Inheritance can be a great way to achieve a high level of code reuse; it entails a cost of tightly coupling superclasses with their subclasses. Subclasses inherit all attributes and methods of a superclass, as long as access modifiers are not private. This means that if you have multiple levels of inheritance, a subclass at the bottom of the inheritance tree can potentially provide access to attributes and behaviors of all the superclasses.

The **coupling objects principle** is a means of circumventing this problem, by providing a means for a high amount of code reuse without using inheritance. The principle states that classes should achieve code reuse through aggregation rather than inheritance. Aggregation and delegation offer less coupling than inheritance, since the composed classes do not

share attributes or implementations of behaviors, and are more independent of each other. This means they have an "arms-length" relationship.

Design patterns like the composite design pattern and decorator design pattern use this design principle. These two patterns compose concrete classes to build more complex objects at run time. The overall behavior comes from the composed sum of the individual objects. An object can reuse and aggregate another object to delegate certain requests to it. The system should be designed so that concrete classes can delegate tasks to other concrete classes.

The way classes will compose themselves is determined during design. Objects can be composed recursively and uniformly, or they can be composed of other objects that have a consistent type. This is illustrated in the UML diagram below.



Composing objects provides your system with more flexibility. When objects are composed, it is not required to find commonalities between two classes and couple them together with inheritance. Instead, classes can be designed to work together without sharing anything. This provides flexibility, especially if the system requirements change. Inheritance may require restructuring the inheritance tree.

Composing objects also allows the dynamic change of behaviors of objects at run time. A new overall combination of behavior can be built by composing objects. Inheritance, on the other hand, requires behaviors of classes to be defined during compile-time, so they cannot change while the program is running.

In addition to these advantages, there are disadvantages to keep in mind for composition. In composition, implementations for all behavior must be provided, without the benefit of inheritance to share code. There could be very similar implementations across classes, which could take time and resources. With inheritance, common implementation is simply accessed within the superclass, so each subclass does not have to have its own implementation of shared behavior.

In summary, the composing objects principle will:

- Provide a means of code reuse without the tight coupling of inheritance.
- Allow objects to dynamically add behaviors at run time.
- Provide your system with more flexibility, so that less time needs to be spent on system updates.

Although composing provides better flexibility and less coupling, while maintaining reusability, there is a place for inheritance. In order to determine which design principle is appropriate to use, you must assess the needs of your system, and choose what best fits the situation and problem at hand.

DID YOU KNOW?

Some good questions to help you decide whether the best solution for your system is composition or inheritance include:

- Do you have a set of related classes or unrelated classes?
- What is a common behaviour between them?
- Do you need specialized classes to handle specific cases?
- Do you need a different implementation of the same behaviour?

Interface Segregation Principle

Many design patterns that this course has explored use generalizations of concrete classes, presented as interfaces for client classes to indirectly invoke the behaviors of the concrete classes. This helps client classes be less dependent on concrete classes and allows for changes to be made more easily. In

general, your design should strive to program to interfaces instead of concrete classes.

