SENG330 - OO Design

Lecture 9 - OO Principles

Neil Ernst

Overview/Learning Objectives

- understand the importance of design
- introduce basic principles in OO design

> A good design principle should help generate ideas and enable you to think through design implications.
> — Rebecca Wirfs-Brock

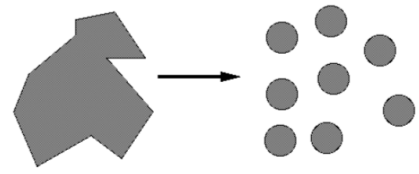
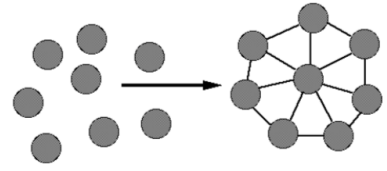
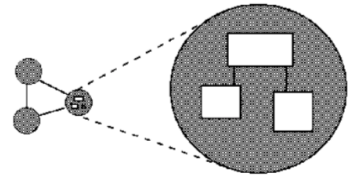
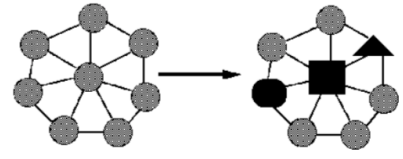
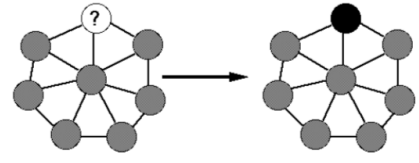
Writing vs. Coding

- As in English class, ultimately we care about *communicating*
- What are some key principles in writing well?
- What about writing code? What bothers you about bad code? What is "bad"?

Developing an Aesthetic

- Design and write a lot of code, and read even more code
- Some resources to help:
  - Beck, Implementation Patterns
  - Hunt and Thomas, Pragmatic Programmer
  - Martin, Clean Code
  - Bloch, Effective Java
  - Kernighan & Pflauger, Elements of Programming Style
  - Architecture of Open Source Applications • Beautiful Code
  - Various blogs and articles on places like InfoQ.

Design for Modularity

Decomposable - can be broken down into modules to reduce complexity and allow teamwork	
Composable - "Having divided to conquer, we must reunite to rule [M. Jackson]."	
Understandable - one module can be examined, reasoned about, developed, etc. in isolation	
Continuity - a small change in the requirements should affect a small number of modules	
Isolation - an error in one module should be as contained as possible	

taken from UW CSE331 - Hal Perkins

Cohesion and Coupling

- Cohesion is internal to an object, describing its independence. High cohesion implies doing one thing well.
- Coupling is the amount of dependency *between* components. High coupling makes the preceding modularity approaches harder.
- Modular systems have objects with high cohesion and low coupling

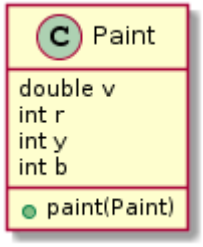
God Class

- the class that does it all
- common when moving from imperative languages or scripting
- low cohesion - does many things, not connected
- low coupling externally but now high coupling internally
- look for classes with many many methods.

Naming

- Surprisingly, one of the most important tasks developers have
- Rely on naming conventions (yours, project, language, corporate)
- Name classes to describe *effects* and *purpose*, not implementation

A DDD example




```
 public void paint (Paint paint) {
  v = v + paint.getV(); //after mixing, sum volume
 //complex color mixing logic
 // assign new r, b, y values
}
```

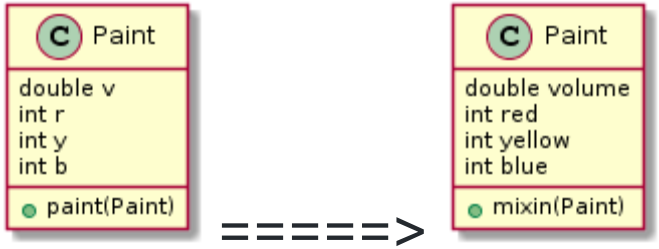
Unhelpful, written for the implementer

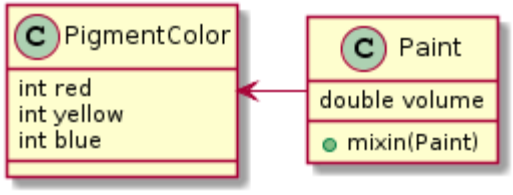
```
public void testPaint() {
  Paint yellow = new Paint(100.0, 0, 50, 0);
  Paint blue = new Paint(100.0, 0,0,50);
  yellow.paint(blue);
  //should get green with 200.0
}
```

