CS 130

Week 1

- **UML basics**
- Use-case diagram
  - Representation of system-actor interactions
  - E.g ATM machine
    - Customer: withdraw, deposit, balance
    - Bank: refill, anti-theft system, monitor transactions, view user's balance, manage machine's cash
  - Limitations
    - Use cases not directly linked to actors
  - Generalizations
    - Pay: with cash, card, EFT
  - Inclusion
    - Customer authentication: deposit+withdraw cash will not work without authentication
    - Place order -> payment, order can't be placed without payment
  - Extension
    - Login <- forget password
  - Uber eats
    - Customer
      - Browse restaurants, view menu, order food (included by restaurant activate), set up contact information, check out included by place order (payment by card
    - Restaurant personnel
      - Activate order, order notified (included by customer order), prepare food
    - Food-delivery personnel
      - Accept delivery (included by restaurant accept order), picked up, your food is on the way, delivered to the customer
    - Customer service
      - Process complaints, contact restaurants, refund

Week 2: UML

- Requirement engineering
  - Good foundation, risk reduction, identify problems early
- Modeling
  - State machines, entity-relationship diagrams, data flow
  - Describe a system at a high level of abstraction

- UML: unified modeling language

- graphical, common, simple representation of software design and implementation
- Nearly everything is optional and incomplete, open to interpretation and designed to be extended
- **Static modeling**
  - Code-level *without* running a program
  - Over-approximate + conservative
  - Describe a program behavior wrt all possible inputs
  - Style-checker, symbolic execution, verification pre-cond post-cond
  - *Class diagram*
- **Behavioral modeling (Dynamic analysis)**
  - *Running* a program with concrete input
  - *Use case diagram, sequence diagram, state diagram, activity diagrams*

- Use-case diagram (Dynamic)

- Stick-figure: actor
- Oval: use cases (v+n), association line between actor and use case
- Generalization: an arrow pointing towards the more general case
- Inclusion: login is a part of tracking a package (part(login) <— whole(tracking))
- Extension: not part of basic flow (whole(basic) <— optional)
- Story, main/alternative flows, nonfunctional requirements, entry/exit conditions

- State diagram (Dynamic)

- Show various stages of an entity during its lifetime, can be used to show the state *transitions* of methods, objects, components
- State: a condition of a modeled entity for some action is performed
  - Could include substates
- Action: an atomic execution, completes without interruption
- Activity: collection of behaviors
- Transition: an arc from one state to another (triggers, guard conditions, actions, trigger[guard] effect/
- Initial/final state: solid black circle

- Class diagram (Static)

- Box: class, inside list variables, attributes, methods
- Multiplicity: number of the components, =1 means singleton

- **Class relationship** (from weak to strong associations)
  - Dependency: A *uses* B (dotted line arrow)
  - Association: A *has* a class B long-term (solid line arrow)

```
public class DoS {
    private List<People> directees = new ArrayList<>();
    public void addDirectee(Student s) {
        directees.add(s)
    }
}
```

  - Aggregation: A *owns* a class B life-time (empty diamond arrow)

```
public class HonorCourses {
    private Module[] module;
```

```
        public HonorCourses(Module[] mod) {
            this.module = mod;  // pointer
        }
    }
    public void main(String[] args) {
        Module mod = new Module[6];
        HonorCourses hc = new HonorCourse(mod);
    }
```

- Composition: A is made up of B, B's purpose is to create A (solid diamond arrow)
  - C++: aggregation is pointers/references, composition is containing instances

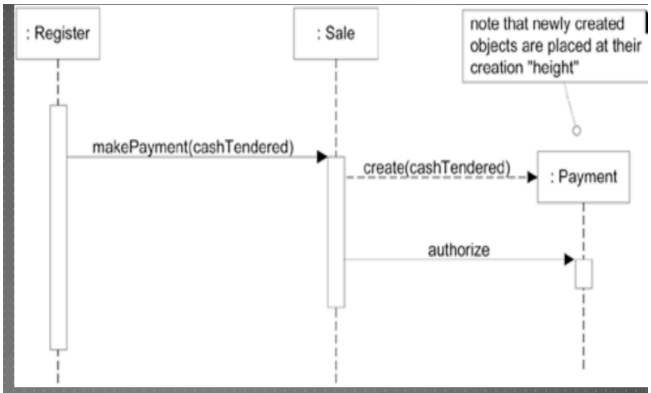
```
 public class Division {
   private List<Employee> division = new ArrayList<Employee>();
   private Employee[] employees = new Employee[10];
 }
public class Board {
    private Square[] squares;
    public Board() {
        squares = new Square[9];  // new instances
    }
}
```

- Generalization: inheritance, subclassing (solid arrow triangle)
  - B is a A, A is a generalization of B. Java extends.
- Realization: sub-typing (dotted line arrow triangle)
  - Java: implements

- Operations in class diagrams
  - Visibility
    - Public +
    - Private -
    - Protected #
    - Package ~
  - Parameter list
    - Direction (in/out)
    - Name
    - Type
    - Multiplicity
  - Polymorphism
  - Abstract operations (italics)
  - **+ func_name(var_name: type): return_type**

