Kevin Chen

12/24/16

Design Pattern Notes

About these notes: almost every concept is presented in two ways in these notes: once by example and once by generalization. The example is different for each pattern.

**Design Patterns**

Understanding the need for design patterns

* Make your systems more flexible and maintainable: deal with change

What are design patterns?

* They are well-tested solutions to problems in software development (often dealing with flexibility and change).
* Not specific for specific types of software. Instead are general solutions to common problems for all type of software systems.
* Guidelines to structure your objects and their behavior. Lots of room for interpretation for actual needs.
* Invented by the “gang of four”, who created 23 original patterns. Now there are many patterns.

Using design patterns

* Use patterns as guidelines.
* Offers shared vocabularies, efficient communications.
* Ultimate in reuse (reusing experience, not code)

**The Strategy Pattern**

Understanding the pitfalls of inheritance and interfaces

* Inheritance is powerful, but we tend to overuse it, resulting in design and code that is inflexible.
* As you add subclasses that have some different behaviors from the superclass, you begin to override the methods more often, defeating the purpose of inheritance. This makes it harder to gain knowledge of the superclass, and we begin having duplicates across the subclasses if the subclasses override methods in a similar way. Changes in the superclass unexpectedly affects the subclasses who implement those superclass methods differently. Additionally, runtime behavior changes are difficult.
  + **Example**: We begin with a Duck abstract class with default implementations for quack() and swim(). There is an abstract display() method. MallardDuck and RedheadDuck extend Duck and thus overrides the display() method. Now you get a request to add a RubberDuck class, so you add it and extend Duck, but it doesn’t quack (it squeaks) so you override the quack() method. Now you get a request to add a fly() method for Duck, so you add it, but then you realize RubberDucks can’t fly, so you override fly() in RubberDuck. Now you get a request to add a DecoyDuck. This duck can’t quack or fly, so you have to override those methods. With each class you add, you are getting the above problems.
* Interfaces containing a single method may seem to solve this problem. If the subclass needs to have a particular method, then implement the interface that contains that method. But this destroys code reuse (every subclass has to supply its implementation code), resulting in a maintenance nightmare. Also does not allow runtime changes in behaviors other than for the interface methods created.
  + **Example** (continued): you make your Duck abstract class have only swim() and display(), you have Flyable interface with fly() and you have Quackable interface with quack(). All four types of ducks extend Duck, MallardDuck and RedheadDuck implements Flyable and Quackable, and RubberDuck implements Quackable. With this design, you get the above mentioned problems.

Encapsulating code that varies

* Design Principle #1: Identify the aspects of your code that vary and separate them from what stays the same.
* “Encapsulate what varies”: if some superclass method is changing depending on the subclass, that’s a sign you should pull it out and separate it. By doing this, you can extend or alter them without affecting the rest of your code.

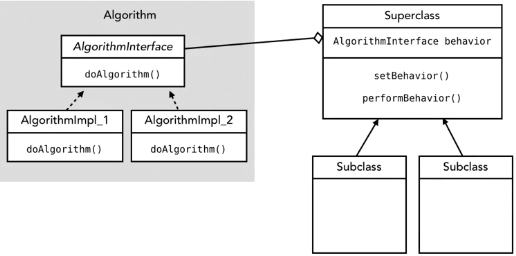
Programming to an interface

* Design Principle #2: Program to an interface, not an implementation.
* For methods that vary, create an interface that only contains the method that varies. For each different implementation of the varying method, create a class that only overrides the interface and contains the implementation for the interface’s method.
* In the original superclass, create a reference for each interface (and write setters for those references). For each instance of the superclass, the instance will contain a reference to the appropriate class that contains the implementation of the interface’s method.
  + We are now using “has a” relationships.
* Rather than relying on an implementation of behavior, we are relying on an interface. This way we are no longer locked into specific implementations. And the subclasses do not need to know details of how they implement the behaviors.
* **Example** (continued): Our Duck abstract class has only swim() and display(). We have a QuackBehavior interface with method quack(), and a FlyBehavior interface with method fly(). For QuackBehavior, we implement a Quack, Squeak, and Mute class (which each override the quack() interface method). For FlyBehavior, we implement a FlyWithWings and a FlyNoWay method (which each override the fly() method). In the Duck superclass, we add a FlyBehavior and a QuackBehavior field, and add setters (setFlyBehavior() and setQuackBehavior()) and executors (performQuack() and performFly()) for these behaviors. Now each subclass of Duck needs to only implement the display() method.