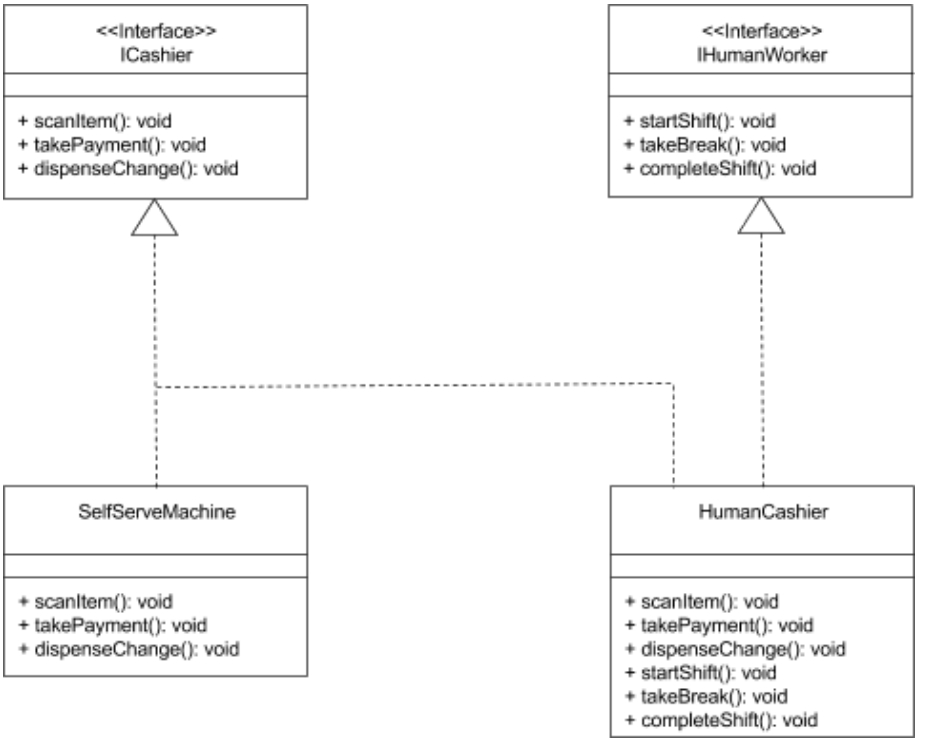
However, if there is only a single interface that holds all the methods for all the clients, then all clients will have to unnecessarily implement all other clients' methods just to make their interface compile. Not all behaviours are shared by clients, so why should they be stored in the same interface? This forces dependency.

The **interface segregation principle** helps tackle this issue so that you won't run into dependency issues or be forced to generalize everything into one interface. The principle states that a class should be forced to depend on methods it does not use. This means that any classes that implement an interface should not have "dummy" implementations of any methods defined in the interfaces. Instead, large interfaces should be split into smaller generalizations.

Let us examine this through an example. Imagine creating a checkout system for a grocery store that allows customers to pay for their goods in two ways: they can use an automated machine, or they can approach a human cashier operating a till. Although the purpose of these two options can be generalized with an interface, there are some behaviors that are not shared between the two.

A poor interface design would entail a single interface for both the **HumanCashier** class and the **SelfServeMachine** class. This interface would include all the methods for both classes, whether or not they fit both classes. These would be "dumb" implementations that do not do anything depending on which class called upon the interface.

A good interface design would apply the interface segregation principle and split the interface into two smaller ones. This allows each interface to be more accurate with its description of expected behaviours. The interface segregation principle would select the correct combinations of interfaces a concrete class should implement. Represented as a UML diagram, this would look as below.



The **HumanCashier** class can implement both interfaces, and the **SelfServeMachine** class would only need to implement the **ICashier** interface. This way, both concrete classes only need to provide implementations for the interfaces that generalize their specific functionality. The interfaces should be kept separate, and the **HumanCashier** class can implement two interfaces. This is preferable to have an **IHumanWorker** interface that inherits the **ICashier** interface, as the **IHumanWorker** interface would inherit behavioral descriptions of a cashier, which may not always apply to an employee. Remember, the single inheritance rule in Java applies to classes, but not to interfaces. An interface can inherit from as many interfaces as it wants to.

Interfaces are an integral part of object-oriented systems. They reduce coupling by generalizing concrete classes. However, they must be used properly. Large interfaces in and of themselves are not necessarily a poor design choice, but you need to take into account the context of your system to decide if the interface segregation principle should be applied and if the interfaces should be split into smaller generalizations.

Note that it will not always be clear how to properly segregate your interfaces, or to predict future changes in requirements that will need interfaces to be split up. Well-defined interfaces

are important – interfaces are descriptions of what parts of your system can do. The better the description the easier it will be to create, update, and maintain software.

In summary, the interface segregation principle states that:

- A class should not be forced to depend on methods it does not use.
- Interfaces should be split up in such a way that they can properly describe the separate functionalities of your system.

Principle of Least Knowledge

The **principle of least knowledge** states that classes should know and interact with as few other classes as possible. This principle arises from one means of managing complexity, which suggests that a class should be designed so that it does not need to know about and depend upon almost every other class in the system. If classes have a narrow view of what other classes it knows, then coupling is decreased, and a system is easier to maintain.