Better - write from interface user point of view

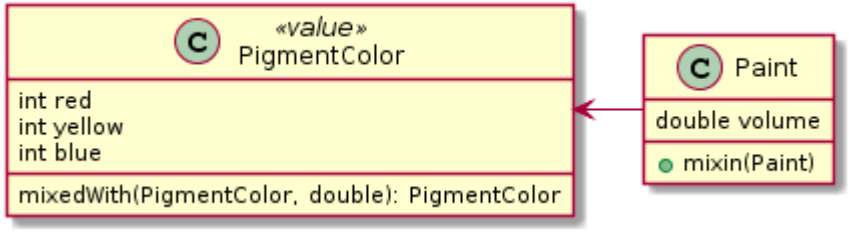
```
public void betterTestPaint() {
  Paint ourPaint = new Paint(100.0, 0, 50, 0);
  Paint blue = new Paint(100.0, 0,0,50);
  ourPaint.mixin(blue);
  //should get green with 200.0
  assertEquals(200.0, ourPaint.getVolume(), 0.01);
}
```

Improved class naming





One more refactoring




```
//in Paint
public void mixIn(Paint other) {
  volume = volume + other.getVolume();
  double ratio = other.getVolume()
  pigmentColor =
pigmentColor.mixedWith(other.pigmentColor(), ratio);
}
in PigmentColor
  // many lines of color-mixing logic
```

Naming

> For example, in the HotDraw drawing framework, my first name for an object in a drawing was DrawingObject. Ward Cunningham came along with the typography metaphor: a drawing is like a printed, laid-out page. Graphical items on a page are figures, so the class became Figure. In the context of the metaphor, Figure is simultaneously shorter, richer, and more precise than DrawingObject."

If a class is hard to name, it is probably doing too much.

Naming

Rate of change:

> Things that change at the same rate belong together. Things that change at different rates belong apart. This principle is true of both the data manipulated by the program--two variables that always change at the same time belong in the same object--and the structure of the program--two functions that change at the same time belong in the same class.

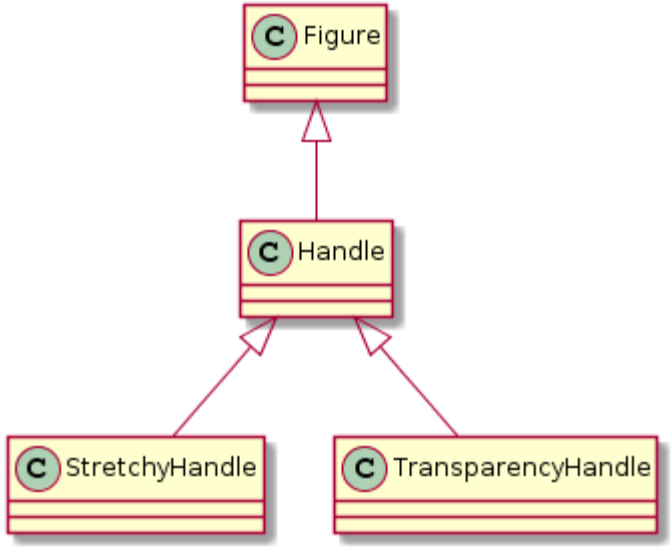
"Empathy in naming" -> "if I searched for this function, what would I want it called?"

https://www.facebook.com/notes/kent-beck/naming-from-the-outside-in/464270190272517/

Subclasses

Need to communicate what class they are like and how they are different.

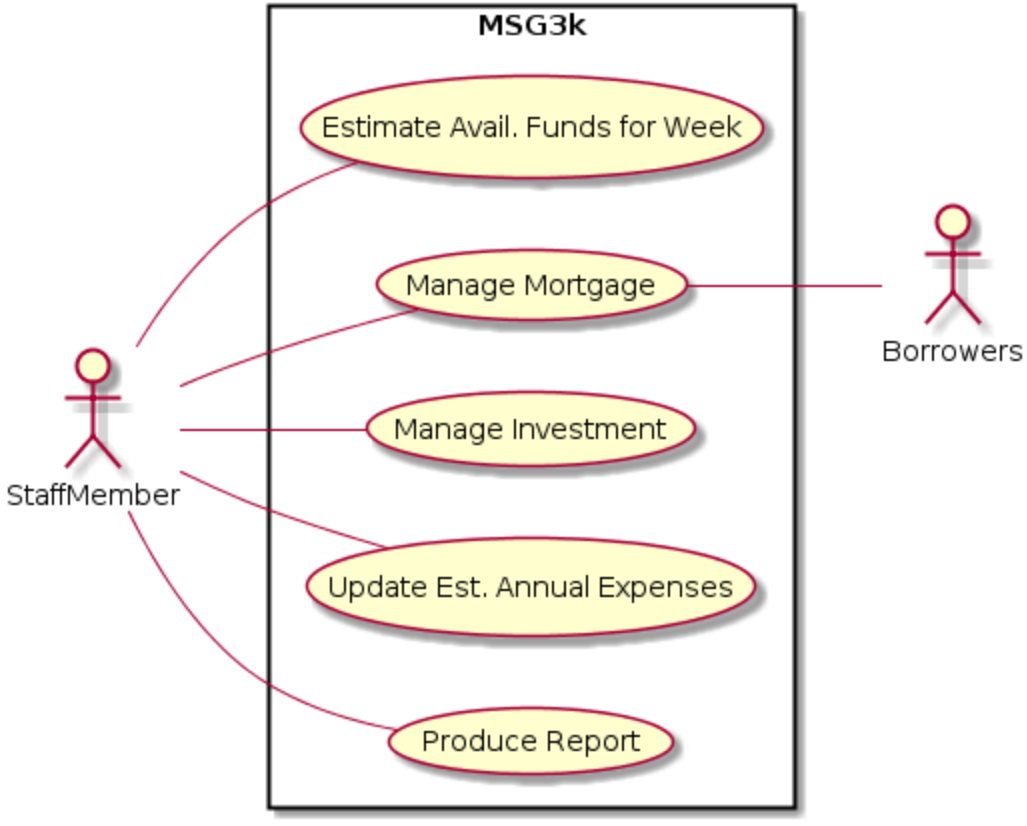
- Figure has a Handle
- Handle has different subclasses of Handle



