Software Design Patterns: Session Two

Sahar Mostafa
UC Berkeley Extension
Summer 2020

Ice Breaker - Weekend update

Tell us about your weekend!

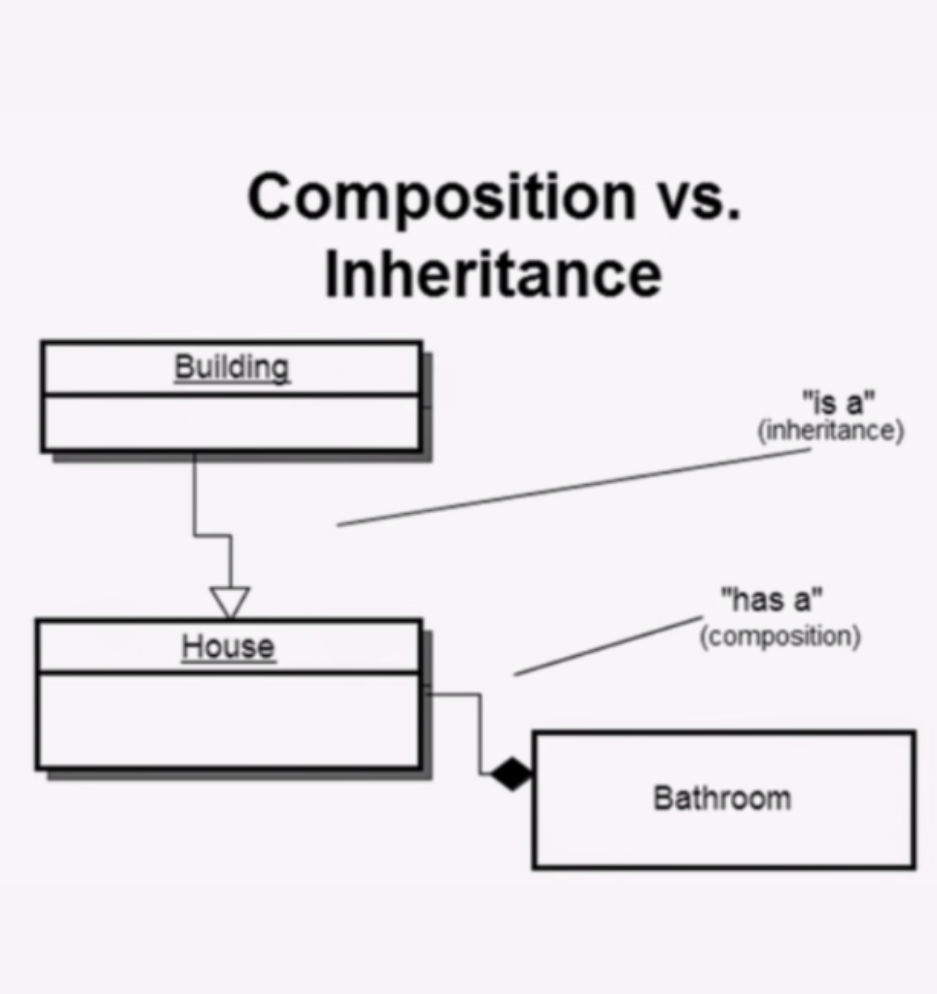
Week In Review

1. What is a software pattern?
2. Why do we need a pattern?
3. How can we apply a pattern?
4. OOP Principles and Class Diagrams

Today's Goals

1. Reusability
2. Refactoring
3. Design Pattern Types

Reusability



- **Inheritance - is a**

  Declare behavior at compile time, no change to implementation inherited from parent class

  inheritance breaks encapsulation, changing parents behavior force subclass to change

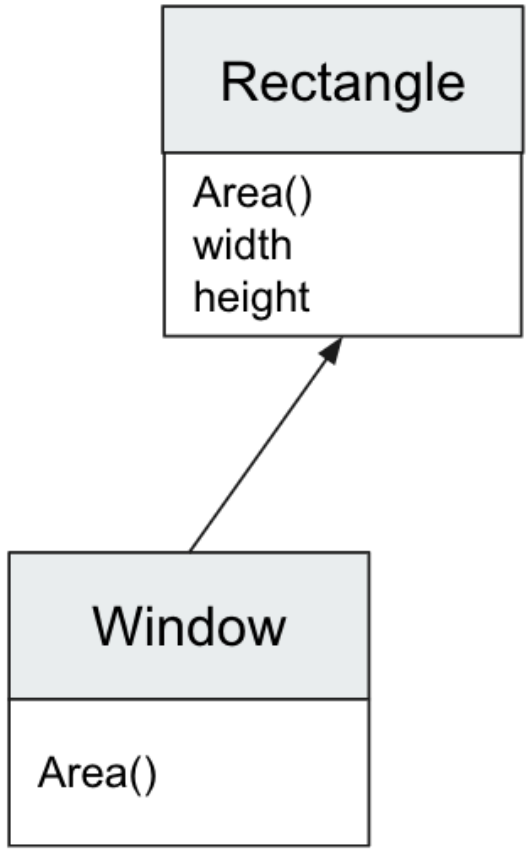
- **Composition - has a**

  dynamically at run time with objects acquiring references to other objects

  interrelationship focus via delegation

  indirection can be less efficient

Reusability



- **White-box reuse: internals of the framework are known - Inheritance**

  Reuse by subclassing base classes and override method

  Extend functionality by inheritance and dynamic binding

- **Black-box reuse: hidden internals are hidden - Composition**

  Assemble objects for a more complex functionality

  Define interfaces can interact and integrate with the framework
  Integrate interfaces via delegation

