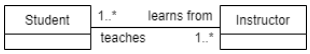
UML Class Diagrams

Classname
+ attribute : type
+ publicMethod() : type - privateMethod() : type # protectedMethod() : type

Association: Used when two models need to communicate between eachother.

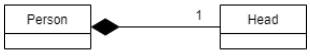


A student can have one or more instructors. An instructor can have one or more students.

Aggregation: Uses a "has a" relationship. The child can exist independently of the parent

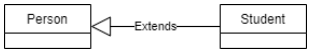
The professor has one or more classes to teach. A class has a professor.

Composition: Has a "part of" relationship. When a container is destroyed, the contents are also destroyed.

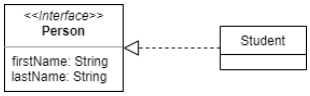


If person is deleted then head is also deleted as a result

Generalization/Inheritance: Has a "is a" relationship. When a subclass is a specialized form of the other

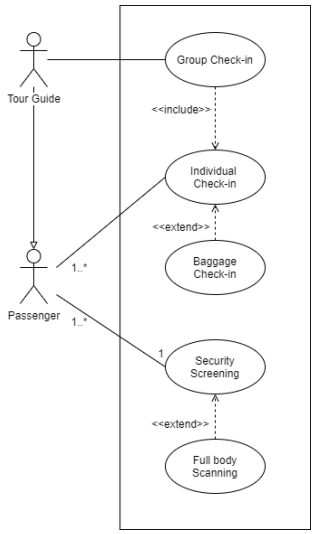


Realization/Implementation: A class implements implements the behavior that the interface specifies



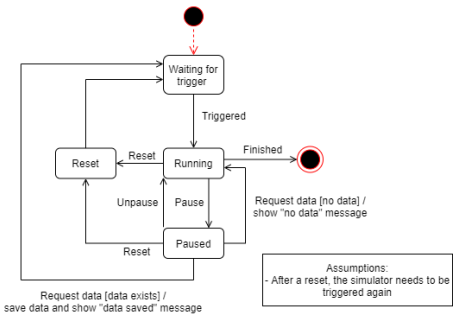
UML Use Case Diagram

Consider a system that deals with group tours. A tour guide has a task of group check-in, and that task involves a task of passenger check-in. When a passenger checks in, they may need to check in baggage. A passenger also has a task of being security screened, where they may need to submit to a full body scanner. A tour guide is also a passenger.



UML State Diagram

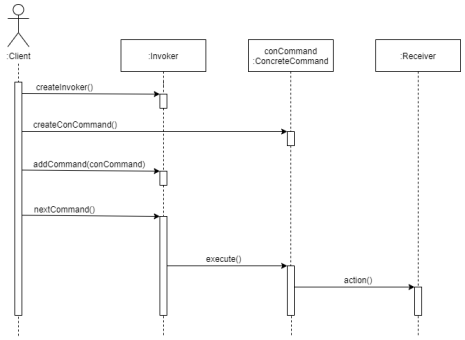
Consider a simulator for an experiment. When triggered to run, the simulator begins running. The simulator can be paused and un-paused. Data can be requested from the simulator when it is paused. If there is data, it can be stored, showing a "data saved" message, after which the simulator must be explicitly triggered to continue running. If there is no data, a "no data" message is shown, and the simulator automatically continues running. Also, a running or paused simulator can be fully reset.



UML Sequence Diagram

Consider the basic command design pattern, with Invoker, ConcreteCommand, and Receiver classes. Assume also there is a Client class to create a ConcreteCommand object and bind it to an Invoker object via dependency injection. Draw a complete and correct UML sequence diagram that depicts:

1. the command creation and its binding to an invoker, and
2. the behavior of the command pattern to have the invoker execute the command, which runs an action on a receiver