MSG Case Study: Mortgages

The MSG foundation lends money to first time borrowers to help them purchase a home. The buyers put a *down payment* of their own money, say 10% of the purchase cost. The foundation loans them the rest and charges interest over a period of time below bank rates. The foundation wants to automate their business.

- Mortgagees sign a document giving the foundation their home as security.
- Mortgagees have to pay insurance and taxes into escrow.
- To qualify mortgagees have to have a certain income.



Key OO Principles - Overview

1. Cohesion/Coupling
2. Polymorphism
3. SOLID
    i. Single Responsibility
    ii. Open/Closed
    iii. Liskov Substitution
    iv. Interface Segregation
    v. Dependency Inversion/Inversion of Control
4. Refactoring

Cohesion and Coupling

- Cohesion is internal to an object, describing its independence. High cohesion implies doing one thing well.
- Coupling is the amount of dependency *between* components. High coupling makes the preceding modularity approaches harder.
- Modular systems have objects with high cohesion and low coupling

Polymorphism

"many shapes"

- refresher: types, inheritance, and classes in Java -> File example

Benefits

- decouple runtime resolution from compile time resolution ('late binding')
    - in Java, with reflection APIs
- extensibility of the design

Drawbacks are comprehensibility and potential for overuse

Single Responsibility

- CRC diagram for MSG case -> "A responsibility is anything that a class knows or does"
- a class should have a single purpose, collecting together a set of related sub-responsibilities
- "class only has a single reason to change"
- a Rectangle that calculates *area* and *draws itself*
- what two types of responsibility should almost certainly be separated?

Open Closed Principle

"Objects are open for extension but closed for modification."

- use inheritance and abstraction

Ask: if a new object of type X is needed, where do you make changes?

- minimize changes to existing objects
- contrast: lengthy `switch` statements (or using `instanceof`)

Encapsulation: keep data and operations together. Ensure internal details don't have external effects (eg. member variables)

- Does inheritance violate this?

Liskov Substitution Principle (LSP)

"Functions that use Base Classes must be able to use Derived Classes without being aware of it"

- a change to a derived class (subtype) shouldn't affect the program using the base class (OCP)

> The validity of a model can only be expressed in terms of its clients (tests).

Interface Segregation

"Clients should not depend on interfaces they don't use"

- do not create coupling between objects through *implicit dependencies* on a common interface (rich interface)

Dependency Injection/Inversion of Control

> A. High Level Modules Should Not Depend Upon Low Level Modules. Both Should Depend Upon Abstractions.
>
> B. Abstractions Should Not Depend Upon Details. Details Should Depend Upon Abstractions."

Separate configuration from use, especially in the use of 3rd party libraries.

DI Frameworks

Frameworks like Guice, Spring allow for "injection" of the dependencies

- e.g., at runtime, allow new instances of Report
- especially useful for testing (code no longer cares if it gets a mock or the real object)

```
public class RealBillingService implements BillingService {
  public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
  CreditCardProcessor processor = new PaypalCreditCardProcessor();
  TransactionLog transactionLog = new DatabaseTransactionLog();
```

DI - 2

```
public class BillingModule extends AbstractModule {
 @Override
 protected void configure() {
  bind(CreditCardProcessor.class).to(PaypalCreditCardProcessor.class);
   // more bindings
}
...
public class RealBillingService implements BillingService {
  @Inject
  public RealBillingService(CreditCardProcessor processor, TransactionLog
transactionLog) {
   this.processor = processor;
  this.transactionLog = transactionLog;
}
```

Refactoring

refactor if there is a symptom, not just for the sake of it. (somewhat violated in learning refactoring!)

Exercise

Critique and refactor/redesign the code.

> A program to calculate and print a statement of a customer's charges at a video store. The program is told which movies a customer rented and for how long. It then calculates the charges, which depend on how long the movie is rented, and identifies the type movie. There are three kinds of movies: regular, children's, and new releases. In addition to calculating charges, the statement also computes frequent renter points, which vary depending on whether the film is a new release.