- Constraints on operations
  - Precondition
  - Postcondition
  - Body condition (invariants): must be overridden by subclasses
- Template class
  - Templates (generic types) allow a developer to design a class without specifiying the exact types on which the class operates
  - insert(element: ElementType): void

- Sequence diagram
  - Think of it as a table
    - Columns: classes + actors
    - Rows: time steps
    - Horizontal: show method invocation flow
    - Vertical: lifeline

- 

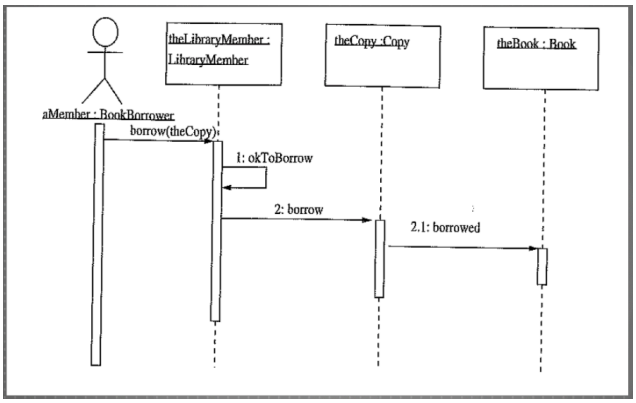

```
public class Register {
   public void method (Sale s) {

      s.makePayment(cashTendered);
   }
}


public class Sale {
   public void makePayment(int amount) {
      Payment p = new Payment(amount);
      p.authorize();
   }
}
```

- 


```
public class A {
   List items =null;
   ...
   public void noName (B b) {
      b.makeNewSale();
      for (Item item: getItems()) {
         b.enterItem(item.getID(), quantity);
         total = total +b.total...
         description = b.desc...;
      }
      b.endSale();
   }
}
```

- 


```
            ...
            LibraryMember theLibraryMember = null;
            theLibraryMember.borrow(theCopy);
        }
        public class LibraryMember {
            public boolean borrow (Copy theCopy) {
                okToBorrow();
                theCopy.borrow();
            }
```

```
    public boolean okToBorrow() { ... }
}
public class Copy {
    Book theBook;
    public void borrow() {
        theBook.borrowed();
    }
}
Public class Book {
     public boolean borrowed(){ ... }
}
```

- Sequence diagram frames
  - Alt: if else
  - Loop: for
  - Opt: if
  - Par: parallel
  - Region: one thread critical region

Week 2: Information hiding

- Software design principles: information hiding principle
  - Low coupling/decoupling: reduce dependencies
  - High cohesion: one class doesn't do multiple things
  - Separation of concerns
  - Misunderstanding:
  - Analyze how **changes** will affect existing code and assessment of changeability, list which interfaces must be modified to accommodate the anticipated changes
    - Not about declaring private attributes and getters -> *not* information hiding!
    - Not about using static call instead of constructor followed by a dynamic call.
    - Not about using primitive types vs. object types.
    - Not about call sequences.
    - *** Modularization

- ***Modularization***
  - Decide *what should be included in the interface vs. implementation? Or API name/inputing types/output?*
  - Module
    - A self contained piece of code that does one job
    - **Independent work assignment**, can make localized changes without communication to other modules
    - Purpose: parallelization of work, reduce shared resources
    - Class/directory/package/library all can be a module

- **Information hiding principles**
  - Hiding secrets: things that are likely to change **independently and isolatedly**
  - Interface: things that are not likely to change, show in API, do not reveal volatile information
  - Observer pattern: publisher doesn't know the implementation of the observer
  - Strategy pattern: hide algorithms from the clients that use them
- Functional decomposition
  - Each module corresponds to each step in a flow chart
  - Data representation is shared

- Case study: KWIC
  - Functional decomposition: flow chart
    - Data representation is shared
    - Expose internal content of array indices
    - Return type: how to decode the returned information?
      - Information of internal data layout was shared across classes -> exposed secrets
      - e.g. expose an entire hash table instead of inputting key outputting value
    - Change internal data structure type makes the program fail
  - Information hiding
    - Line storage
      - If getLine() returns char[][] = as worse as #1
    - Share information about data, not every secret

Week 3: Design Pattern

- Aggregation vs. composition
  - Aggregation: "has-a" relationship in terms of collection, B consists of A, A's purpose is to create B
    - Eg: A has an arrayList of B, a channel consists of videos (the case in which you can have a video without a channel)
  - Composition: in the case which you can't have a video without a channel
- Use-case diagram
  - Representation of interactions with more than one actor
    - To represent use case or functionality from each user's perspective
    - Sequence diagrams represent interaction between multiple participating objects
- Terminal state in a state diagram
  - Not a must
  - Waiting for the initial state
- Recap
  - Module: individual working assignment, for parallel development,
  - API (interface of a module): name, input argument type, return type, shown to other modules
    - Attributes that are not likely to change
  - Implementation of a module: secrets that are likely to change
  - Violation of information hiding
    - clients must know the content/order of the data structure
    - Not about not having getting method
    - Not about using dynamic call