Setting behavior dynamically

* The executor methods in the abstract class should call the single method of the corresponding Behavior interface when called.
  + **Example** (continued): performFly() calls flyBehavior.fly() and performQuack() calls quackBehavior.quack();
* The subclasses set their initial behaviors in the constructor and can later update those behaviors by calling the appropriate setters.
  + **Example** (continued): MallardDuck sets quackBehavior to a new instance of Quack and flyBehavior to a new instance of FlyWithWings.

Exploring the strategy pattern



* Benefits: we can choose which algorithm each object gets, we can change the algorithm at run time, and we can achieve code reuse if multiple objects use the same algorithm
* Official definition: The strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Understanding why Has-A is better than Is-A

* Instead of inheriting behavior, we’re composing it. This is an important technique captured in design principle.
* Design Principle #3: favor composition over inheritance

**The Observer Pattern**

Using the observer pattern in the real world

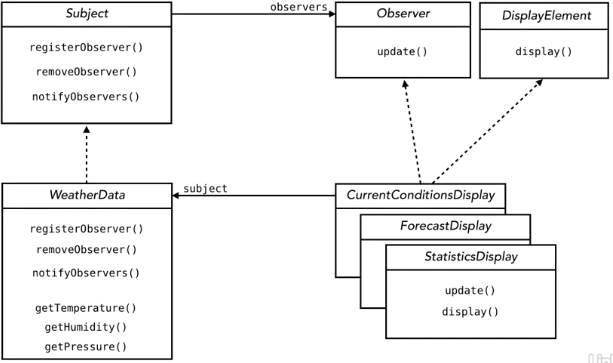
* Think about a publisher and subscribers as sets of objects
  + We have a publisher object to which any other object can send a request to subscribe. When the request is received by the publisher, the requesting object becomes a subscriber.
  + Any object can ask to be a subscriber (and there will be non-subscribing objects)
* The publisher typically holds some data of interest.

Exploring the observer pattern

* The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.
  + One to many relationship between the one subject and the many dependents
  + The dependents are essentially observers of the subject; they get notified anytime the subject state changes
  + Subject methods: registerObserver(), removeObserver(), notifyObservers()
  + Observer method: update(). The notifyObservers() in the Subject calls the update the method() in all of its observers when the data in the subject has changed.
  + The ConcreteSubject and ConcreteObserver classes implement the Subject and Observer interfaces. The ConcreteSubject may have a getter and setter for the state.

Understanding the observer pattern

* Example of the observer pattern (weather station):
  + Weather data is the subject. Consists of humidity, temperature, and pressure.
  + Each display is an observer: current conditions, forecast, and weather stats.
* UML Diagram:

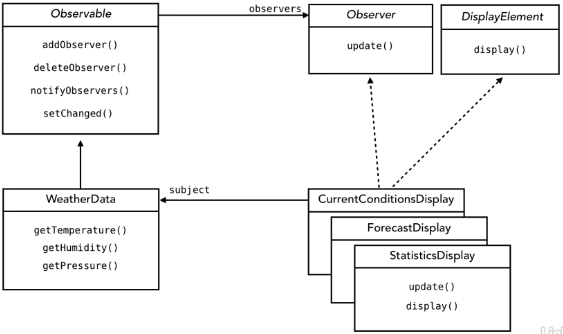


Implementing the observer pattern

* In the Subject interface, registerObserver() and removeObserver() take in an object of type Observer.
* The update method in the Observer interface takes in all the data of the subject
  + Example (continued): it would take in temp, humidity, and pressure.
* In the ConcreteSubject class, the data is stored as fields. The constructor sets up an ArrayList of observers. registerObserver() and removeObserver() adds and removes from the ArrayList. notifyObservers() calls the update method for each of the Observer in the ArrayList, passing in all the ConcreteSubject data. All setters of the ConcreteSubject class call the notifyObservers() method.
* In each ConcreteObserver class, the data is stored as fields. The ConcreteObserver has a reference to the Subject, and this value is set in the constructor. The ConcreteObserver updates its fields in the update() method, and does whatever necessary changes
  + Example (continued): after updating its fields, each display would call a display() method to update its display.
* The observer pattern allows us to add new observers without changing the subject.

