Object oriented principles

Principles supported by design patterns

Goals of object oriented software design

Object oriented programming languages exist to allow us to write software which is

- Robust (won't easily break)
- Flexible (ready for changes)
- Modular (reusable modules make up a whole)
- Expressed through the use of abstractions (makes it easy to map the actual problem components to software components, lets us focus on the essential and not the details)

Some principles exist to help achieve the goals

- Abstraction - Do not focus on the details (because details change and may vary)
- Encapsulation - Use objects which have both state and behavior (they provide good abstractions btw)
- Composition is often preferable to inheritance
- Program to an interface (a supertype)
- Avoid strong coupling (dependence) between classes (isolate them from the impacts of change)
- Don’t repeat yourself

Some more principles

- Single responsibility (don’t make classes with more than one reason to change)
  - For instance, don’t make a class which both calculate a result and prints it
  - Make one class for one job, e.g. calculator vs printer
- Avoid situations where a class or interface has to change (but allow for extension - backwards compatibility)
- High-level modules should be insulated from low-level modules, by the use of abstractions (it should be possible to change low-level modules without having to change the high-level module code)

Some examples - Abstraction

Abstraction - A module for persistent storage of a system should focus on the job (storage), not on the way things actually are being stored.

Think about what high-level operations a storage layer should have (and not how these operations will or even could be implemented).

Consider writing an interface or abstract class for your abstraction. Without actual implementation, there is no need for details!

Some examples - Encapsulation

Avoid putting functionality in one class, and data in another.

Think about what your objects represent in terms of what they know and what they can do.

Put the behavior (methods) and data (variables) in one single class, so that an object of the class knows its state and offers behavior which could vary depending on that state.

A game character: Knows of its name and health, knows how to fight with and talk to other characters.

Favor composition to inheritance

In Java you are only permitted to extend one class (at the time). Going for inheritance locks your design into what the new class primarily ***is*** but perhaps it is more important what it has (in terms of state and capabilities).

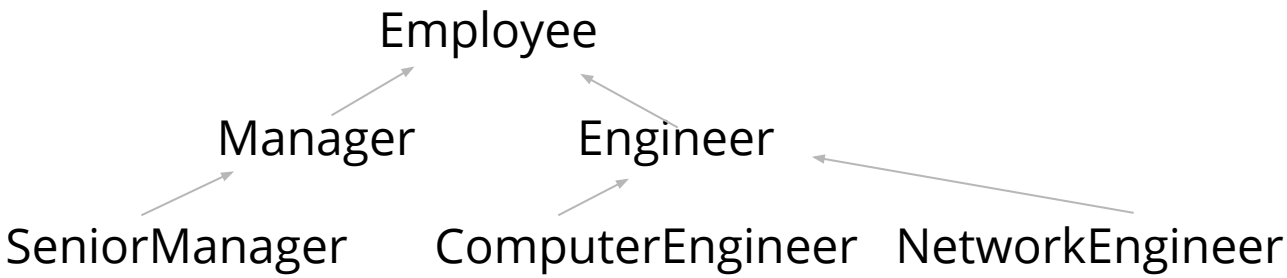
Making a weapon in a game a class to be inherited, leads to weird abstractions such as a class `Brick extends Weapon`. No all bricks are primarily weapons but they can be used for building houses too. Build house with weapons?

A character could have a method for fighting inherited from a superclass. But this method can’t be changed during runtime.

Make Weapon an interface, Brick implement Weapon, and Character have a weapon!

Another example on composition vs inheritance

Using inheritance we could have:



What happens when a ComputerEngineer gets promoted to Manager?

The type of this Employee cannot be both a ComputerEngineer and a Manager

Another example on composition vs inheritance

Using composition we could have:

Employee <has a> Role[]

What happens when a ComputerEngineer gets promoted to Manager?

The employee with the role computer engineer ads manager to its list of roles *in runtime.*

Example - program to a supertype

Abstractions lets us focus on the big picture (and forget about the details).

Polymorphism allows a reference of a supertype refer to a variety of objects of subtypes.

Sticking to the supertype protects code from changes.

A method can be more general accepting a Character (of any subtype) and doesn’t have to change when new types of Characters are added.

This requires that all subtypes behave like the Character superclass describes.

(Liskov’s substitution principle)

More examples of programming to supertype

The Character class in a game could have a weapon of interface type. This way, the code in the Character class doesn’t contain the implementation of the useWeapon() method. There is no need to add or change code in the Character class if the weapon usage is to change - the weapon can be replaced in runtime. The code in the Character class “programs to the supertype Weapon (which happens to be an interface).

There is no coupling between a character and for instance a Sword

Even more examples of programming to supertype

Imagine a method in some class which returns a list of objects in an application:

```
List objects = Storage.getList();
```

In the code above, we are programming against the interface List which will protect us against changes in the Storage class. We don’t know (and we don’t want to know) what type of concrete list we are getting from `getList()`.

The moment we start to care, we’ll be tempted to use this knowledge and become dependent on changes in the method.

Examples of Don’t repeat yourself

Consider these two constructors:

```
public Character(String name){
  this.name = name;
}

public Character(String name, Weapon weapon){
  this.name = name;
  this.weapon = weapon;
}
```

Examples of Don’t repeat yourself continued

We decide to throw a NullPointerException if name is null. How many places would we have to change? What happens if we forget one of them?

```
public Character(String name){
  this.name = name;
}

public Character(String name, Weapon weapon){
  this.name = name;
  this.weapon = weapon;
}
```

Examples of Don’t repeat yourself continued

We decide to throw a NullPointerException if name is null. How many places would we have to change? What happens if we forget one of them?

```
public Character(String name){
  if(name==null){
    throw new NullPointerException("Name cannot be null");
  }
  this.name = name;
}
public Character(String name, Weapon weapon){
  this.name = name;
  this.weapon = weapon;
}
```

Examples of Don’t repeat yourself continued

Solution:

```
public Character(String name){
  if(name==null){
    throw new NullPointerException("Name cannot be null");
  }
  this.name = name;
}

public Character(String name, Weapon weapon){
  this(name);
  this.weapon = weapon;
}
```

Insulate high level modules from low level ditto

Consider this:

```
List<Character> players = Storage.getCharacters();
```

Nice and insulated?

What about:

```
try{
  List<Character> players = Storage.getCharacters();
}catch(SQLException sqle){
  alertUser("Old game could not be loaded");
}
```

More about this in a coming lecture!

We'll talk more about how not to leak implementation details between layers in a coming lecture.