Design Pattern

- A solution to a problem in a context
- A language for communication solutions with others
- Class names and directory structures are not equal to good design
- Depend on programming languages (related to object destruction, creation), tradeoffs
- Provide abstraction, a target for reorganization and refactorization of class hierarchies
- Pieces of a design pattern
  - Pattern name and classification
  - **Intent**: future/anticipated usage, correspond to the localized changes to the system. Easier to maintain if matched with your intent

- ***Requirements***
  - Add a new type for object where certain behaviors do not make sense
  - You would like not to create too much code duplication
  - You would like clients not to be confused by APIs
  - You would like an object to be associated to a different behavior implementation at runtime

- **Strategy pattern**
  - Motivation
    - **Multiple behavior implementations** of strategy
    - Assign different behavior to different subclasses at both static and runtime easily
  - Use subclassing and subtyping, field delegation, represent abstract class with abstract methods, runtime behavior association
  - Encapsulation, **makes algorithms interchangeable** at runtime. Strategy lets the algorithm vary independently from clients from that use it
  - Eg: delegate fly behaviors and quack behaviors to the class FlyBehavior and QuackBehavior
  - Example: inheritance does not make sense for all subclasses(unwanted behaviors), destroy code reuses, change behavior defined by interfaces

```
public interface FlyBehavior {
    public void fly();
```

```
}
public class FlyNo implements FlyBehavior {
    public void fly() { System.out.println("Does not fly"); }
}
/* ============================================================*/
public abstract class Duck {
    // shared by all types of ducks
    public void swim() {...}
    abstract public void display();

    // different ducks have different behaviors
    private FlyBehavior fb;
    public void setFly(Flybehavior f) { fb = f; }
    public void performFly() { fb.fly(); }
}

public class DuckA extends Duck {
    public DuckA() { FlyBehavior flyA = new FlyNo(); } // Set fly behavior
    public void display() { System.out.println("I'm DuckA"); }
}
/* ============================================================*/
public void main(String[] args) {
    DuckA a = new DuckA();
    a.performFly();
}
```

- Observer pattern

- **Intent**
  - Easily add/remove new type of observer
  - **Challenging to add new type of subject** with different subject states, or modify update interface
- 1 to n relationships between objects so that when 1 (weather data) changes state, n dependents (display types) are notified and updated automatically. —> public/subscribe model
- Information hiding (loose coupling)
  - The only thing that the subject knows about an observer is that it implements an observer interface
  - Do not need to modify subject to add/remove observers
  - Change to subject/object does not affect each other
- ConcreteSubject: WeatherData
- ConcreteObserver: StatisticsDisplay

```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();

}

public class WeatherData implements Subject { // concrete subject

    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() { observers = new ArrayList(); }
    public void registerObserver(Observer o) { observers.add(o); }
    public void removeObserver(Observer o) { observers.remove(observers.indexOf(o)); }
    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.update(temperature, humidity, pressure);  // update info for each obs
erver
        }
    }

    public void measurementsChanged() { notifyObservers(); }
    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }
}
/* ============================================================*/
public interface Observer {
    public void update(float temperature, float humidity, float pressure);
}
public interface DisplayElement {
    public void display();
}

public class CurrentConditionDisplay implements Observer, DisplayElement { // concrete obs
erver
    private float temperature;
    private float humidity;
    private Subject weatherData;

    public CurrentConditionDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }
    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
```

```
        this.humidity = humidity;
        display();
    }
    public void display() { /*print weather measurements*/ }
}

/* ============================================================*/
public static void main(String[] args) {
    WeatherData wd = new WeatherData(); // subject
    CurrentConditionDisplay ccd = new CurrentConditionDisplay(wd);  // observer
    wd.setMeasurements(1, 1, 1);  // will notify to get observer updated
}
```

- Mediator pattern

- An extended form of observer pattern: n to n relationship
- To centralize complex **communications** and control between related objects
- Inter-communication between subjects, spawn a series of actions and restraints/order on which these objects communicate
- If (someEvent) { //a series of actions }
- Mediator:
  - Different objects tell the mediator that their state changed
  - They respond the requests from the mediator
  - Different objects do not need to know about each other anymore
- With the mediator in place, all objects are decoupled from each other
- Subject needs new change -> all logic will be added to the mediator
- Benefits
  - Increase reusability of the objects, decoupling them from the system
  - Simplifies maintenance of the system by centralizing control logic
  - Simplifies and reduces the variety of message sent between objects in the system
- Drawbacks
  - Overly complex
  - Single point of vulnerability and complexity

- Conclusion

- **Strategy:**
  - Choice of algorithms can be easily changed at runtime
  - Independent from the clients that use it
- **Observer:**
  - add new observers and new types of observer
  - Hard to add new types of subjects, sensor data associated with the subject
  - Information hiding to hide subject data?
    - But change of subject data still need to be communicated to other observers
    - Harder to debug
- **Mediator:**
  - Encapsulates dependencies between objects