Java’s Observer and Observable Classes

* Java has built in Observable and Observer classes in the java.util package
  + The subject extends the built-in Observable class. There is a two-step process to notify observers.
  + The observer implements the built-in Observer interface. Observers can pull data from the Subject or the Subject can push data to the Observers
* Observable is now a class that implements the add, delete, and notify observer methods. The method setChanged is used in the notification process. WeatherData class only implements getters and a setter for its fields.



* Two steps to notify observers: call setChanged() method, which indicates the state has changed in the Subject. Then call notifyObservers():
  + For pull: call notifyObservers(). For push: notifyObservers(Object arg), where arg is any object you choose that contains the data.
  + When you call notifyObservers(), the update() method in the observers is called. The first parameter contains the subject (type Observable). The second parameter is the object you passed into arg if you using push.

Implementing the observer pattern with Java’s Observer and Observable classes

* The ConcreteObservable class contains a private field for each of its data. The setter method now calls setChanged() and notifyObservers() (which are extended from the Observable class). Good to have getters for each field.
* Each of the constructors of the ConcreteObserver classes takes in an Observable object, and calls the addObserver() method of the Observable object, passing inside itself. (The ConcreteObserver class can hold a reference to the Observable object if needed.) Each ConcreteObserver class overrides the update method, casting the Observable parameter to the ConcreteObservable class (and the Object class to the appropriate Data class if using the push method.) (Of course, first check it is the correct type before casting it.) The ConcreteObserver class now does whatever it wants with the new data.
* Observer design pattern principle: never count on the order in which observers are notified, or on how many observers you have. The Subject and Observers are loosely coupled.

The advantages of loose coupling

* Subject and observers are loosely coupled: they interact (and thus are coupled) but have little knowledge of each other (and thus loosely coupled)
  + Subject only knows that it implements a specific interface. Does not need to know the concrete class of the observer.
  + No changes to the subject or the observer will affect the other.
* Because object interdependencies are minimized, loose coupling allows us to build more flexible OO architectures.
* Design Principle #4: Strive for loosely coupled designs between objects that interact

**The Decorator Pattern**

Creating chaos with inheritance

* Example: we have a coffee shop with 4 beverages for sale (house blend, dark roast, decaf, and espresso), each with a description and cost. And you can add various condiments to each drink (steamed milk, mocha, soy, whip, etc.)
* Design attempt #1: have Beverage abstract class with description attribute and cost() method. Then have each concrete beverage extend Beverage, overriding the cost() method. But then you need subclasses for each combination of condiments for each beverage (e.g. HouseBlendWithSoyMocha). Class explosion…eww
* Design attempt #2: Beverage has a boolean attribute for each condiment (along with getters and setters). Seems better, but some problems still: the cost() method is still being overridden in each subclass, and so a price change could affect each existing class. New condiments require changing the superclass. Condiments are not appropriate for some beverages. And can’t handle orders like double mocha.

Understanding the open-closed principle

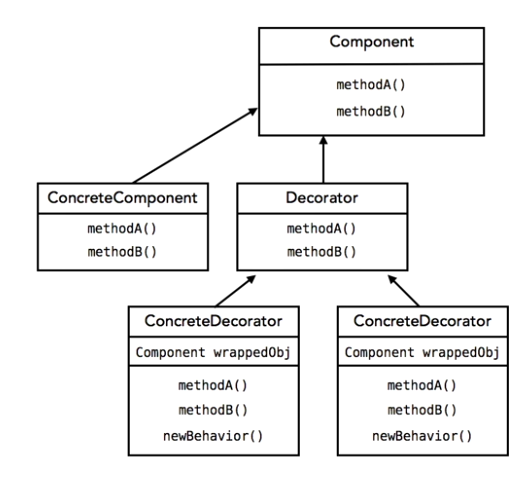
* Design principle #5 (the open closed principle): Classes should be open for extension but closed for modification.
  + Example (continued): when we add new types of coffee or condiments, we should not need to change existing code.
  + We want classes to be open to extensions of behavior, but we also want classes closed to modification. Our goal is to be able to easily augment what we have without modifying existing code.
  + This is one of the most important design principles.
* Remember the problems of inheritance and how composition can be better
  + Inheritance is powerful, but can lead to inflexible designs. When we subclass, all classes inherit the same behavior, resulting in static, inflexible, compile-time decisions.
  + With composition, we inherit behavior instead, allowing us to make dynamic runtime decisions. We can add new behavior without altering existing code, i.e. including behaviors not even considered by the creator. Results in fewer bugs and unintended side effects, and overall we get more flexible designs.

