DESIGN PATTERNS

* THE STRATEGY PATTERN
  + In computer programming, the strategy pattern (also known as the policy pattern) is a behavioral software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use.[1]
  + Strategy lets the algorithm vary independently from clients that use it.[2] Strategy is one of the patterns included in the influential book Design Patterns by Gamma et al.[3] that popularized the concept of using design patterns to describe how to design flexible and reusable object-oriented software. Deferring the decision about which algorithm to use until runtime allows the calling code to be more flexible and reusable.
  + For instance, a class that performs validation on incoming data may use the strategy pattern to select a validation algorithm depending on the type of data, the source of the data, user choice, or other discriminating factors. These factors are not known until run-time and may require radically different validation to be performed. The validation algorithms (strategies), encapsulated separately from the validating object, may be used by other validating objects in different areas of the system (or even different systems) without code duplication.
  + Typically the strategy pattern stores a reference to some code in a data structure and retrieves it. This can be achieved by mechanisms such as the native function pointer, the first-class function, classes or class instances in object-oriented programming languages, or accessing the language implementation's internal storage of code via reflection.
* THE OBSERVER PATTERN
  + Push vs pull relationship
  + The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.
  + It is mainly used to implement distributed event handling systems, in "event driven" software. Most modern languages such as C# have built-in "event" constructs which implement the observer pattern components.
  + The observer pattern is also a key part in the familiar model–view–controller (MVC) architectural pattern.[1] The observer pattern is implemented in numerous programming libraries and systems, including almost all GUI toolkits.
  + The Observer pattern addresses the following problems:[3]
  + A one-to-many dependency between objects should be defined without making the objects tightly coupled.
  + It should be ensured that when one object changes state an open-ended number of dependent objects are updated automatically.
  + It should be possible that one object can notify an open-ended number of other objects.
  + Defining a one-to-many dependency between objects by defining one object (subject) that updates the state of dependent objects directly is inflexible because it couples the subject to particular dependent objects. Tightly coupled objects are hard to implement, change, test, and reuse because they refer to and know about (how to update) many different objects with different interfaces.
  + What solution does the Observer design pattern describe?[edit]
  + Define Subject and Observer objects.
  + so that when a subject changes state, all registered observers are notified and updated automatically.
  + The sole responsibility of a subject is to maintain a list of observers and to notify them of state changes by calling their update() operation.
  + The responsibility of observers is to register (and unregister) themselves on a subject (to get notified of state changes) and to update their state (synchronize their state with subject's state) when they are notified.
  + This makes subject and observers loosely coupled. Subject and observers have no explicit knowledge of each other. Observers can be added and removed independently at run-time.
  + This notification-registration interaction is also known as publish-subscribe.
  + See also the UML class and sequence diagram below.
* THE DECORATOR PATTERN
  + Wrap the wrapper in a wrapper
  + In object-oriented programming, the decorator pattern is a design pattern that allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class.[1] The decorator pattern is often useful for adhering to the Single Responsibility Principle, as it allows functionality to be divided between classes with unique areas of concern.[2] The decorator pattern is structurally nearly identical to the chain of responsibility pattern, the difference being that in a chain of responsibility, exactly one of the classes handles the request, while for the decorator, all classes handle the request.
  + he Decorator [3] design pattern is one of the twenty-three well-known GoF design patterns that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.
  + What problems can it solve?[edit]
  + Responsibilities should be added to (and removed from) an object dynamically at run-time.[4]
  + A flexible alternative to subclassing for extending functionality should be provided.
  + When using subclassing, different subclasses extend a class in different ways. But an extension is bound to the class at compile-time and can't be changed at run-time.
  + What solution does it describe?[edit]
  + Define Decorator objects that
  + implement the interface of the extended (decorated) object (Component) transparently by forwarding all requests to it and
  + perform additional functionality before/after forwarding a request.
  + This enables to work through different Decorator objects to extend the functionality of an object dynamically at run-time.
  + See also the UML class and sequence diagram below.
* THE FACTORY PATTERN
  + In class-based programming, the factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor.
  + The Factory Method [1] design pattern is one of the twenty-three well-known "Gang of Four" design patterns that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.
  + The Factory Method design pattern solves problems like: [2]
  + How can an object be created so that subclasses can redefine which class to instantiate?
  + How can a class defer instantiation to subclasses?
  + Creating an object directly within the class that requires (uses) the object is inflexible because it commits the class to a particular object and makes it impossible to change the instantiation independently from (without having to change) the class.
  + The Factory Method design pattern describes how to solve such problems:
  + Define a separate operation (factory method) for creating an object.