- Factory method pattern

- Encapsulate object creation. This decouples the client code in the super class from the object creation that happens in the subclass.
- Defines an interface for creating an object but let subclasses decide which class to instantiate —> defer instantiation to subclasses.
- Creator classes (Pizza stores) + Product classes (Pizzas)
- Pros: make it easy to add a new product type
  - Easy to add a new creator
  - Easy to change the conditions under which different objects are created
  - Client applications do not know individual concrete types of object being created.
- Cons:
  - Complicated to understand two different parallel class hierarchies
  - Complicated to know which specific type of an object is created
  - Adding new ingredients is hard

```
// Factory Method pattern

// Product
public abstract class Pizza {
    String name;
    void prepare() {...}
    void bake() {...}
    public String getName() { return name; }
}
public class CheesePizza extends Pizza { }
public class GreekPizza extends Pizza { }

/* ============================================================*/
// Factory
public abstract class SimplePizzaFactory {
    public abstract Pizza create(String type);
}
// Concrete factory
public class NYFactory extends SimplePizzaFactory {
    public Pizza create(String type) {
        System.out.println("New york style pizza: ");
        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("greek")) {
            pizza = new GreekPizza();
        }
    }
}

/* ============================================================*/
// Creator
public abstract class PizzaStore {

    SimplePizzaFactory factory; // has different implementation of create pizza
    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza createPizza(String type) { // factory method
        factory.create(type);
    }

    public Pizza orderPizza(String type) {
        Pizza pizza = createPizza(type);  // does not know which factory creates the pizza
        pizza.prepare(); // common operations
        pizza.bake();
        return pizza;
    }
}
```

```
// Concrete Creator
public class NYPizzaStore extends PizzaStore { }

/* ============================================================*/
public static void main(String[] args) {
    SimplePizzaFactory nyfactory = new NYFactory();
    PizzaStoreA store = new NYPizzaStore(nyfactory);
    Pizza pizza = store.orderPizza("cheese");
}
```

- Strategy vs. Factory Method?
  - Similarities
    - Subclassing, delegation
    - Interface mechanism: interface, abstract class
  - Differences

Strategy	***Factory Method***
Interface of FlyBehavior + its concrete implementations	PizzaFactory abstract class
In superclass: - declare FlyBehavior instance - setFlyBehav(FlyBehavior fb); - performFly() { fb.fly() };	Concrete implementations of PizzaFactory: - Pizza createPizza(String type) { … }
In child classes: - setFlyBehav(new FlybehaviorA()); - …	Client code - PizzaFactory pf = new NyFactory(); - Pizza p = pf.createPizza(“cheese”)
Behavioral pattern	Creational pattern
Instantiate behavior at runtime	Create objects of a specific type

- Abstract factory pattern

- Provide an interface (ingredient collection) for creating families of related or dependent objects without specifying their concrete class.
- Create ingredient collection interface for products
- Implement ingredient collection with concrete ingredients createIngredient()
- Integrate these new ingredient factories into the PizzaStore code

```
// Abstract Factory Pattern
/* ============================================================*/
// Product
public abstract class Pizza {
    CheeseIngredientCollection cheese;
    GreekIngredientCollection greek;
    void prepare() {...} // common operations
    void bake() {...}
}
public class CheesePizza extends Pizza {
    IngredientCollection collection;
    public CheesePizza(IngredientCollection collection) { // Which cheese is created is de
termined at run time by the factory passed at object creation time
        this.collection = collection;
        this.cheese = collection.createCheese();
        this.greek = collection.createGreek();
    }
}
public class GreekPizza extends Pizza {...}

/* ============================================================*/
// Ingredient collection
public interface IngredientCollection {
    public Dough creatDough();
    public Sauce createSauce();
}
public class CheeseIngredientCollection implements IngredientCollection {
    public Dough createDough() {
        return new ThinCrustDough();
    }
    public Sauce createSauce() {
        return new MarinaraSauce();
    }
}
public class GreekIngredientCollection implements IngredientCollection { }


/* ============================================================*/
// Creator
public abstract class PizzaStore {
    IngredientCollection collection = null; // Abstraction

    public abstract Pizza createPizza(String type); // factory method

    public Pizza orderPizza(String type) {
        Pizza pizza = createPizza(type);  // does not know which instance creates the pizz
a
        pizza.prepare(); // common operations
        pizza.bake();
        return pizza;
    }
}

// Concrete Creator
public class PizzaStoreA extends PizzaStore {
    public Pizza createPizza(String type) { // must implement abstract method
        Pizza pizza = null;
        if (type.equals("cheese")) {
            IngredientCollection collection = new CheeseIngredientCollection();
```

```
            pizza = new CheesePizza(collection); // object creation time --
> determine the type of ingredients
        } else if (type.equals("greek")) {
            ...
        }
        return pizza;

    }
}

/* ============================================================*/
public static void main(String[] args) {
    PizzaStoreA store = new PizzaFactoryA();
    Pizza pizza = store.orderPizza("cheese");
}
```