Story Time

A story about weeds, moss, grass and eventually coding ...

There is a yard, healthy looking with green grass with a side that sees no sun, moss is growing on that side

Moss requires zero effort to grow With a side effect: moss offers growth channel to weed, weed chokes out grass

Some prefer moss to bare dirt

Refactoring

“If it stinks, change it.” — Grandma Beck, discussing child-rearing philosophy

- Make the code more readable and more structured
- Repeated code can cause harm
- Extend the current behavior of a system
- Break down huge chunks of code
- Irrational usage of resources

Refactoring

“If it stinks, change it.” — Grandma Beck, discussing child-rearing philosophy

- **Long Methods**

  Break down to smaller methods

- **Bulky Class**

  A sign the design is wrong, break down into smaller classes

- **Extract method**

  Method with divergent logic

- **Abstract Logic**

  Move common logic to super class

Refactoring Dos and Do nots

Do

Break the code down into sections

Pull common logic to superclass

Push special logic to subclasses

Encapsulate logic Extract classes add inner classes

Build interfaces

Do not

Disrupt the whole code

Upcoming deadline

Refactoring effort outweighs the system's full redevelopment


15 mins Break

Exercise

Refactoring Exercise

Step 1: git repo clone session two

https://github.com/SMostaf/COMPSCIX418.2

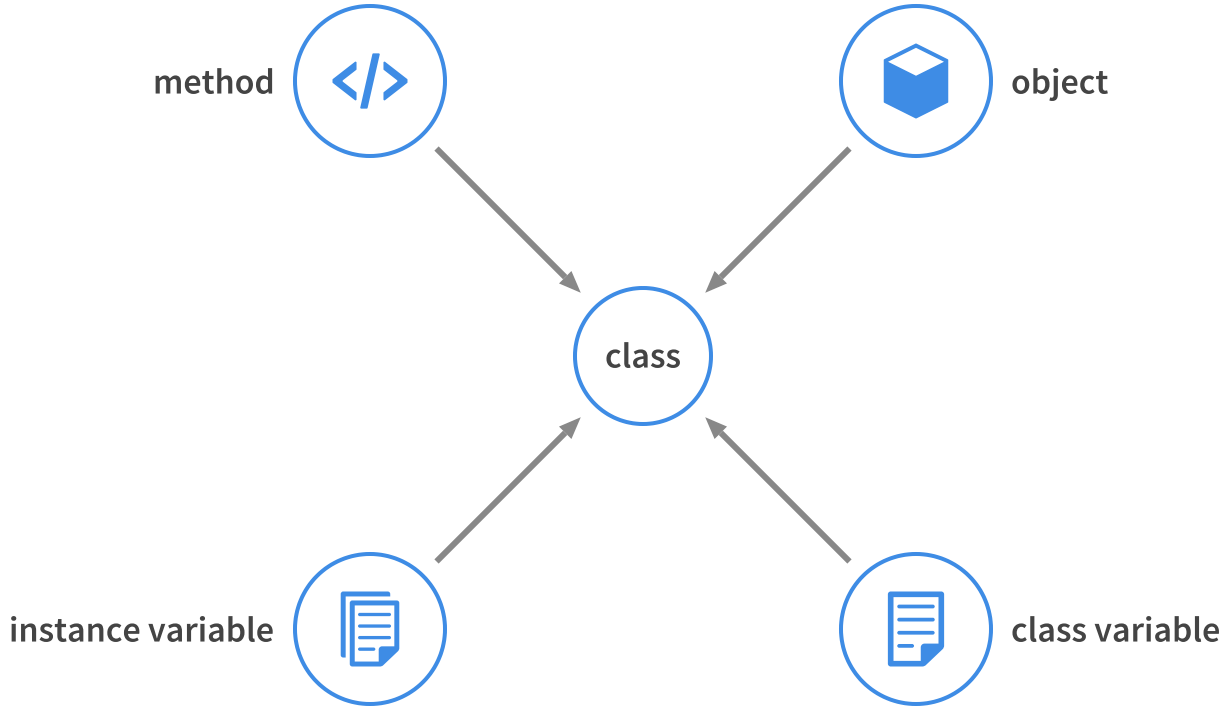
Step 2: README

Point out why we need to refactor the code, we will discuss the reasoning and direction in class