This principle is also realized in a rule known as the **Law of Demeter**. The Law of Demeter is composed of several different rules that provide guidelines as to what kind of method calls a particular method can make – in other words, they help determine how classes should interact. This design principle focuses on how method calls should be made, rather than on specific ways of structuring the design of a system.

The four rules of the Law of Demeter as summarized below: A method, M, of an object should only call other methods if they are:

1. Encapsulated within the same object
2. Encapsulated within an object that is in the parameters of M
3. Encapsulated within an object that is instantiated inside the M
4. Encapsulated within an object that is referenced in an instance variable of the class for M

Let us examine each of these rules in turn.

The first rule states that a method, M, in an object, O, can call on any other method within O itself. In other words, a method

encapsulated within a class is allowed to call any other method also encapsulated within the same class.

The second rule states that a method, M, can call the methods of any parameter P. As a method parameter is considered local to the method, methods in the class for the parameter are allowed to be called.

The third rule states that a method, M, can call a method, N, of an object, I, if I is instantiated within M. This means that if a method makes a new object, then the method is allowed to use that new object's methods. The object is considered local to the creating method, the same way an object is considered local when it is a parameter.

The fourth rule states that in addition to local objects, any method, M, in object O, can invoke methods of any type of object that is a direct component of O. This means that a method of a class can call methods of the classes of its instance variables. If a class has direct reference to the Friend class, any method inside of O is allowed to call any method of Friend class. See below for an example of this rule in Java code.

```
public class Friend {
   public void N() {
       System.out.println("Method N invoked");
   }
}

public class O {
   public Friend I = new Friend();
   public void M() {
       this.I.N();
       System.out.println("Method M invoked");
   }
}
```

The Law of Demeter may seem complicated and abstract, but at its core, boils down to the idea that a method should not be allowed to access another method by "**reaching through**" an object. This means that a method should not invoke methods of any object that is not real.

The four rules explained above help outline what is considered a local object. These objects should be passed in through a

parameter, or they should be instantiated within a method, or they should be in instance variables. This allows methods to directly access the behaviors of local objects.

These conditions commonly occur when you have a chain method calls to objects you should not know about, or when you use methods from an "unknown" type of object that is returned back to you from your local method call. A good metaphor for this is driving a car. When you drive a car, you do not issue individual commands to every component of the car. Instead, you simply tell the car to drive, and the car will know to deal with its components.

Another way you can "reach through" an object is when the method receives an object of an unknown type as a return value, and you make method calls to the returned object.

Returned objects must be of the same type as:

- Those declared in the method parameter
- Those declared and instantiated locally in the method
- Those declared in instance variable of the class that encapsulates the method

The Law of Demeter defines how classes should interact with each other. They should not have full access to the entire system because this causes a high degree of coupling. A class should only interact with those that are considered immediate "friends". This prevents unwanted effects from cascading through the entire system.

Although the Law of Demeter can help reduce coupling, you should recognize that it also requires more time when designing and implementing a system. It may not always be feasible to follow the Law of Demeter due to limitations of time and resources. Other times, some degree of coupling may be unavoidable, at which point, you'll need to decide how much coupling is tolerable.

Anti-Patterns and Code Smells

No matter how well you design your code, there will still be changes that need to be made. **Refactoring** helps manage this. It is the process of making changes to your code so that the external behaviours of the code are not changed, but the internal structure is improved. This is done by making small, incremental changes to the code structure and testing

frequently to make sure these changes have not altered the behavior of the code.

Ideally, refactoring changes are made when features are added, and not when the code is complete. This saves time and makes adding features easier.

Changes are needed in code when bad code emerges. Just like patterns emerge in design, bad code can emerge as patterns as well. These are known as **anti-patterns** or **code smells.**

The book *Refactoring* by Martin Fowler has identified many of these anti-patterns. Code smells help "sniff out" what is bad in the code. This book also provides ways to reactor changes in order to transform the code, and "improve the smell".

This next lesson will examine some of the code smells explored in Fowler's book. For more information on these code smells, and proposed resolutions to them, Fowler's book is an excellent resource. You can find bibliographic information about the book in **Course Resources**.