- Abstract factory pattern vs. factory method pattern

Factory method	***Abstract Factory***
	Concrete implementations of PizzaFactory: - Pizza createPizza(IngredientCollection ic) { … }
	Client code - PizzaFactory pf = new NyFactory(); - Pizza p = pf.createPizza(new CheeseCollection())
	A IngredientCollection object in PizzaFactory
	A concrete IngredientCollection object in NyFactory class

Week 4: Design Pattern

- Singleton

- Motivation
  -
- Definition
  - Ensures a class has only one instance and provides a global point of access to that instance
- Global variable?
  - Anyone can mess around with it.
  - Could be modified with another reference
  - Eager instantiation: beginning of program
- When constructor is private:
  - public static MyClass getInstance() {… new MyClass() ..}
  - Private unique instance + getInstance()
  - Eg. logger, game board
  - Synchronized block: avoid two threads creating the new instance
- Late instantiation
  - ***Why not eager instantiation?***
    - Suppose that singleton object is super resource-intensive to create, eg. A threadpool
    - End up creating an expensive resource
- **Thread-safe** instantiation
  - volatile: have to get it main memory, not able to be cached locally by a thread
  - Check twice in getInstance() so that instance retrievals don't apply a lock —> performance

```
//Singleton
public class Singleton {
    private static volatile Singleton uniqueInstance;
    private Singleton() { }
    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            synchronized (Singleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueSingleton;
    }
}
```

- Command pattern

- Motivation
  - Multiple objects to be controlled
  - Different actions on different objects
  - Decouple a receiver object's actions from invokers.
  - **Common interface** to request commands to be invoked on varying objects
- Command objects
  - Encapsulates a request to do something on a specific object, e.g. turn on alight on a specific lamp
    - Thereby letting you parametrize other objects with different request, queue or log requests, and support undoable operations.
  - Store a command object for each button, invoke command when pressing button
  - Promotes decoupling: the button doesn't have to know anything about the command
    - Easy to change types of object (light, ledlight, discolight)…
    - Easy to change types of command on this object: turnon, turnoff…
    - RemoteControl does not change.
- Roles
  - client: create invoker, concrete command, receiver
  - **command**: execute(), undo() defines a binding between action and command receiver
  - **Invoker**: setCommand() and buttonWasPressed() to execute the command
  - **receiver** action(): simple

- **Create Command Object**
  - Interface command // same interface, operate on different types of objects
  - ConcreteCommand implements Command

```
// Command Object Interface
public interface Command {
    public void execute();
}

// Concrete Command Object
public class LightOnCommand implements Command {
    Light light;
    public LightOnCommand(Light light) { // pass specific light that this command controls
        this.light = light;
    }
    public void execute() {
        light.on();  // Light has an action on()
    }
}
```

- **Using the command object**

```
// Invoker
public class RemoteControl {
    Command slot;  // one slot to hold one command, control one device
    public RemoteControl() {}
    public void setCommand(Command command) { // modify the behavior of the button
        slot = command;  // pass in command
    }
    public void buttonWasPressed() { // called when button is pressed
        slot.execute();
    }
}
```

- **Simple test of the remote**

```
// Client
public class RemoteControlTest {
    public static void main(String[] args) {
        RemoteControl remote = new RemoteControl(); // invoker
        Light light = new Light(); // receiver
        LightOnCommand lightOn = new LightOnCommand(light); // create a command and pass t
o receiver
```

```
            remote.setCommand(lightOn); // pass the command to the invoker
            remote.buttonWasPressed();
        }
    }
```

- **Command vs. Mediator**
  - Mediator in an ***intermediate*** that reduces n-n to n-1 relationship
  - Command is an ***interface*** for an action that encapsulates an underlying device
  - Command interface is ***not device-dependent.***
  - Devices have different sets of names for their actions.

- How does the remote know the difference between the kitchen light and the living room light?
  - It doesn't. The remote only sets the command and activates the command when needed. It does not know about the object that the command is being executed on.
- How would you implement a history of undo operations?
  - Keep an array of executed commands in the object. In concreteCommand, append to the array.

- Adaptor

- Motivation
  - Legacy code that does not fit the target interface you desire
  - Client —> *Target*:request() <— **Adapter:request()** —> Adaptee: specificrequest()
  - Client can only see the Target Interface. Target interface has a legacy implementation called Adapter which is composed with the Adaptee.
- Adaptor calls the *nonideal* interface of adaptee.
- Definition
  - Converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise due to incompatible interfaces.