Improve the code in steps, which class would you start with and what improvement would you add

Impact on the client side, if any

Terminology



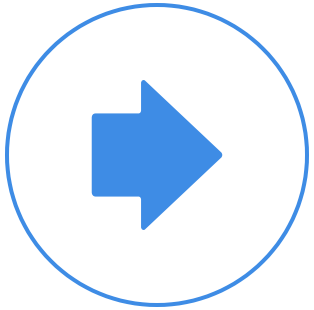
Terminology

Conglomeration

different method same object

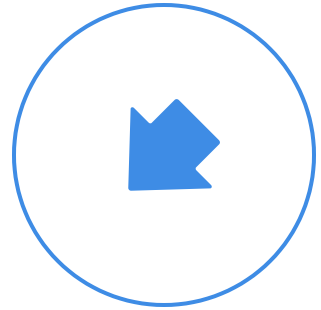
Recursion

same method same object



Redirection

same method different object



Delegation

different method different object

Software Design Patterns

Creational

Singleton

Only one instance is allowed in the system. For example: DB connection or a logger utility

Factory

Create objects based on a type. Actual instatiation is delegated to subclasses

Builder

The builder hides the process of building objects with complex structure, separates the representation of the object and its construction

Prototype

Useful when dealing with cloning objects and dynamic loading of classes

Structural

Adapter

Allow two incompatible interfactes to communicate by converting interfaces

Decorater

Used when there is a need to enhance or extend the behavior of an object dynamically

Facade

A single uber interface to one or more subsystems or interfaces

Proxy

Represents another object and can act on its behalf shielding an object from direct interaction

Behavioral

State

Mutating data allowing an object to alter behavior when its internal state changes

Observer

USed when one object is interested in the state of other dependent objects. When one object changes state all the dependents are notified

Strategy

A common interface with differnet internal algorithms allowing switching out one algorithm for another seamlessly.

Design Pattern Types

1

Creational

Focus on ways to instantiate an object

Singleton, Factory, Builder, Prototype

2

Structural

Rules governing building an object

Adaptor, Facade, Decorator, Proxy

3

Behavioral

Changing state of an object and ways to communicate that change

State, Observer, Strategy

How to Select a Design Pattern?

Review the intent, purpose and consequences of the design pattern

Reasons for redesign

How patterns interrelate

How the design can evolve in the future

Patterns Constituents

Problem: intent, motivation, describe problem space

Solution: structure, implementation and sample code

Context: environment, applicability, consequences and related patterns

Participants and Collaborators


15 mins Break

Exercises

Parking Lot

Step 1: git repo clone session two

https://github.com/SMostaf/COMPSCIX418.2

Step 2: README

Deliver classes

Deliver UML Diagram


15 mins Break

Chess Game

Step 1: git repo clone session two

https://github.com/SMostaf/COMPSCIX418.2

Step 2: README

Deliver classes

Deliver UML Diagram


15 mins Break

Author Book

Step 1: git repo clone session two

https://github.com/SMostaf/COMPSCIX418.2

Step 2: README

Deliver classes

Deliver UML Diagram

Code Review

```
func(A a) {
        if (a instanceof B) {
          B b = (B) a;
          b.f1();
          b.f2();
        } else {
          C c = (C) a;
          c.f1();
          c.f2();
        }
 }
```

```
func(Type type) {
        switch(type) {
          case a:
            break;
          case b:
            break;
        }
  }
```

```
if (POST) {
        if (valid(request)) {
          saveModel(request);
           redirect('index.html');
        }
}
if (valid(request)) {
 readModel();
 redirect('show.html');
}
```

```
boolean isValidRequest = valid(request);
if (POST) {
        if (isValidRequest) {
           saveModel(request);
           redirect('index.html');
        }
}
if (isValidRequest) {
 readModel();
 redirect('show.html');
}
```

Example

```
//save by ClassA.java
package packageA;

public class A {

protected void printHello() {

  System.out.println("Hello");

 }
}
```

```
//save by B.java
package packageB;
import packageA.*;

class B extends A {

public static void main(String args[]) {
 B obj = new B();
 obj.printHello();
}
}
```

Refactoring

```
public int getSum() {

 List<Order> listOrder = readOrders();
 int sum = 0;
 for (int i = 0; i < listOrder.size(); i++)  {
     sum += listOrder[i].getItemsCount();
  }
 return sum;
}
```

Refactoring - Extract Method

```
function printLogic(Order order) {
      double price  = calculatePrice(order);

      console.log("name:" + order.getCustomer());
      console.log("number:" + order.getId());
      console.log("amount:" + price);
}
```

Refactoring - Pull Method

```
class Employee {   }

class Salesman extends Employee {
 getName(){}
}

class Engineer extends Employee {
  getName(){}
}
```

Session Three

Next Week

Next Week

1. **Review Session Two**
2. **Creational Design Pattern**

   Singleton- Factory- Builder- Prototype
3. **In class exercises**