Extending behavior with composition

* Example (continued), Design Attempt #3: Go back to design attempt #1. But instead of using a specific class for the entire beverage, create a new object for each condiment and wrap it around existing classes. (So for a dark roast with mocha and whip, have the Mocha wrap around the Dark Roast, and have the whip wrap around the Mocha.) Each object implements cost(), which returns the sum of the cost of the object and the objects inside it. (So Whip’s cost() returns the sum of the cost of itself and the cost returned by Mocha’s cost(), and Mocha’s cost() returns the sum of itself and the cost returned by dark roast().)

The decorator pattern

* The decorator pattern attached additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
* In the example, the Components are the types of beverages, and the Decorators are the condiments. The Component class is an abstract class or interface, and each ConcreteComponent extends/implements it. The Decorator abstract class also implements the Component abstract class/interface, and each ConcreteDecorator extends the Decorator abstract class.
* The Decorator implements the Component because we want to treat each class in the same way. (We want to wrap any decorator around any of the components.)
  + In the example, we want to wrap all the condiments around all the coffees, and then call the cost() or description() methods on any of the objects



Implementing the decorator pattern

* Let method1(), … , methodN() be the methods implemented using composition. The Component contains either an abstract or default implementation of method1(), … , methodN(). Each ConcreteComponent needs to override at least the abstract methods, but does not rely on any of the Decorators to implement the methods. The Decorator extends the Component, and should make the methods with a default implementation in the Component class abstract in the Decorator class to require the ConcreteDecorators to reimplement those methods. Each ConcreteDecorator holds a reference to an object of type Component, which represents the object that the ConcreteDecorator wraps around. The constructor takes in an object of type Component and sets this reference to the parameter. Each ConcreteDecorator should implement method1(), … , methodN() by adding its own piece to the algorithm and then calling the same object method on the Component object that it has a reference to.
  + The caller uses these classes by creating a new ConcreteComponent object. Then, for each decorator that the caller wants to add, the caller creates a new object of type ConcreteDecorator and passes in the current object. After this step, method1(),…,methodN() should work properly.
* Example (continued): the Beverage class contains a string description with default value of “Unknown Beverage”. The two composition methods are getDescription() and cost(). getDescription() contains default implementation of returning the string description, and cost() is abstract. For each ConcreteBeverage, the constructor sets the description to the name of the beverage, and overrides the cost() method, which returns the cost of that beverage without any condiments. The CondimentDecorator class contains an abstract method getDescription() to require each ConcreteCondiment to reimplement it. Each ConcreteCondiment has a reference to a Beverage, and sets this value in the constructor (which takes in an object of type Beverage). Each ConcreteCondiment overrides the getDescription() method, returning “beverage.getDescription() + *“*, *NameOfCondiment*”. The cost() method is also overridden, which returns the cost of the condiment added onto the cost returned by the cost() method of the beverage.
  + The caller creates a new ConcreteBeverage and stores a reference to it of type Beverage. Then for each ConcreteCondiment that the caller wants to add to the beverage, the caller creates a new object of type ConcreteCondiment, and reassigns it back to the reference of type Beverage. Once finished, the caller can call getDescription() and cost() on the object. Example code:

Beverage beverage = new HouseBlend();

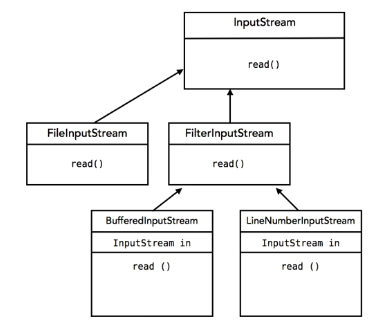
beverage = new Soy(beverage);

beverage = new Mocha(beverage):

System.out.println(beverage.getDescription() + “ $” + beverage.cost());

Understanding decorators in Java libraries

* Some classes in Java use the decorator pattern
* FileInputStream (java.io) can be used to read data from a text file with read(). You can decorate this component with several decorators:
  + The BufferedInputStream buffers input to improve performance
  + The LineNumberInputStream counts line numbers as it reads data



Using java.io decorators

* This is an example of how you can implement a decorator for file input. We will create a Decorator to convert all the characters to lower case.
* Create a class called LowerCaseInputStream that extends the FilterInputStream (which is the Decorator). The constructor takes in an InputStream (the Component) and calls the superclass’s constructor (passing in the InputStream), which saves the InputStream as a field. Create a read() method that takes the next character of the InputStream and converts it to lower case (using Character.toLowerCase()).