Comments

One of the most common examples of bad code is **comments**. This code smell can occur between two extremes. If no comments are provided in the code, it can be hard for someone else, or even the original developer, to return to the code after some time has passed and tried to understand what the code is doing or should be doing.

On the other hand, if there are too many comments, they might get out of sync as the code changes. Comments can be a "deodorant" for bad-smelling code. The use of a lot of comments to explain complicated design can indicate that bad design is being covered up.

There are other ways that comments can indicate bad code, though. If the comments take on a "reminder" nature, they indicate something that needs to be done, or that if a change is made to one section in the code, it needs to be updated in another method -- these are two examples that indicate bad code.

Comments might reveal that the programming language selected for the software is not appropriate. This could happen if the programming language does not support the design

principles being applied. For example, when Java was in its early years, the concept of generics did not exist yet. Developers would use comments to explain what they were doing with code when casting types. The use of comments is common in young programming languages! However, generics have now been built into Java.

Comments are very useful for documenting application programmer interfaces (APIs) in the system and for documenting the rationale for a particular choice of data structure or algorithm, which makes it easier for others to understand and use the code. But comments should be used in balance, or they may indicate a bad smell!

Duplicate Code

Duplicated code occurs when blocks of code exist in the design that is similar but has slight differences. These blocks of code appear in multiple places in the software, which can be a problem, because if something needs to change, then the code needs to be updated in multiple places. This applies to adding functionalities, updating an algorithm, or fixing a bug.

Instead, if the code only needed to be updated in one location, it is easier to implement the change. It also reduces the chance that a block of code was missed in an update or change.

This anti-pattern relates to the **D.R.Y. principle**, or "Don't Repeat Yourself", which suggests that programs should be written so that they can perform the same tasks but with less code. You can review this rule in the first course of this specialization.

Long Method

The **long method** anti-pattern suggests that code should not have long methods. Long methods can indicate that the method is more complex or has more occurring within it than it should.

Determining if a code is too long can be difficult. There is even some debate if length of code is even a good measure of code complexity. Some methods, such as setting up a user interface, can be naturally long, even if focused on a specific task. Sometimes a long method is appropriate.

This anti-pattern may also depend on the programming language for the system. For example, Smalltalk methods tend to be short, with a 15-line average. Designs in this language tend to have highly cohesive classes and methods, but are also highly coupled. Context is therefore important in this anti-pattern.

DID YOU KNOW?

Some developers suggest that having an entire method visible at once on the screen is a good guideline, with no more than around 50 lines of code. However, some studies show that programmers can handle methods of a couple hundred lines before they introduce bugs! Determining "how long is too long" for a method isn't always a straightforward process!

Large Class

The **large class** anti-pattern suggests that classes should not be too large. This is similar to the long pattern above.

Large classes are commonly referred to as God classes, Blob classes, or Black Hole classes. They are classes that continue to grow and grow, although they typically start out as regular-sized classes. Large classes occur when more responsibilities are needed, and these classes seem to be the appropriate place to put the responsibilities. Over time, however, these responsibilities might attract more responsibilities, and the class continues to get larger and larger and take on more and more responsibilities.

This growth will require extensive comments to document wherein the code of the class certain functionalities exist. Comments, as discussed previously in this lesson, are another indicator of bad code. It is helpful to catch this code smell early, as once a class is too large, it can be difficult and time-consuming to fix.

Classes should have an explicit purpose to keep the class cohesive, so it does one thing well. If functionality is not specific to the class's responsibility, it may be better to place it elsewhere.

Data Class

The **data class** anti-pattern is on the opposite end of the spectrum of the large class. It occurs when there is too small of a class. These are referred to as data classes. Data classes are classes that contain only data and no real functionality. These classes usually have only **getter** and **setter** methods, but not much else. This indicates that it may not be a good abstraction or a necessary class.

An example of a data class would be a 2D point class that only has x and y coordinates. Could something else be placed in the class? What are the other classes manipulating this data? Would some of their behaviors be better placed in one of the data classes? The 2D point class could have various transformation functions that move the point. This would make better code.

