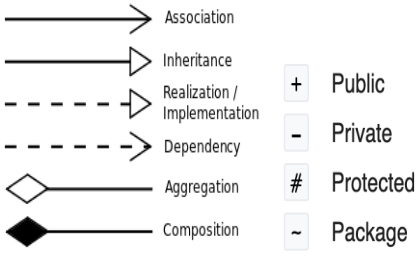
Design Patterns

- **Factory Pattern**
  - 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.
    - Depending on the type of information given to the factory method, it may use a switch statement.
  - Allows the user to create new objects without having to know the details of how they're created, or what their dependencies are - they only have to give the information they actually want.
- **Builder Pattern**
  - Separates the construction of a complex object from its representation. It is used to construct a complex object step by step and the final step will return the object.
  - The builder typically replaces the constructor for an object (making it private), and offers many simple methods for setting various attributes and specifying how to create the object, and one method for putting it all together and building the actual object desired.
- **Singleton Pattern**
  - A 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.
  - Commonly used for the creating the abstract factory, or builders.
- **Command Pattern**
  - 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.
  - Typically involves an ICommand interface with a abstract method for executing the command, and a invoker object which is responsible for calling this execution command.
- **Adapter Pattern**
  - A design pattern that allows the interface of an existing class to be used as another interface. It is often used to make existing classes work with others without modifying their source code and so that it matches what the client is expecting.
    - 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.
  - This pattern may either extend the adaptee class, avoid it, or implement its interface (best option probably). Either way, the adaptor should ALWAYS implement the target interface.
- **Strategy Pattern**
  - A 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.
  - While in many ways similar to command pattern, the strategy pattern relies on a different relationship between the context and the strategy (command). In this case, the context holds onto (has a field) for the strategy which it is initialized with or set to have, and contains a method with some parameters for evoking the strategy, typically with the same parameters.
- **Decorator Pattern**
  - 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.
  - This is achieved by designing a new *Decorator* class that wraps the original class. This wrapping could be achieved by the following sequence of steps:
    - Subclass the original *Component* class into a *Decorator* class (see UML diagram);
    - In the *Decorator* class, add a *Component* pointer as a field;
    - In the *Decorator* class, pass a *Component* to the *Decorator* constructor to initialize the *Component* pointer;
    - In the *Decorator* class, forward all *Component* methods to the *Component* pointer; and
    - In the *ConcreteDecorator* class, override any *Component* method(s) whose behavior needs to be modified.
    - Additional methods may be provided but then in addition to extending the base class, the decorator should implement another interface extending the original interface with the new method.

UML Diagrams




| Relationship | Depiction | Interpretation |
|---|---|---|
| Dependency | A --> B | A depends on B<br>This is a very loose relationship and so I rarely use it, but it's good to recognize and be able to read it. |
| Association | A → B | An A sends messages to a B<br>Associations imply a direct communication path. In programming terms, it means instances of A can call methods of instances of B, for example, if a B is passed to a method of an A. |
| Aggregation | A ◇→ B | An A is made up of B<br>This is a part-to-whole relationship, where A is the whole and B is the part. In code, this essentially implies A has fields of type B. |
| Composition | A ◆→ B | An A is made up of B with lifetime dependency<br>That is, A aggregates B, and if the A is destroyed, its B are destroyed as well. |

Design Patterns Examples

## Command Pattern

```
// Codify commands as function objects by implementing this interface or one like it
// Useful link: http://gameprogrammingpatterns.com/command.html
interface Command {
  public void execute(Human human);
}

// The command pattern nicely supports things like undoing
interface UndoableCommand extends Command {
  public void undo(Human human); // NOTE: you could also have the Human come from a constructor
  // and store it as state. This would prevent you from easily using lambdas instead of concrete
  // classes, though.
}

// Commands override that method, and any parameters that specific command may want would appear
// in the constructor.

class RunCommand implements Command {
  private int miles; // Any state for this command is stored in the class and set in the constructor

  public RunCommand(int miles) {
    this.miles = miles;
  }

  @Override
  public void execute(Human human) {
    human.say("I ran " + miles + " miles today!");
  }
}

class AgeCommand implements UndoableCommand {
  int prevAge;

  @Override
  public void execute(Human human) {
    prevAge = human.getAge();
    human.setAge(prevAge + 1);
  }

  @Override
  public void undo(Human human) {
    human.setAge(prevAge);
  }
}

class CommandMain {
  public static void main(String[] args) {
    Human person = Human.builder().build(); // Courtesy of our default builder

    // Make them run
    new RunCommand(10).execute(person); // > Hello world! and I ran 10 miles today!

    System.out.println("Age = " + person.getAge()); // > Age = 42
    UndoableCommand cmd = new AgeCommand();
    cmd.execute(person);
    System.out.println("Age = " + person.getAge()); // > Age = 43
    cmd.undo(person); // Fun stuff
    System.out.println("Age = " + person.getAge()); // > Age = 42
  }
}
```