  + Create an object by calling a factory method.
  + This enables writing of subclasses to change the way an object is created (to redefine which class to instantiate).
  + See also the UML class diagram below.
* THE SINGLETON PATTERN
  + n software engineering, the singleton pattern is a software design pattern that restricts the instantiation of a class to one "single" instance. This is useful when exactly one object is needed to coordinate actions across the system. The term comes from the mathematical concept of a singleton.
  + Critics consider the singleton to be an anti-pattern in that it is frequently used in scenarios where it is not beneficial, introduces unnecessary restrictions in situations where a sole instance of a class is not actually required, and introduces global state into an application.
  + he singleton[4] design pattern is one of the twenty-three well-known "Gang of Four" design patterns that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.
  + The singleton design pattern solves problems like:[5]
  + How can it be ensured that a class has only one instance?
  + How can the sole instance of a class be accessed easily?
  + How can a class control its instantiation?
  + How can the number of instances of a class be restricted?
  + The singleton design pattern describes how to solve such problems:
  + Hide the constructor of the class.
  + Define a public static operation (getInstance()) that returns the sole instance of the class.
  + The key idea in this pattern is to make the class itself responsible for controlling its instantiation (that it is instantiated only once).
  + The hidden constructor (declared private) ensures that the class can never be instantiated from outside the class.
  + The public static operation can be accessed easily by using the class name and operation name (Singleton.getInstance()).
* THE COMMAND PATTERN
  + In object-oriented programming, the command pattern is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time. This information includes the method name, the object that owns the method and values for the method parameters.
  + Four terms always associated with the command pattern are command, receiver, invoker and client. A command object knows about receiver and invokes a method of the receiver. Values for parameters of the receiver method are stored in the command. The receiver object to execute these methods is also stored in the command object by aggregation. The receiver then does the work when the execute() method in command is called. An invoker object knows how to execute a command, and optionally does bookkeeping about the command execution. The invoker does not know anything about a concrete command, it knows only about the command interface. Invoker object(s), command objects and receiver objects are held by a client object, the client decides which receiver objects it assigns to the command objects, and which commands it assigns to the invoker. The client decides which commands to execute at which points. To execute a command, it passes the command object to the invoker object.
  + Using command objects makes it easier to construct general components that need to delegate, sequence or execute method calls at a time of their choosing without the need to know the class of the method or the method parameters. Using an invoker object allows bookkeeping about command executions to be conveniently performed, as well as implementing different modes for commands, which are managed by the invoker object, without the need for the client to be aware of the existence of bookkeeping or modes.
  + The Command [1] design pattern is one of the twenty-three well-known GoF design patterns that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.
  + What problems can the Command design pattern solve? [2]
  + Coupling the invoker of a request to a particular request should be avoided. That is, hard-wired requests should be avoided.
  + It should be possible to configure an object (that invokes a request) with a request.
  + Implementing (hard-wiring) a request directly into a class is inflexible because it couples the class to a particular request at compile-time, which makes it impossible to specify a request at run-time.
  + What solution does the Command design pattern describe?
  + Define separate (command) objects that encapsulate a request.
  + A class delegates a request to a command object instead of implementing a particular request directly.
  + This enables one to configure a class with a command object that is used to perform a request. The class is no longer coupled to a particular request and has no knowledge (is independent) of how the request is carried out.
  + See also the UML class and sequence diagram below.
* THE ADAPTER AND FACADE PATTERN
  + The Facade Pattern wiki page has a brief note about this.
  + "An Adapter is used when the wrapper must respect a particular interface and must support a polymorphic behavior. On the other hand, a facade is used when one wants an easier or simpler interface to work with."
  + I heard an analogy that you should think of your universal remote control that you've set up to work with all your different stereo systems - you press "on" and it turns on your cable box, your receiver, and your TV. Maybe it's a really fancy home theater and it dims the lights and draws the shades too. That's a Facade - one button/function that takes care of a more complicated set of steps.
  + The Adapter pattern just links two incompatible interfaces.
  + EDIT: A quick analogy for the Adapter pattern (based on the comments) might be something like a DVI-to-VGA adapter. Modern video cards are often DVI, but you've got an old VGA monitor. With an adapter that plugs into your video card's expected DVI input, and has its own VGA input, you'll be able to get your old monitor working with your new video card.