Data Clumps

Data clumps are groups of data appearing together in the instance variables of a class, or parameters to methods. Consider this code smell through an example. Imagine a method with integer variables x, y, and z.

```
public void doSomething (int x, int y, int z) {

   ...
}
```

If there are many methods in the system that perform various manipulations on the variables, it is better to have an object as the parameter, instead of using the variables as parameters repeatedly. That object can be used in its place as a parameter.

```
public class Point3D {

   private int x;
   private int y;
   private int z;

   public int getX() {
           return x;
   }

   public int getY() {
```

```
        return y;
    }

    public int getZ() {
        return z;
    }

    public void setX(int newX) {
        x = newX;
    }

    public void setY(int newY) {
        y = newY;
    }

    public void setZ(int newZ) {
        z = newZ;
    }
}
```

Be careful not to just create data classes, however. The classes should do more than just store data. The original **doSomething()** method, or a useful part it, might be added to the Point3D class to avoid this.

DID YOU KNOW?

Fixing code smells can lead to creating new smells elsewhere in the code! In the example above, to fix the data clump smell, you might have accidentally created a data class smell. Be careful when fixing code smells, and keep an eye out for these occurrences of new ones, so they can be fixed.

Long Parameter List

Another code smell is having **long parameter lists**. A method with a long parameter list can be difficult to use. They increase the chance of something going wrong. The common solution to reduce the number of parameters is often to have global

variables but these have issues of their own and generally should be avoided.

Methods with long parameter lists require extensive comments to explain what each of the parameters does and what it should be. Extensive commenting, as explained earlier in this lesson, is another code smell.

If long parameter lists are not commented, however, then it may be necessary to look inside the implementation to see how they are used which breaks encapsulation.

The best solution for long parameter lists is to introduce parameter objects. A parameter object captures context. They are common in graphics libraries, where toolkits are passed into an object to provide context for a graphics routine. Instead of setting details, such as the pen color, line width, pen shape, opacity, etc., as individual parameters, this information would be provided as a parameter object. When the program needs to execute a method, all of the information about what the stroke should look like can be an individual parameter value.

Divergent Change

Some code smells occur when making changes to the code itself. A **divergent change** is one such code smell. It occurs when you have to change a class in many different ways, for many different reasons. This relates to the large class code smell, where a large class has many different responsibilities. These responsibilities may need to be changed in a variety of ways, for a variety of purposes. Poor separation of concerns is therefore a common cause of divergent change.

Classes should have only one specific purpose because this reduces the number of reasons that the code would need to change, and reduces the variety of changes needed to be implemented. If a class is being changed in multiple ways, then this is a good indicator that the responsibilities of the class should be broken up into separate classes, and responsibilities should be extracted into their own classes.

Separation of concerns resolves two code smells – large class and divergent change.

Shotgun Surgery

Shotgun surgery is a code smell that occurs when a change needs to be made to one requirement, and numerous classes all over the design need to be touched to make that one change. In good design, a small change is ideally localized to one or two places (although this is not always possible).

DID YOU KNOW?

This code smell was so named because it draws on the metaphor of a shotgun. If you want to make a change, it should not be like taking a shotgun to your code and creating a large splatter effect.

This is a commonly occurring code smell. It can happen if you are trying to add a feature, adjust code, fix bugs, or change algorithms. If changes need to be made all over the code, then the chances of missing a change or creating an issue elsewhere increase.

Modular code is not always an option, however. Some changes require shotgun surgery no matter how well designed the code is. For example, changing a copyright statement or licensing will require changing every file in the system to update the block of copyright text.

The shotgun surgery smell is normally resolved by moving methods around. If a change requires you to make changes to many methods in different classes, then that can be an indicator that the methods may be better consolidated into one or two classes. However, you must be careful not to move so many methods into one class because that it becomes a large class code smell.

If a change in one place leads to changes in other places, then this indicates that the methods are related in some way. Perhaps there is a better way to organize them. If not, then you may have to deal with it as it is.

Feature Envy

Feature envy is a code smell that occurs when there is a method that is more interested in the details of a class other than the one it is in. If two methods or classes are always talking to one another and seem as if they should be together, then chances are this is true.