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

// Adaptee
public interface Turkey {
    public void gobble();
    public void fly();
}

// Adapter
public class TurkeyAdapter implements Duck {
    Turkey turkey = null;
    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }
    // ideal interfaces
    public void quack() { turkey.gobble(); }
    public void fly() {
        for (int i = 0; i < 5; i++)
            turkey.fly();
    }
    // other nonsupported ideal interfaces
    public void bjky() {
        throw new UnsupportedOperationException();
    }
}
```

- Facade Pattern

- Motivation
  - A simple interface for a family of complex subsystems
- Facade vs. Adapter
  - Facade simplifies an interface, decouples a client from a subsystem of components
  - Intent is to simplify, not to convert
- Defintion
  - Provides a unified interface to as et of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

- Template method pattern

- Motivation
  - Varying objects want to do similar things but slightly different ways
  - The overall **work flow** is similar
  - Certain steps are identical but certain steps differ
  - Abstraction of a common procedure/skeleton/**workflow** for varying object
- Definition
  - Defines the skeleton of an algorithm in a method, **deferring some steps to subclasses**. Let subclasses refine certain steps without changing the algorithm's structure
- Template vs. Strategy

Template	Strategy
Defines outline of algorithm	Defines a family of algorithms and makes them interchangeable
Keeps control over the algorithm's structure	No
Puts duplicate code in super class	Uses composition so its more flexible to switch with a new strategy

- Subclassing vs. subtyping
  - Subclassing: Inheritance, reusing inheriting an extension from a superclass
  - Subtyping: Polymorphism declaring an interface , implements

```
public abstract class CaffeineBeverage {
    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    // varying steps
    abstract void brew();
    abstract void addCondiments();

    // common steps
    void boilWater() { }
    void pourInCup() { }
}
```

- State Pattern

- Motivation
  - Complex state transition
  - Not forget handling all transitions for each state at compile time
  - State + transition
- Defintion
  - Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **State pattern mechanics**
  - Encapsulate the varying behavior
  - Create a new class for every State
  - Localize the behavior for each State
  - Adding new State —> add new class
  - Handing new requirements is easy
- What if there is no transition between most states?
  - Drawback of state pattern
  - You have to implement(override) transition between a new state to every existing state
- Change State from an interface to a class, concrete states extends State?
  - Can't check if you have implemented each state
- **Implementation**
  - Define State interface: all transitions
  - Implement State classes
  - Elimination the conditional code by delegating the work to the state class
  - Context: set state, get current state
- Pros
  - Encapsulate state behaviors
  - Avoid inconsistent state changes
  - Remove conditional statements
  - Ensuring each state takes care of all transitions are **done by compile time check**
- Cons
  - Brittle interface
  - Bulky code more objects
  - Class explosion problem
- Strategy vs. State
  - Similar: declaring a field state and delegating work to the field. State is a specific case of using strategy, where actions have consequences on the machine's state.
  - Strategy pattern is a flexible alternative to subclassing, with strategy you can change the behavior by composing with a different object.
  - State pattern is an alternative to putting lots of conditions, by encapsulating the behavior within state objects, you can simply change the state object in context to change its behavior.
- Information hiding principle
  - Changes that are likely to happen in the future —> add new state
  - Less likely —> add new transition

```
// Context
public class GumballMachine {
    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;

    State state = soldOutState;
    int count = 0;

    public GumballMachine(int nGumballs) {
        soldOutState = new SoldOutState(this);
        noQuarterState = new NoQuarterState(this);
```

```
        hasQuarterState = new HasQuarterState(this);
        soldState = new SoldState(this);
        this.count = nGumballs;
        if (nGumballs > 0) { state = noQuarterState; }
    }

    // transitions
    public void insert() { state.insert(); }
    public void eject() { state.eject(); }
    public void turnCrank() { state.turnCrank(); }

    // setter/getter methods
    public State getNoQuarterState() { return noQuarterState; }
    ...
    public void setState(State state) { this.state = state; }
}

/* ============================================================*/
// State
public interface State {
    // transitions
    public void insert();
    public void eject();
    public void turnCrank();
    public void dispense();
}
public class SoldState implements State {
    GumballMachine machine;
    public SoldState(GumballMachine machine) {
        this.machine = machine;
    }