Singleton Pattern

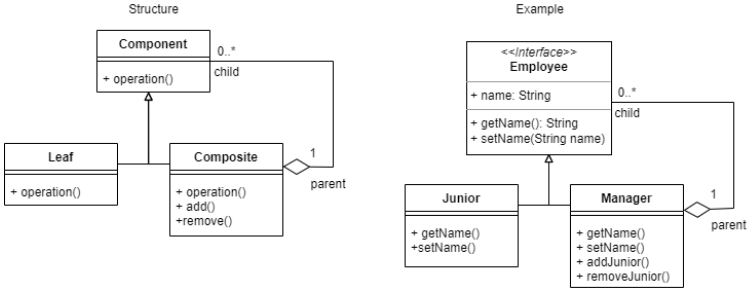
Intent: Ensure a class only has one instance and provide a global point of access to it

```
public class ExampleSingleton { // lazy construction
    private static ExampleSingleton instance = null;

    // protected constructor makes it possible to
    // create instances of subclasses
    protected ExampleSingleton() {
        ...
    }
    // lazy construction of the instance
    public static ExampleSingleton getInstance() {
        if (instance == null) {
            instance = new ExampleSingleton();
        }
        return instance;
    }
}
```

Composite Pattern

Intent: To compose individual objects to build up a tree structure (a folder can contains files and other folders). The individual objects and the composed objects are treated uniformly (files and folders both have a name)

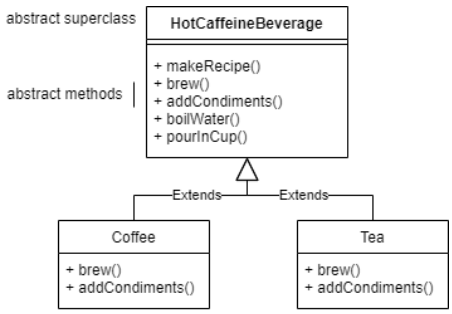


Command Pattern

Intent: encapsulate a request as an object, so you can run, queue, log, undo/redo these requests. Also known as action/-transaction. A class may want to issue a request without knowing anything about the operation being requested or the receiver object for the request

Template Method Pattern

Intent: Define the skeleton of an algorithm in a method, deferring some steps to subclasses. Used when two classes share a lot of common methods. It reduces duplication and enhances reuse.

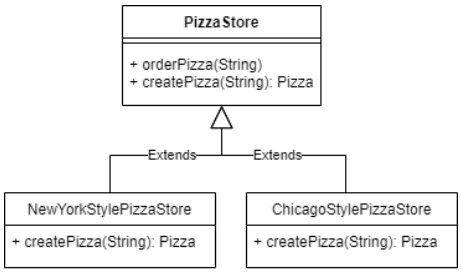



```
public abstract class HotCaffeineBeverage {
    // serves like a "template" for an algorithm,
    // where subclasses provide certain parts
    public final void makeRecipe() {
        boilWater();
        brew(); // from subclass
        pourInCup();
        addCondiments(); // from subclass
    }
    // let the subclasses determine how
    public abstract void brew();
    public abstract void addCondiments();
    public void boilWater() {
        System.out.println( "Boiling water" );
    }
    public void pourInCup() {
        System.out.println( "Pouring into cup" );
    }
}
```

```
// subclasses inherit
// makeRecipe, boilWater, pourInCup
public class Coffee extends HotCaffeineBeverage {
        public void brew() {
        System.out.println( "Brewing the coffee" );
        }
        public void addCondiments() {
        System.out.println( "Adding sugar, milk" );
        }
}
public class Tea extends HotCaffeineBeverage {
        public void brew() {
        System.out.println( "Steeping the tea" );
        System.out.println( "Removing the tea" );
        }
        public void addCondiments() {
        System.out.println( "Adding lemon" );
        }
}
```

Factory Method Pattern

Intent: Define an interface for creating an object, but lets subclasses decide which actual class to instantiate.



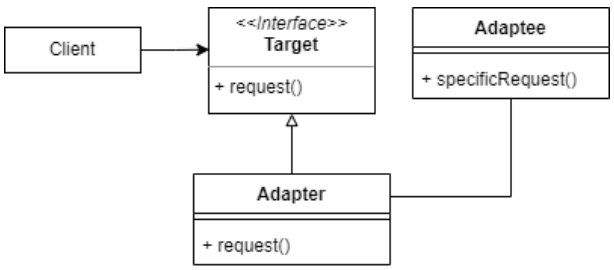

```
public abstract class PizzaStore {
    public Pizza orderPizza( String pizzaType ) {
        Pizza pizza;
        pizza = createPizza( pizzaType );

        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
    // defer to subclass to instantiate
    // Pizza of the appropriate type
    public abstract Pizza createPizza(String pizzaType);
}
```

```
public class NewYorkStylePizzaStore extends PizzaStore {
    public Pizza createPizza( String pizzaType ) {
        if (pizzaType.equals( "pepperoni" ) {
            Pizza pizza =
            new NewYorkStylePepperoniPizza();
        } else if (pizzaType.equals( "veggie" ) {
            Pizza pizza =
            new NewYorkStyleVeggiePizza();
        }
        return pizza;
    }
}
```

Adapter Pattern

Intent: convert the interface of a class into another interface that clients expect. This lets classes work together that couldn't otherwise because of incompatible interfaces. Also known as a wrapper.