Inappropriate Intimacy

Inappropriate intimacy is a code smell that occurs when two classes depend too much on one another through two-way communication.

If two classes are closely coupled, e.g., so a method in one class calls methods of the other, and vice versa, then it is likely necessary to remove this cycle. Methods should be factored out so both classes use another class. At the very least, this makes communication one way and should create looser coupling.

Cycles are not always necessarily a bad thing. Sometimes, they are necessary. But, if there's a way to make the design simpler and easier to understand, then it is a good solution.

Message Chains

Message chains is a code smell that occurs when the code has long message chains where you are calling, and get an object back, and then calling again, and getting another object. This is a bad message chain, and it can cause rigidity, or complexity in your design, and makes code harder to test independently. It also potentially violates the Law of Demeter, which specifies which methods are allowed to be called.

In a bad message chain, to find an object in the code, it might be necessary to navigate the chain and these object's dependencies. However, particular navigation should not be baked into objects. If the design were reworked, and the dependencies were restructured, it will break the code, which is a sign of a brittle design.

Long chains of calls could be appropriate, however, if they return a limited set of objects that methods are allowed to be called on. If those objects follow the Law of Demeter, then the chain is appropriate.

Primitive Obsession

Primitive obsession is a code smell that occurs when you rely on the use of built-in types too much. Built-in types, or **primitives**, are things like **ints**, **longs**, **floats**, or **strings**. Although there will be a need for them in the code, they should only exist at the lowest levels of the code. Overuse of primitive

types occurs when abstractions are not identified, and suitable classes are not defined.

Theoretically, almost everything in a system could be defined or encoded as **Strings** and put into **Arrays**. In fact, in the 1960s, this is how code looked. Languages have evolved though, to allow for defining your own types for better abstraction. Otherwise, you have a **non-OO** way of thinking. If everything is stored as a string, then a key abstraction could be buried in the detailed code and would not be evident when looking at the design of the system, such as through a UML diagram. There would be no clear separation between strings.

If you are using primitive types often at a high level, then it is a good indicator that suitable classes are not being declared, and there is a primitive obsession code smell in the system.

Switch Statements

Switch statements are code smells that occur when switch statements are scattered throughout a program. If a switch is changed, then the others must be found and updated as well.

Although there can be a need for **long if/else** statements in the code, sometimes switch statements may be handled better. For example, if conditionals are checking on type codes or the types of something, then there is a better way of handling the switch statements. It may be possible to reduce conditionals down to a design that uses polymorphism.

Speculative Generality

The code smell **speculative generality** occurs when you make a superclass, interface, or code that is not needed at the time, but that may be useful someday. This practice introduces generality that may not actually help the code, but "over-engineers" it.

In Agile development, it is best to practice **Just in Time Design**. This means that there should be just enough design to take the requirements for a particular iteration to a working system. This means that all that needs to be designed for are the set of requirements chosen at the beginning of an iteration. All other requirements in a backlog can be ignored, as they may never actually be needed.

Software changes frequently. Clients can change their mind at any time, and drop requirements from the backlog. Your design should stay simple, and time should not be lost on writing code that may never be used.

If generalization is necessary, then it should be done. This change may take longer at the time, compared to if it is set up beforehand, but it is better than writing code that may not be needed down the line.

Refused Request

A **refused request** code smell occurs when a subclass inherits something but does not need it. If a superclass declares a common behavior across subclasses, and subclasses are inheriting things they do not need or use, then they may not be appropriate subclasses for the superclass. Instead, it may make more sense for a stand-alone class. Or perhaps the unwanted behaviors should not be defined in the superclass. Finally, if only some subclasses use them, then it may be better to define those behaviors in the subclasses only.

Code smells help identify bad code and bad design. It is good practice to review your code frequently for code smells to ensure that the code remains reusable, flexible, and maintainable.

This brings us to the end of this course. The next course in this specialization is on Software Architecture.

Course Resources

Course Readings

- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Upper Saddle River, NJ: Addison-Wesley Professional.

- Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley.