**The Singleton Pattern**

What is the singleton pattern?

* The Singleton Pattern ensures a class has only once instance, and provides a global access point to it.
* Uses: anywhere you want to ensure a resource exists only once (e.g. connection and thread pools, logging facilities, preference and registry objects)
* If you had more than one object, you could end up with inconsistent or incomplete results, causing app/system to crash.
* You could just instantiate a normal object only once, but unlike the Singleton Pattern, there is no code that is preventing other objects from instantiating it multiple times.

Understanding the classic singleton pattern

* Create the Singleton class, and make its constructor (which does nothing) private. Thus, only code that exists inside this class can instantiate it.
* Incorrect Attempt: Create a static method called getInstance() that returns a new Singleton. But this method can be called multiple times, resulting in multiple instances of the Singleton object.
* Solution: Create a field that holds a reference to a Singleton object. The getInstance() method checks if this field is null. If it is, it creates a new Singleton object. assigns it to this field, and returns the object. If not, then it just returns the object.

Implementing the classic singleton pattern

* public class Singleton {

private static Singleton uniqueInstance;

private Singleton() {}

public static Singleton getInstance() {

if (uniqueInstance == null) {

uniqueInstance = new Singleton();

}

return uniqueInstance;

}

}

* Lazy instantiation: instantiation that only occurs if and when it’s really needed. This is done in the Singleton pattern by checking if uniqueInstance is null before instantiating it.
* The Singleton class can contain various other methods once the above code is copied.
* To use the Singleton class, use the getInstance() method, and then call whatever methods you want that the Singleton class has.
* Note, this is not thread safe!

Dealing with multithreading

* Multiple threads can mess up the Singleton pattern.
* How to fix it: make sure the getInstance() method is synchronized is some way, or create a Singleton using a way other than lazy instantiation

Improving the singleton pattern implementation

* Initialize the Singleton instance as soon as the class is loaded by the JVM before any thread can access the private variable:

public class Singleton {

private static Singleton uniqueInstance = new Singleton();

private Singleton() {}

public static Singleton getInstance() {

return uniqueInstance;

}

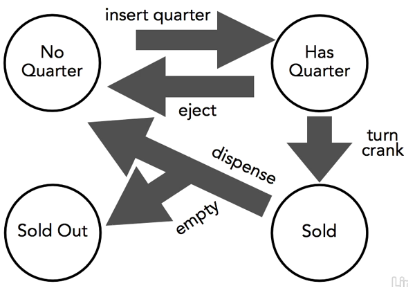
}

* + Disadvantage: we are creating it eagerly instead of lazily: the Singleton is always created, even when not needed
* Use the synchronized keyword in the getInstance() method.
  + Take the existing code and add the keyword “synchronized” in the getInstance() method header: public static synchronized Singleton getInstance() {
  + Only one thread can access this at a time.
  + Disadvantage: synchronization is expensive. Every call to getInstance() is expensive. But advantage of lazy instantiation.

**The State Pattern**

What is a state machine?

* State machine: code based on a set of states and transitions
* Example, state of gumball machine



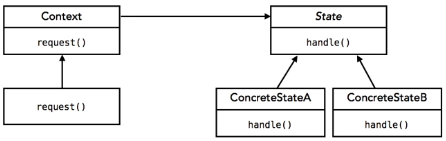
* Implementation attempt #1:
  + static int for each state (e.g. SOLD\_OUT = 0), and an int that holds onto the current state.
  + Then for each transition, write a method (e.g. insertQuarter()). Need a switch block that handles each of the current states you may be in.
  + Note that this requires a lot of code to handles states that doesn’t make sense (e.g. trying to insertQuarter() when you are already in the Has Quarter state.)
  + Adding another state and set of transitions will result in a maintenance nightmare.

Revisiting the design for a state machine

* Problems with implementation attempt #1: is not really object-oriented at all, any additions require many changes to code, difficult to understand all the states and transitions, and violates open-closed principle.
* Design attempt #2 (the solution):
  + Create an object for each state. Each ConcreteState object implements a State interface that contains a method for each state transition.
  + Adding a new state requires just creating a new ConcreteState class. Adding another transition just involves adding a new method to the existing classes and states.