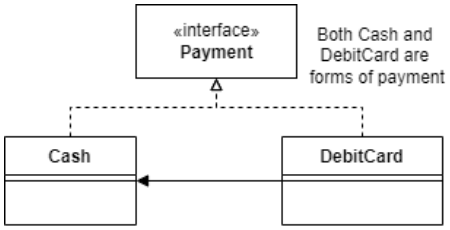

```
// target interface
public interface Duck {
    public void fly();
    public void quack();
}
```

```
// adaptee
public class Turkey {
    public void fly() { ... }
    public void gobble() { ... }
}
```

```
// adapter
public class TurkeyAdapter implements Duck {
    Turkey turkey;
    public TurkeyAdapter( Turkey turkey ) {
        this.turkey = turkey;
    }
    public void fly() {
        for (int i = 0; i < 5; i++) turkey.fly();
    }
    public void quack() {
        turkey.gobble();
    }
}
```

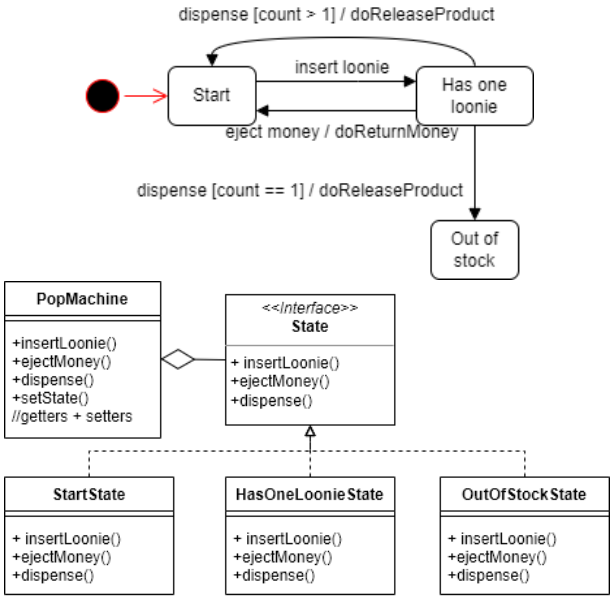
Proxy Pattern

Intent: to provide a surrogate or placeholder for another object to control access to it. It is used to defer the full cost of creation and initialization of an object until we actually need to use it



State Pattern

Intent: allow an object to alter its behavior when its internal state changes. To code a state model. For example, a simple pop machine where you can insert a loonie, press dispense button, and get a pop. We could eject to return money and the machine has a limited supply. The state diagram is the following:




```
// common interface for pop machine state classes
interface State {
    // all potential triggers
    public void insertLoonie( PopMachine popMachine );
    public void ejectMoney( PopMachine popMachine );
    public void dispense( PopMachine popMachine );
}
```

```
class StartState implements State {
    public void insertLoonie( PopMachine popMachine ) {
        System.out.println( "loonie inserted" );
        popMachine.setState(
        popMachine.getHasOneLoonieState());
    }
    public void ejectMoney( PopMachine popMachine ) {
        System.out.println( "no money to return" );
    }
    public void dispense( PopMachine popMachine ) {
        System.out.println( "payment required" );
    }
}
```