Glossary

Word	Definition
"Don't Repeat Yourself" rule (D.R.Y. rule)	A rule relating to the design principle of generalization. D.R.Y. suggests that we should write programs that are capable of performing the same tasks but with less code. Code should be reused because different classes or methods can share the same blocks of code. D.R.Y. helps make systems easier to maintain.
Adapter design pattern	A structural pattern that facilitates communication between two existing systems by providing a compatible interface.
Anti-patterns	Patterns of bad code commonly emerge in software design. Also known as code smells.
Base object	An object in an "aggregation stack" that contains no other object and has its own set of behaviors.
Behavioral design patterns	Patterns that focus on how objects distribute work, and how they collaborate together to achieve a common goal.
Boundary object	A type of object, usually introduced in the solution space of a software problem that connects to services outside of the system.
Chain of responsibility design pattern	A design pattern that is a change of objects responsible for handling requests.
Cloning	Duplication of an object.

Code smells	Patterns of bad code that commonly emerge in software design. Also known as anti-patterns.
Coding to an interface	Occurs in factory objects, where changes need only be made to a concrete instantiation and not the client method.
Command design pattern	A behavioral design pattern that encapsulates a request as an object of its own.
Comments	An anti-pattern that appears if there are a lot of comments in a code, which can indicate that a developer is relying on those comments to explain what the code does, thereby covering up a bad design.
Composite design pattern	A structural pattern used to achieve two goals: to compose nested structures of objects, and to deal with the classes for these objects uniformly.
Concrete instantiation	The act of instantiating a class to create an object of a specific type.
Control object	A type of object, usually introduced in the solution space of a software problem, that receives events and coordinates actions.
Controller	The part of a MVC pattern that is responsible for interpreting user requests and changing the model.
Coupling objects principle	An underlying design principle for design patterns. The principle states that classes should achieve code reuse through aggregation rather than inheritance.
Creational design patterns	Patterns that tackle how to create or clone new objects.
Data class	An anti-pattern that occurs when a class is too small. Data classes contain only data and no real functionality. They might have getter and setter methods and not much else.
Data clumps	A code smell that occurs when groups of data appear together in the instance variables of a class, or parameters to methods.
Decorator design pattern	A structural pattern that allows additional behaviors or responsibilities to be dynamically attached to an object, through the use of aggregation to combine behaviors at run time.

Dependency inversion principle	An underlying design principle for design patterns. The principle states that high-level modules should depend on high-level generalizations, and not on low-level details.
Design patterns	A reusable solution to a problem identified in software design.
Divergent change	A code smell that occurs when you have to change a class in many different ways, for many different reasons.
Duplicated code	An anti-pattern that occurs when blocks of code are similar, but have slight differences, and appear in multiple places in the software.
Entity object	A type of object, often identified in the problem space, that represents items used in the application.
Façade	A wrapper class that encapsulates a subsystem in order to hide the subsystem's complexity, and acts as a point of entry into a subsystem without adding more functionality in itself.
Façade design pattern	The façade design pattern is a structural pattern used to provide a single, simplified interface for client classes to interact with a subsystem.
Factory Method pattern	A creational pattern, which uses a factory method in the same class to create objects.
Factory object	An object in which object creation happens, allowing methods that use factory objects to focus on other behavior.
Feature envy	A code smell that occurs when a method is more interested in the details of a class other than the one it is in.
Gang of Four	These are the four authors, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, who wrote the famous book on design patterns, *Design Patterns: Elements of Reusable Object-Oriented Software* (1994).
Gang of four's design pattern catalog	A catalog of design principles was developed by the gang of four in their famous book, *Design Patterns: Elements of Reusable Object-Oriented Software* (1994).
High-level dependency	A form of dependency where a client class is dependent on a high-level generalization, instead of a low-level concrete class.