Understanding the state pattern

* We have an object that represents the entire state machine (we’ll call this StateMachine). Instead of using integers to represent states, we use state objects. Look at “Design attempt #2” above to see how the state objects are created.
  + With this implementation, a particular transition method in the StateMachine, which will call the same transition method in the current state object.
* Each ConcreteState handles what needs to be done for a particular transition instead of the State interface.



* The Context represents the StateMachine.
* The Context can delegate the request to any of the states without knowing how the states are implemented, so there is loose coupling between the Context and the State.
* The State Pattern allows an object to alter its behavior when its internal state. The object will appear to change its class.
  + The pattern encapsulates state into separate classes.
  + The context delates to the current state to handle requests.
  + Once a request is handled, the current state may change (and each state has a different behavior, causing the object to appear to change its class).
* By following the state machine pattern, we are encapsulating what varies, favoring composition over inheritance, and keeping a class closed for modification but open for extension.

Implementing the state pattern

* The StateMachine contains a reference to each of the State objects. These State objects are created within the constructor of the StateMachine. The StateMachine also has a field that holds a reference to the object representing the current state. Each of the methods representing transitions in the StateMachine delegate the task to the current state object. Add getters for each of the state objects and a setter for the current state.
* The State interface contains a method for each transition method.
* Each ConcreteState implements the State interface (and thus implements all the methods in the State interface). It holds a reference to the StateMachine object, which is initialized during the constructor. To change the state of the StateMachine object, call the setter for the current state.
* Sometimes the State machine may need to call multiple methods from the current state if a transition immediately follows another transition (i.e. you stay in a particular state only for a fixed amount of time)
* **Example** (continued)
  + We have a GumballMachine class which represents the entire state machine. The GumballMachine class has four fields of type State (e.g. soldOutState), each of which is a reference to an object that represents one of the four states. This class also has a current state field and a field that represents the count of gumballs. The constructor for this class initializes the four state fields, the current state, and the count. (Example: soldOutState = new SoldOutState(this);) Getters for each state field and setter for the current state.
  + The State interface contains insertQuarter(), ejectQuarter(), etc.
  + There are four subclasses of State to represent the four different types of states (e.g. SoldOutState). It holds a reference to the GumballMachine, which is set in the constructor. It then implements each of the methods in the State interface.

Comparing the state and strategy patterns

* The state pattern and strategy pattern are similar in the following way we’re using composition to determine runtime behavior.
* But the intents are different. With state pattern, we are modeling a system by stepping through different states in a predetermined order (through transitions), thus changing its behavior all the time. With strategy pattern, the intent is to assign a behavior to a class and then stay with it.

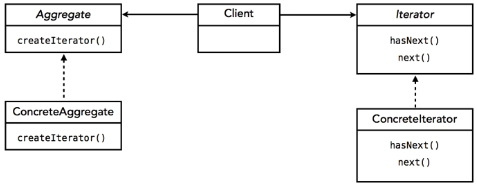
**The Collection Pattern**

Encapsulating iteration

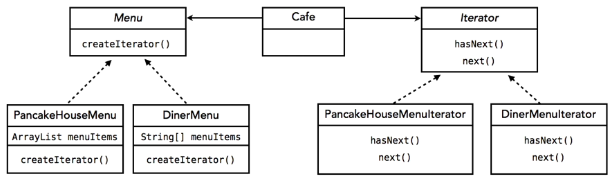
* The iterator pattern allows us to separate what varies and encapsulate it with regards to collections
* It provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Exploring the iterator pattern

* Example: we got two menus with two different implementations. One uses an array, while the other uses an ArrayList. The Café needs to know the underlying implementation of those menus to access the choices in the menu. If someone decides later to change how one of the menus is implemented, that would break the code in the Café. Can reduce the dependency between the Café and the menus by using The Iterator Pattern
* The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation
  + The user asks the collection object for its iterator using createIterator(), and then the user uses the iterator to iterate through the items in the aggregate using hasNext() and next(). The iteration code works with any kind of aggregate object.



* Example (continued): The Aggregate is Menu, and each ConcreteMenu is a ConcreteAggregate. The client obtains an iterator by calling createIterator on the menu, and then using the methods hasNext() and next() to traverse through the elements.