  + The facade pattern (also spelled façade) is a software-design pattern commonly used in object-oriented programming. Analogous to a facade in architecture, a facade is an object that serves as a front-facing interface masking more complex underlying or structural code. A facade can:
  + improve the readability and usability of a software library by masking interaction with more complex components behind a single (and often simplified) API
  + provide a context-specific interface to more generic functionality (complete with context-specific input validation)
  + serve as a launching point for a broader refactor of monolithic or tightly-coupled systems in favor of more loosely-coupled code
  + Developers often use the facade design pattern when a system is very complex or difficult to understand because the system has a large number of interdependent classes or because its source code is unavailable. This pattern hides the complexities of the larger system and provides a simpler interface to the client. It typically involves a single wrapper class that contains a set of members required by the client. These members access the system on behalf of the facade client and hide the implementation details.
  + The Facade [1] design pattern is one of the twenty-three well-known GoF design patterns that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.
  + What problems can the Facade design pattern solve? [2]
  + To make a complex subsystem easier to use, a simple interface should be provided for a set of interfaces in the subsystem.
  + The dependencies on a subsystem should be minimized.
  + Clients that access a complex subsystem directly refer to (depend on) many different objects having different interfaces (tight coupling), which makes the clients hard to implement, change, test, and reuse.
  + What solution does the Facade design pattern describe?
  + Define a Facade object that
  + implements a simple interface in terms of (by delegating to) the interfaces in the subsystem and
  + may perform additional functionality before/after forwarding a request.
  + This enables to work through a Facade object to minimize the dependencies on a subsystem.
  + See also the UML class and sequence diagram below
  + In software engineering, the adapter pattern is a software design pattern (also known as wrapper, an alternative naming shared with the decorator pattern) that allows the interface of an existing class to be used as another interface.[1] It is often used to make existing classes work with others without modifying their source code.
  + An example is an adapter that converts the interface of a Document Object Model of an XML document into a tree structure that can be displayed.
  + The adapter[2] design pattern is one of the twenty-three well-known GoF design patterns that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.
  + The adapter design pattern solves problems like:[3]
  + How can a class be reused that does not have an interface that a client requires?
  + How can classes that have incompatible interfaces work together?
  + How can an alternative interface be provided for a class?
  + Often an (already existing) class can't be reused only because its interface doesn't conform to the interface clients require.
  + The adapter design pattern describes how to solve such problems:
  + Define a separate adapter class that converts the (incompatible) interface of a class (adaptee) into another interface (target) clients require.
  + Work through an adapter to work with (reuse) classes that do not have the required interface.
  + The key idea in this pattern is to work through a separate adapter that adapts the interface of an (already existing) class without changing it.
  + Clients don't know whether they work with a target class directly or through an adapter with a class that does not have the target interface.
  + See also the UML class diagram below.
* WHAT IS A CLASS
  + A class is a blueprint
  + Fields (instance variables
    - What an object knows
  + Methods (Functions)
    - What an object does
* WHAT IS INHERITENCE
  + What do classes have in common
  + Abstract out those features
  + Override or extend methods that don’t work
  + Fields and Methods are already defined when you extend the super class
  + You define only the changes
  + Main is only to create objects and to interact
* HOW TO HIDE THE DATA
  + Fields (Instance variables are private)
  + Getters and setters methods are public (Getter = Accessors | Setter = Mutator)
* INSTANCE AND LOCAL VARIABLES
  + Fields (Instance Variables) are declared in the class
  + Local Variables are declared in a method
* Is A? Versus Has A?
  + Is A? Helps you decide if a class should extend another
    - Is a “Dog” an “Animal”?
    - Is a “Dog” a “Cat”
  + Has A? Helps you decide if something is a field
    - “Dog” has a “Height”
* WHEN TO USE INHERITENCE
  + The subclass is a superclass “Dog is an animal”
  + When a subclass needs most of the methods in a superclass
    - Almost every method in Animal is used in Dog
  + Don’t: use inheritance just to reuse code, or they or they don’t have a “is a” relationship
  + Avoid duplicate code
  + Changes to the superclass code is instantly reflected in subclasses
  + User knows that all subclasses have all of the methods of the superclass
* WHAT IS POLYMORPHISM?
  + Polymorphism allows you to write methods that don’t need to change if new subclasses are created
  + Dog can add a new method without changing animal
  + If dog wants to override a method, it can without effecting animal
* WHATS AN ABSTRACT CLASS
  + if you want the power of polymorphism without the work
  + abstract public class Creature
  + public abstract void setName();
  + There are no abstract fields
  + All methods don’t have to be abstract
  + You can have static methods
* WHATS AN INTERFACE
  + A class with only abstract methods
  + You can add as many interfaces to a class using implements as you want
  + YOU can only use public static and final fields
  + Interfaces provide the ultimate in flexibility

Classes from different inheritance trees can use a common interface