    // transitions
    public void insert() { }
    public void eject() { }
    public void turnCrank() { }
    public void dispense() {
        machine.releaseBall();
        if (machine.getCount()>0) { // set conditions
            machine.setState(machine.getNoQuarterState()); // set state transition
        } else {
            machine.setState(machine.getSoldOutState());
        }
    }
}
```

Week 5: Build Management

- Introduction
  - A program cannot be immediately executed after a change, intermediate steps:
    - Source to object
    - Link objects together to create program
    - Documentation generated from program text could also change
- Definitions
  - Source components: **manually** created
  - Derived components: created by the computer/compiler. If a component has changed, all its derived components must be rebuilt.
- Make
  - Tool for building programs, during the course of which existing derived files are reused
- ANT
  - Allows specific extensions by the user as well as third parties.
  - Task-oriented
- Commenting
  - Bad < no
  - Brief and concise
  - Documentation synced with source code

Week 6: Testing

- Testing
  - Most common quality assurance methods
  - Testing is Means of detecting/revealing errors
  - Debugging is a means of *diagnosing and correcting* the root causes of errors that have already been detected
- Types of testing
  - Unit testing
  - Component testing
  - Integration testing
  - System testing
  - Regression testing
- Another classification of testing
  - Black-box testing
  - White-box testing
    - Better testing method
- Test coverage
  - Statement coverage (go through all lines of code)
  - Branch coverage (go through all branches)
  - Path coverage
- Infeasible path
  - Dead code: no input to exercise a particular path of program
- Loop unrolling
- Symbolic execution
  - Static behavior in terms of logical constraints
- As few tests as possible, but exercises more coding paths
- Recognize redundant test cases

- Branch executions
  - # of branch executions that have the freedom to be evaluated true or false.
  - If 2 times: 2^2 is paths. Branch is 2*2 = 4.
  - If 3 times: 2^3 is the total number of paths. Branch execution is 2*3 = 6.
  - If a branch always evaluates to one result, it does not contribute to the total number of paths
  - If the same branch gets executed over and over again, it is still one branch
  - # branch executions $\neq 2^{loopiteration}$
  - # path = 2^# branch
- Test Coverage (to assess how good our test suite is, test adequacy)
  - Statement
  - Branch
  - Path
    - For a loop=free program with k brances, the number of paths 2^k if no infeasible oath exists.
  - Program w/o a loop
- CFG in the presence
  - Loop unrolling
  - To count # of paths
  - Program with a loop
  - Short circuit
    - Consider source code as is in this class for the purpose of answering questions
- Infeasible paths
  - Dead code

  - No input that satisfies to give a path condition
- Symbolic execution
  - ***Static*** analysis that reasons about program behavior in terms of logical constraints
  - Path condition (logical constraint) + effect
  - Why Symbolic execution tree?
    - Test input generation for a given program
    - Find a concrete input assignment for each path condition
    - Fewer tests with high coverage
  - Symbolic execution (static) tree
    - Node: branch -> T/F
    - Get path conditions + effects

- Branch coverage
  - When you have redundant branches (multiple branches go to same result), 100% statement coverage but not 100% branch coverage
  - Evaluate each one of the conditions at least once
- Regression testing
  - Definition
    - When changes have been made, generate test designed to make sure that the software hasn't taken a step backward, or "regressed" is called "regression testing".
  - RTS: regression test selection
    - Key idea is to select tests that will exercise ***dangerous edges*** in the new program version
    - Dangerous edges: an edge in old CFG who target change after making a change —> about ***old CFG***
    - Safe RTS: guarantee to catch the same bugs every time
  - Step 1: Build CFG
  - Step 2: Run T = {T1, T2, …} on P
  - Step 3: Build edge coverage matrix
  - Step 4: Traverse two CFGs in Parallel
  - Step 5: Select tests relevant to dangerous edges
- Grammar based testing
  - Context free grammar
- Model-driven testing/Modern Testing
- Randoop: random/systematic testing
- Evosuite: search-based testing
- Major: test evaluation, mutation testing
- Decision mutation
  - Generate input
- Statement mutation
- Mutant: humans are similar to mutants
- Recap
  - Systematic in search space but randomize to explore variations
  - General
- Generate tests from UML activity diagram
  - Use node and branch coverage of the diagram
  - Node/edge/path coverage

Week 9: Hoare Logic

- Software inspection methods

- Collaborative construction
  - Pair programming
    - Achieve code quality similar to formal inspections
    - Cost is 10-25% higher than the cost of solo development, but reduction in development time is 45%.
    - Reduced the cost of defect correction from 40% to 20% via emphasis on software inspections.
  - Formal inspection
    - Specific kind of view show to be effective in detecting defects
    - Checklist focus the reviewer's attention on areas that have been problems in the past. The inspection focuses on defect **detection NOT correction**.
    - Reviewers arrive with a list of problems they've discovered.