## Decorator Pattern

```
// Component interface (like a JComponent)
interface Car {
  void assemble();
}

// At least one component implementation (like a JPanel)
class CarImpl implements Car {
  @Override
  public void assemble() {
    System.out.println("Basic car.");
  }
}

// Decorator - HAS-A relationship with component interface
class CarDecorator implements Car {
  protected Car car; // has-a

  public CarDecorator(Car car) {
    this.car = car;
  }

  @Override
  public void assemble() {
    this.car.assemble(); // This one just passes the method along - a useless decorator here
  }
}

// More interesting decorators (think JScrollPane, JGridPane)

class SportsCar extends CarDecorator {

  public SportsCar(Car c) {
    super(c);
  }

  @Override
  public void assemble() {
    super.assemble();
    System.out.print(" Adding features of Sports Car.");
  }
}

class LuxuryCar extends CarDecorator {

  public LuxuryCar(Car c) {
    super(c);
  }

  @Override
  public void assemble() {
    super.assemble();
    System.out.print(" Adding features of Luxury Car.");
  }
}
```

## Adapter Pattern

```
interface IDog {
  void bark();

  void goForAWalk();
}

class DogImpl implements IDog {

  @Override
  public void bark() {
    System.out.println("WOOF");
  }

  @Override
  public void goForAWalk() {
    System.out.println("walk walk walk walk");
  }
}

interface IHuman {
  void say(String msg);

  void openDoor();
}

class HumanImpl implements IHuman {

  @Override
  public void say(String msg) {
    System.out.println(msg);
  }

  @Override
  public void openDoor() {
    System.out.println("*Open door with opposable thumbs*");
  }
}

// IDog -> IHuman adapter
class DogPretendingToBeAHuman implements IDog, IHuman {
  IDog dog;

  public DogPretendingToBeAHuman(IDog dog) {
    this.dog = dog;
  }

  @Override // Human method implemented with dog methods
  public void say(String msg) {
    String[] words = msg.split("\\s+");
    for (String word : words) {
      bark();
    }
  }

  @Override // Human method implemented with dog methods
  public void openDoor() {
    System.out.println("*Scratch at door feebly*");
  }

  // Dog methods - can just pass them along since we have a real dog

  @Override
  public void bark() {
    dog.bark();
  }

  @Override
  public void goForAWalk() {
    dog.goForAWalk();
  }
}
```

## Strategy Pattern

```
// A Strategy is a function object
interface DanceStrategy {
  void go(HumanImpl human);
}

class BoxStep implements DanceStrategy {
  @Override
  public void go(HumanImpl human) {
    human.say("Forward-side-together, backwards-side-together.");
  }
}

class Moonwalk implements DanceStrategy {
  @Override
  public void go(HumanImpl human) {
    human.say("Lift heel, push, lift heel, push.");
  }
}

class DoTwo implements DanceStrategy {
  DanceStrategy first;
  DanceStrategy second;

  public DoTwo(DanceStrategy first, DanceStrategy second) {
    this.first = first;
    this.second = second;
  }

  @Override
  public void go(HumanImpl human) {
    first.go(human);
    second.go(human);
  }
}

class RandomMove implements DanceStrategy {
  DanceStrategy[] strategies;

  public RandomMove(DanceStrategy... strategies) {
    this.strategies = strategies.clone();
  }

  @Override
  public void go(HumanImpl human) {
    int index = new Random().nextInt(this.strategies.length);
    this.strategies[index].go(human);
  }
}

class Dancer extends HumanImpl {
  void dance(DanceStrategy danceStrategy) {
    danceStrategy.go(this);
  }
}
```