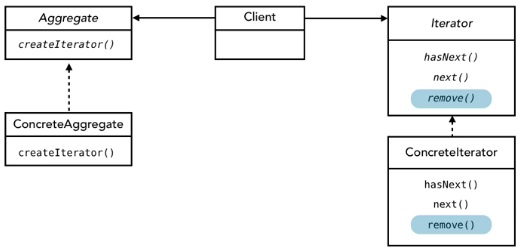
* Design Principle #6 (the single-responsibility principle): a class should have only one reason to change
  + The Iterator Pattern reflects this principle since if the way we are iterating changes, neither the Café nor the menu has to change; we simply change the way the iteration works in the iterator object.

Implementing the iterator pattern

* Each ConcreteAggregate implements the Aggregate class (which contains the method createIterator(), returning an Iterator) and has a collection of data. It has a createIterator() method that returns a new ConcreteIterator for the ConcreteAggregate’s data.
  + This is an example of programming to an interface, not to an implementation.
* The ConcreteIterator implements the Iterator interface, and thus implements the methods next() and hasNext(). next() returns an Object, and hasNext() returns a boolean.
* The user of these objects will call the *concreteAggregateName*.createIterator() method, storing the iterator in type Iterator (NOT ConcreteIterator). The user can then call *iteratorName*.next() and *iteratorName*.hasNext();
* Example (continued)
  + The DinerMenu, which uses an array to store the collection of choices, returns a new DinerMenuIterator for the createIterator() method. The DinerMenuIterator keeps track of the index of the next item to return when next() is called.
  + The PancakeHouseMenu uses an ArrayList to store the choices. It returns a new PancakeHouseMenuIterator The PancakeHouseMenuIterator keeps track of the index of the next item to return when next() is called.
  + The user can create an iterator for DinerMenu by calling dinerMenu.createIterator(), storing the iterator in a variable of type Iterator (NOT DinerMenuIterator). The user can then use this iterator, calling hasNext() or next() on it, to retrieve the elements.

Using Java’s built-in iterators

* Java offers a built-in iterator interface in java.util.Iterator and java.util.Iterable.
* Used as an interface for creating your own iterators and used as the type for built-in collection iterators (e.g. ArrayList, Vector, LinkedList)
* Iterable has a method called iterator() that returns a java.util.Iterator instance.
* In addition to next() and hasNext(), Iterator specifies a remove() method which removes the last item returned by the next() item.



Implementing with Java’s built-in iterators

* Make your collection implement the Iterable interface.
* If the backing data structure you use implements iterable, then you can just call the iterator() method on that data structure. Otherwise, you need to create your own custom ConcreteIterator class that implements the Iterator interface.
* Note that iterator() turns an Iterator with a type parameter (Iterator<E>)

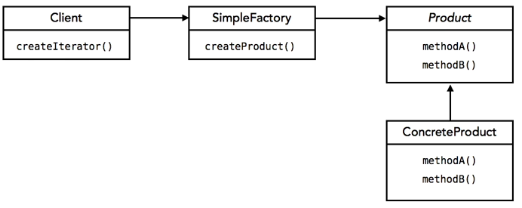
Making sure classes have only one responsibility

* Every responsibility increases chance of change. Thus, if you have two responsibilities, that is two areas of potential change. But we want to avoid change whenever possible.
* Results in design principle #6: a class should have only one reason to change
* The problematic way of iterating through the collection of data is to allow a class to handle both the responsibilities of managing a collection and the iteration responsibilities. That’s why we pull the Iterator responsibility into its own class.
* We like to combine things, finding commonalities. But we need to look for multiple responsibilities and separate them.

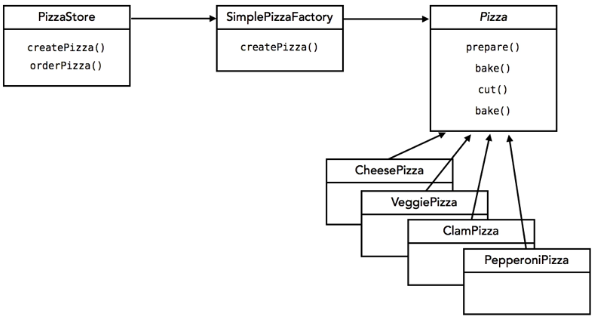
**Factories**

Encapsulating object creation

* When we use the “new” operator for creating a new ConcreteClass, we get locked into concrete types. This can result in code changes, violating our open-closed principle
  + Example: we are programming orderPizza(), passing in the string type of the pizza. If the pizza type is “cheese”, we create a new CheesePizza. If “pepperoni”, we create a new PepperoniPizza. But what if we create or delete a pizza type? Then we have to change existing code.
* So instead, we create a Factory class that is responsible for creating a new GeneralClass based on various conditions. This factory has a method called createGeneralClass, passing in whatever information is needed to determine which ConcreteClass is created. This returns an object of type GeneralClass.