Inappropriate intimacy	A code smell that occurs when two classes depend too much on one another through two-way communication.
Information hiding	Allows modules in a system to give others the minimum amount of information needed to use them correctly and "hide" everything else. This allows modules to be worked on separately.
Inheritance	According to the principle of generalization, repeated, common, or shared characteristics between two or more classes are taken and factored into another class. Subclasses can then inherit the attributes and behaviors of this generalized or parent class.
Interface segregation principle	An underlying design principle for design patterns. The principle states that a class should not be forced to depend on methods it does not use.
Invoker	An object in a command pattern, that invokes the command objects to complete whatever task it is supposed to do.
Just in Time design	An agile development design that suggests that there should be just enough design to take the requirements for a particular iteration to a working system.
Large class	An anti-pattern that occurs when a class is too long. Large classes may also be referred to as God classes, Blob classes, or Black Hole classes.
Law of Demeter	A law that states that shares the principle of least knowledge, classes should know about and interact with as few other classes as possible, but that is composed of several rules that provide a guideline as to what kind of method calls a particular method can make.
Lazy creation	Occurs when an object is not created until it is truly needed.
Letting the subclasses decide	In the Factory Method pattern, the class that uses a factory method is specialized or subclassed. These subclasses must define their own factory method. This is known as "letting the subclasses decide" how objects are made.
Long method	An anti-pattern that occurs when code has long methods.
Long parameter lists	A code smell that occurs when there is a long parameter list.

Low-level dependency	A form of dependency where the client classes are naming and making direct references to a concrete class.
Message chains	A code smell that occurs when there is a long message chain; violates the Law of Demeter and which methods are allowed to be called.
Model	The part of a MVC pattern that contains the underlying data and logic that users want to see and manipulate through an interface.
Model, View, Controller (MVC) pattern	A pattern that divides the responsibilities of a system that offers a user interface into three parts: model, view, and controller.
Non-OO	A way of thinking where everything in a system is defined or encoded as strings and put into arrays, instead of using abstraction.
Observer	Objects in a design pattern, that rely on a subject-object to inform them of changes to the subject.
Observer design pattern	A behavioral design pattern where a subject keeps a list of observers.
Open/closed principle	An underlying design principle for design patterns. The open/closed principle states that classes should be open for extension, but closed to change.
Pattern language	A pattern language is a collection of patterns that are related to a certain problem space.
Polymorphism	In object-oriented languages, polymorphism is when two classes have the same description of behavior but the implementation of the behavior may be different.
Primitive	A built-in type of a system, such as ints, longs, floats, or strings.
Primitive obsession	A code smell occurs when there is too much reliance on built-in types in the system.
Principle of least knowledge	An underlying design principle for design patterns. The principle states that classes should know about and interact with as few other classes as possible
Protection proxy	A proxy class used to control access to the real subject class.

Proxy	A simplified, or lightweight version, of an original object.
Proxy design pattern	A structural pattern that allows a proxy class to represent a real "subject" class.
Reaching through	A term that means that another object needs to be used to pass along a request, or that methods of objects are being used outside of what is considered local.
Recursive composition	Another name for composite design patterns. Also, a concept that allows objects to be composed of other objects that are of a common type.
Refactoring	The process of making changes to code so that the external behaviors of the code are not changed, but the internal structure is improved.
Refused request	A code smell occurs when a subclass inherits something but does not need it.
Remote proxy	A proxy class that is local while the real subject class exists remotely.
Separation of concerns	A principle of software design, which suggests that software should be organized so that different concerns in the software are separated into different sections and addressed effectively.
Shotgun surgery	A code smell occurs when a change needs to be made to one requirement, and numerous classes all over the design need to be touched to make that one change.
Singleton design pattern	A creational pattern in which there is only one object of a class.
Spaghetti code	Tangled or structureless software code.
Speculative generality	A code smell occurs when a superclass, interface, or code is created but that is not needed at the time, and instead may be useful someday.
State design pattern	A behavioral design pattern that can occur as objects in your code are aware of their current state, and thus can choose an appropriate behavior based on their current state. When the current state changes, this behavior can be altered.
Structural design patterns	Patterns that describe how objects are connected to each other.

Subject	An object in an observer design pattern that keeps a list of observers.
Switch statements	A code smell occurs when switch statements are scattered throughout a program, and changing one requires finding all of them to update.
Template method pattern	A behavioral design pattern that defines an algorithm's steps generally, deferring the implementation of some steps to subclasses. In other words, it is concerned with the assignment of responsibilities.
View	The part of a MVC pattern that allows users to see all or part of the model.
Virtual proxy	A proxy class is used in place of a real subject class that is resource-intensive to instantiate.