    - Distinct roles are assigned to all participants.
    - The moderator of the inspection is not the author and has received training. — keep inspection moving
    - Author wrote the design or code.
    - Reviewer has a direct interest in the code but not the author — role is to find defects during preparation.
    - Scribe records errors that are detected and the assignment of action items.
    - Management — not included
  - Informal technical reviews
- Code reviews and Pair programming

- Hoare logic

- Motivation
  - Two quality assurance: code review and testing
  - Compute weakest preconditions can find subtle bugs and corner cases during peer code review.
  - Code comprehension, collaboration, code review, bug finding.
  - In order to post conditions to be true (must reach this point and produce no errors), make sure the weakest pre conditions are met.
  - Make writing pre/post conditions effective
- Modern code review process
  - CodeFlow
  - Primary goals: find defects, improve maintainability, share knowledge, broadcast progress
  - Why code review do not find bugs? 15% of comments provided indicate a possible defect.
  - During peer code reviews, under what basis should programmers approve code changes?
    - Reproducible failure. Presence of a test case.
    - Good test coverage, good code quality.
    - Does not break existing tests —> show that you run the regression test suites. No additional test failure.
    - Write a weakest precondition + postconditions
    - Style check
    - Variable name reasonable
- State predicate
  - A predicate is a boolean function on the program state
  - Examples: x == 8, x < y, m <= n =>(…)
    - ***true***: input reaches this point and produce no errors
    - ***false***: no input reaches this code and produces no error

- **Hoare triples**: A formal inspection technique
  - **{P} S {Q}** for any predicates P and Q and any program S
    - P: pre condition
    - S: program
    - Q: post condition
    - If S is started in P, it terminates in Q

- Examples
  - {true} x:=12 {x=12}
  - {x < 40} x:=12 {10 <= x}
  - {x < 40} x:x+1 {x <= 40}
  - {m <= n} j:=(m+n)/2 {m<=j<=n}

- Precise triples
  - **If {P} S {Q} and {P} S {R}, then P S {Q ∧ R}**
    - *Strongest postcondition* of S with respect to P: the *most precise* Q such that {P} S {Q}.
  - **If {P} S {R} and {Q} S {R}, then {P ∨ Q} S {R}**
    - ***Weakest precondition*** of S with respect to R: the *most general* P such that {P} S {R}.
    - Stronger preconditions are narrower
    - Write the weakest precondition as assertion before a program and document in Javadoc
    - Written *wp(S,R)*
  - Triples and WP: **{P} S {Q} if and only if P => wp(S,Q)**

  - **skip**
    - No-op
    - wp(skip, R) ≡ R
  - **assert**
    - If P holds, do nothing, or else terminate
    - **wp(assert P, R) ≡ P ∧ R**
    - wp(assert x<10, 0<=x)
      ==x < 10 AND x>=0
      ==0<=x<10
    - wp(assert x=y*y, 0<=x)
      ==x=y*y AND x>=0
      ==x=y^2
    - wp(assert false, x<=10)
      ==false AND x<=10
      ==false
      (*no input* that will execute the program statement and ends up with a post condition x <= 10)
  - **assignment**
    - Evaluate E and change value of w to E. Plug in from front to back
    - **wp(w := E, R) ≡ R[w:=E]**
    - wp(x:x+1, x<=10)=={x+1<=10}=={x<=9}
    - wp(x:=15, x<=10)=={15 <= 10}== false
    - wp(y:=x+3*y, x<=10)=={x<=10}
    - wp(x,y:=y,x, x < y) — swapping
      =={y<x}

- Program compositions
  - **If {P} S {Q} and {Q} T {R}, then {P} S;T {R}**
    - Sequential composition
      - **wp(S;T) ≡ wp(S, wp(T, R))**
      - wp(x := x+1 ; assert x ≦ y, 0 < x)
        ≡ wp(x := x+1, wp(assert x ≦ y, 0 < x))
        ≡ wp(x := x+1,0 < x≦y)
        ≡ 0 < x+1≦y
        ≡ 0≦x<y
      - wp(y := y+1 ;x := x + 3*y,y≦10∧3≦x)
        ≡ wp(y := y+1,wp(x := x+3*y,y≦10∧3≦x))
        ≡ wp(y := y+1,y≦10∧3≦x+3*y)
        ≡ y+1≦10∧3≦x+3*(y+1)

≡ y<10∧3≦x+3*y+3
≡ y<10∧0≦x+3*y

- **If {P ∧ B} S {R} and {P ∧ ¬ B} T {R}, then {P} if B then S else T end {R}**
  - Conditional composition
    - **wp(if B then S else T end, R) ≡ (B ∧ wp(S,R)) ∨ (¬B ∧ wp(T,R))**
    - wp(if x<y then z:=y else z:=x end, 0<=z)
      ==={x<y AND wp(z:=y, 0<=z) OR (x>=y AND wp(z:=x, 0<=z)}
      ==={x<y AND 0<=y} OR {x>=y AND wp(z:=x, 0<=z)}
      ==={x<y AND 0<=y} OR {x>=y AND 0<=x}

- Application: Loop Invariance
  - To check correctness about a loop
    - check if an invariant holds in the beginning and after the initialization
    - check if its maintained in the iteration
    - check if the invariant and the exit condition implies a post condition
  - Loop Invariant reasoning
    - {P} while (B) do {S} {Q}
    - **J is a loop invariant**, vf is a rank/variant function
    - P => J // in the beginning J holds
    - {J AND B} S {J} // during iteration, J holds
    - {J AND NOT B} => Q // after loop, J holds
    - vf >= 0
    - {J AND B AND vf=VF} S {vf<VF}
  - How to guess a loop invariant? Modify a post condition and a guard
  - How to guess a variant function? A function N-k if increases by 1
  - Automatically find loop invariants? Generate candidates and verify
  - Infer pre/post conditions? Yes from tests.
  - Caveats:
    - more than one loop invariants.
    - In the absence of meaningful post conditions, the loop invariant is often not meaningful as well.

Final topics
- Hoare logic
- Weakest precondition
- Regression testing
- Mutation testing
- Design problems and refactoring
- UML based testing
- Design pattern
- Symbolic execution and path condition
- Test coverage