* + Example (continued): we have a PizzaFactory class with method createPizza that has parameter of string called “type”, and creates the specific type of pizza based on the string parameter, or null if a particular type is not recognized. (For example, if pizza type is “cheese”, we return a new CheesePizza()). The return type of this method is Pizza.



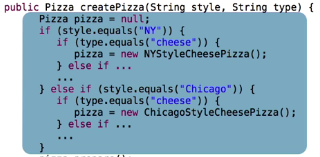
* All changes involving adding and removing ConcreteClasses are now moved to the Factory instead of the code trying to use the class.

Implementing the Simple Factory idiom

* (Look at the previous section, “Encapsulating object creation”, to see how to create the Factory class)
* It is easy to use a different type of Factory to create a different type of object.
  + Example (continued): if we have one factory create EastCoastPizzas and one factory create WestCoastPizzas, it is easy to switch the factory we use in the code for the pizza store to switch to a different type of pizza.
* The user of this Factory first instantiates the Factory. Then he/she calls the create*ObjectName*() method of the Factory, passing in whatever information is needed to know which concreteObject to create. The user can then do whatever he/she wants with this object created.

Exploring the factory method pattern

* We want to be able to use the same algorithm to create a new object, but create a different type of object.
  + Example (continued): We want to be able to orderPizza() the same way, but create different types of pizza (e.g. NY vs. Chicago Pizza). This is BAD code:



* The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factor Method lets a class defer instantiation to subclasses.
* We now have an abstract class for the type of user of the factory. This abstract class has a defined implementation for the procedures we want to keep the same. Each different type of user extends this abstract class, implementing the method that instantiates the object differently.
  + Example (continued): We have abstract class PizzaStore that has defined implementation for orderPizza() but abstract method createPizza(). The NYStylePizzaStore extends this abstract class and overrides the createPizza method. If the type is “cheese”, then it returns a new NYStyleChessePizza(). Does something similar for other types of pizza. The ChicagoStylePizzaStore does the same thing as the NYPizzaStore except it creates the corresponding Chicago-version of each pizza.
  + Notice that we don’t have a factory anymore because the pizza stores are creating them directly.
  + Pizzas can easily be added or deleted from each type of store if the menu changes.

**Conclusion**

Looking at what we’ve done and where to go from here

* We have covered the following design patterns: Strategy, which is a pattern for changing the behavior of an object. Observer, which gives us a way to allow objects to communicate in a loosely coupled manner. Decorator, which allows us to build up behavior through composition. Singleton, which ensures only one copy of an object exists. State, which changes behavior based on a set of state objects and transitions. Iterator, which encapsulates iteration. And Factory, which encapsulates creation.
* Other patterns in the gang-of-four book: command pattern, adapter pattern, façade program, template method pattern, composite pattern, proxy pattern, and 9 more less-used ones.
* Other types of pattern include domain specific patterns like enterprise design patterns.
* Compound patterns are built from two-or-more patterns. MVC pattern (combines strategy, observer, and composite pattern) to separate the UI, logic, and data model
* Design principles: Encapsulate what varies, favor composition over inheritance, program to an interface, strive for loosely coupled designs, classes should remain open for extension and closed for modification. And, a class should have only one reason to change.
* Definition of a design pattern: a solution to a problem in a context
* Using patterns: don’t make patterns a hammer looking for nails; everything is a trade-off.
  + Save patterns for the places where you code is really likely to change, or else you are adding needless complexity to your designs.
  + KISS: keep designs simple.
  + But when you do see a need, do not reinvent, use patterns
* Do not stop learning patterns

Additional resources

* Books:
  + “Head First Design Patterns”
  + “Design Patterns: Elements of Reusable Object Oriented Software”
* There are hundreds of patterns, so don’t try to learn them all. Memorize and work with a few core ones.
* Additional Lynda courses
  + Foundations of programming: object-oriented design
  + Java Essential/Advanced Training
  + Foundations of Programming: Refactoring Code (helpful for refactoring code to use a certain design principle.)