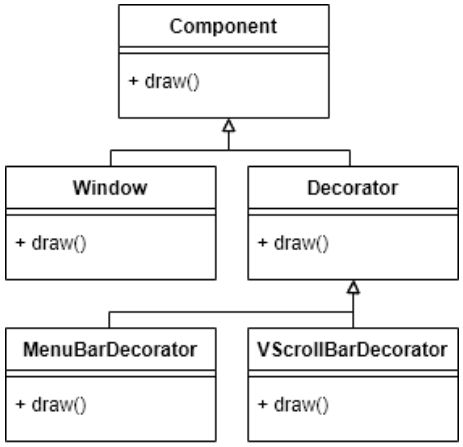
```
class HasOneLoonieState implements State {
    public void insertLoonie( PopMachine popMachine ) {
        System.out.println( "already have one loonie" );
    }
    public void ejectMoney( PopMachine popMachine ) {
        System.out.println( "returning money" );
        popMachine.doReturnMoney();
        popMachine.setState(
        popMachine.getStartState());
    }
    public void dispense( PopMachine popMachine ) {
        System.out.println( "releasing product" );
        popMachine.doReleaseProduct();
        if (popMachine.getCount() > 0) {
            popMachine.setState(
            popMachine.getStartState());
        } else {
            popMachine.setState(
            popMachine.getOutOfStockState());
        }
    }
}
```

```
class OutOfStockState implements State {
    public void insertLoonie( PopMachine popMachine ) {
        System.out.println( "machine out of stock" );
    }
    public void ejectMoney( PopMachine popMachine ) {
        System.out.println( "no money to return" );
    }
    public void dispense( PopMachine popMachine ) {
        System.out.println( "machine out of stock" );
    }
}
```

```
public class PopMachine {
    private State startState;
    private State hasOneLoonieState;
    private State outOfStockState;
    private State currentState;
    private int count;
    public PopMachine( int count ) {
        // make the needed states
        startState = new StartState();
        hasOneLoonieState = new HasOneLoonieState();
        outOfStockState = new OutOfStockState();
        if (count > 0) {
            currentState = startState;
            this.count = count;
        } else {
            currentState = outOfStockState;
            this.count = 0;
        }
    }
    public void insertLoonie() {
        currentState.insertLoonie( this );
    }
    public void ejectMoney() {
        currentState.ejectMoney( this );
    }
    public void dispense() {
        currentState.dispense( this );
    }
    public void setState( State state ) {
        currentState = state;
    }
    public int getCount() {
        return count;
    }
    // getters for state objects, machine actions, etc.
}
```

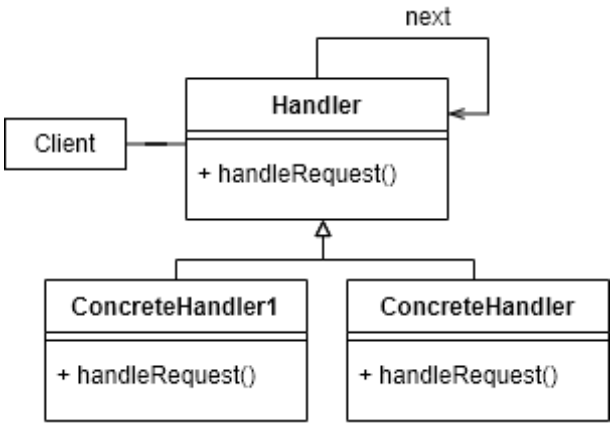
Decorator pattern

Intent: attach additional responsibilities to an object dynamically. It is usually used for making user interface embellishments such as adding decorations like a menu bar, vertical scrollbar, or horizontal scrollbar to a basic window. It's used when you don't want too many new subclasses. You use aggregation instead of inheritance

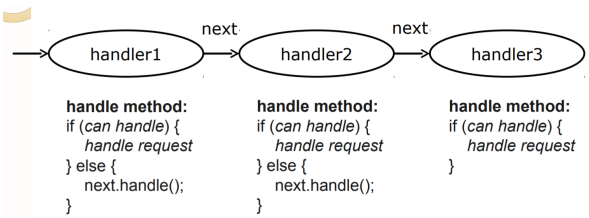


Chain of responsibility pattern

Intent: avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.



Example:



Design Principles

The goals is to enhance flexibility under changing needs and improve reusability in different contexts.

Open closed principle: Classes should be open for extension but closed for modification. Feel free to extend the classes and add new classes when needing changes. Existing classes are tested and work so do not tinker with them. "Encapsulate what varies"

Dependency inversion principle: Depend upon abstractions. Do not depend on concrete classes. Program to interfaces, not implementations. Favor composing objects over implementation inheritance.

Principle of least knowledge: "Only talk to your immediate friends." For an object, reduce the number of classes it knows about and interacts with. Reduces coupling and changes cascading throughout the system.

Law of demeter: Avoid calling methods of objects returned by other methods. "One dot only rule" For method M of object O, only call methods of the following objects:

1. Object O itself
2. parameters of method M
3. any objects instantiated within method M
4. direct component objects of object O