In the “real world” we’re taught that design is pretty. Here we learn that design is about productivity.
Introduction to Design

Concepts and Diagnosis
Some things we’d like to be true

Mostly about easing CHANGE

- Easy to find what code to modify to add a feature
- ...I only have to modify one class (in addition to writing the new code)
- ...It’s easy to understand the class I have to change
- ...My teammate can add another feature without us colliding or stopping working to talk
- ...When I test my code, nobody else’s code needs to work

Good software design gets us close to these ideals
First, Diagnosis

- We can’t cure the patient until:
  - We know what’s wrong
  - Just how healthy s/he can be

- So, before I teach you how to design

- Let’s take a look at:
  - What some bad designs looks like
  - What some good designs looks like
Two Diagnostics for Good Design

- **Single Responsibility Principle (SRP)**
  - Each class should be responsible for one thing (capability, entity, computation, etc.)
  - Can phrase this as “mind your own business”
    - object do its own calculations
    - object should not do calculations for another
  - Easy to violate this because objects need to be connected to one another
    - e.g., Events happen as part of Dates

- **Don’t Repeat Yourself (DRY)**
  - Each computational idea should be expressed just once
  - Violations often the result of
    - cut-and-paste programming
    - incomplete class (others have to do calculations for it, which also violates SRP)
  - But also over-specialization of classes (class = object)
A Concise Theory of Object-Oriented

- **Object represents a “thing”**
  - *person, car, date, ...*
  - *(not two things, not ½ thing)*

- **Object responds to messages**
  - *(method calls)*
  - *Things it does to itself (SRP)*
  - That is, other objects ask the object to do something to itself

- **Objects are “opaque”**
  - Can’t see each others’ data/vars
  - Messages (calls) are only way to get things done
Because objects are completely opaque, we don’t need to know what’s really inside them.
- Each car object could be implemented with its own unique code.
- If two cars behave the same, then really should have same code.
  - Otherwise would violate DRY.
  - And a huge amount of coding work.
- So all cars are made from a common car template.
  - Template = class.
  - The car template is not a car, it’s a “blueprint” for a car.
  - Helps satisfy DRY.
So OO Design is easy, right? Uh, no.

Tendency is to cram “related” functionality into existing classes, rather than creating new ones.
SRP design has classes for “doers”

The four misplaced methods

Now Automobile has only a single responsibility: dealing with its own basic functions.

It's a driver's responsibility to drive the car, not the automobile itself.

A CarWash class can handle washing an automobile.

A mechanic is responsible for changing tires and checking the oil on an automobile.
So OO Design is easy, right? No. (pt II)

A related challenge is when two classes closely collaborate, like the iSwoon Date & Event classes.
class Date {

protected static ArrayList<String> allowedEvents; /* override in sub */
protected ArrayList<Event> events = new ArrayList<Event>();

public void seeMovie() {
    Event event = new seeMovieEvent();
    if (validateEvent(event))
        events.add(event);
    else
        throw eventNotAllowedOnDateEvent(event, this);
}

public void goToRestaurant() {
    Event event = new goToRestaurantEvent();
    if (validateEvent(event))
        events.add(event);
    else
        throw eventNotAllowedOnDateEvent(event, this);
}

public void orderFlowers() {
    Event event = new orderFlowersEvent();
    if (validateEvent(event))
        events.add(event);
    else
        throw eventNotAllowedOnDateEvent(event, this);
}

public boolean goOnDate() { /* important code here */ }

Repetition (violates DRY)
This code violates SRP. Why?

A. reuses the responsibility of which events go with which date

B. it checks validity of events and also stores list of events

Better phrasings:
A. Date does not “validates-events itself”
B. Changes to Event (like adding new event type) requires changing Date
Responsibility for Events (violates SRP)

Not just calling event method (that's OK), but calculating on event data to derive event property.

Also note that the only difference between subclasses is a constant data value.
Repetition (violates DRY)

Also note that only difference in subclasses is a constant
Refactored iSwoon Design

```java
class Date {
    protected int dateNum;
    protected ArrayList<Event> events = new ArrayList<Event>();

    protected Date(int dateNumber) {
        dateNum = dateNumber;
    }

    public void addEvent(Event event) {
        if (event.dateSupported(dateNum))
            events.add(event);
        else
            throw eventNotAllowedOnDateEvent(event, this);
    }

    public boolean goOnDate() { /* important code here */ }
}
```

- No class for each date!
- Replaces 3 Event constructors
class Event {

protected String name;
protected int firstAllowedDate = Integer.MAX_VALUE; // fail hard if no init

public Event(int eventsFirstAllowedDate, String eventName) {
    firstAllowedDate = eventsFirstAllowedDate;
    name = eventName
}

protected boolean dateSupported(int dateNumber) {
    return dateNumber >= firstAllowedDate;
}

/*
 * static Factory methods, for convenience and correctness.
 * Note that Date can't even tell if Event has subclasses.
 */

public static Event makeSeeMovie() { return new Event(1, "SeeMovie"); }

public static Event makeGoToRestaurantEvent() {
    return new Event(1, "GoToRestaurant");
}

public static Event makeOrderFlowers() {
    return new Event(2, "OrderFlowers");
}

}
Why is it OK to have `dateSupported(int)` in `Event`, but not `validateEvent(Event)` in `Date`?

A. **Because whether an Event is allowed is a property of the Event itself, not the Date**

B. The only thing that’s going to use a Date is an Event

C. Dates are hard numbers, while events are flexible

D. You wouldn’t have to change bit of code if you were to add another valid Event
Why is it OK to have `dateSupported(int)` in Event, but not `validateEvent(Event)` in Date? **My answers:**

A. `dateSupported` is checking the appropriateness of an *individual* Event for how mature the relationship is (not about the ordering of selected events)

B. Said another way: Date class is about the ordering of selected Events for a date (not event validity)

C. `dateSupported` is computing on an int, not a Date
Design Diagnosis Review

- Three common mistakes in design
  - TOO MUCH: Put all X-related functionality in class X (Automobile)
  - TOO FRIENDLY: Blending of closely related classes (Date & Event)
  - TOO LITTLE: Defining object-like classes (Date & Event)

- A few diagnostic techniques
  - SRP: do the “_____ itself” test on methods
  - SRP: a change in one class causes change in another class
  - DRY: repetitive code
  - DRY: A “small” change requires many similar changes across methods or classes
  - Constant Classes: Only diff. between classes is constants (same methods)

- Repairs to design
  - For non-SRP functionality
    - Create additional classes, move there (Automobile)
    - Move into existing classes (Date & Event)
  - DRY: Create new method out of repetitive code, call it
  - Merge repetitive, similar classes and encode differences with variables
Take-Aways from Class Today

- Object-oriented design is intuitive, but subtle
  - Java is just a tool, does not guarantee good design
    - (Just because I have an expensive camera does not make me a good photographer :)
  - Easy to put functionality in wrong place, make classes too big, or make too small

- Possible to diagnosis and repair a design before or after the coding (may require both)
  - SRP, DRY
  - Change in one class affects another (SRP)
  - Small change affects multiple classes or methods

- Unfortunately, there are many kinds of design mistakes, and unique repairs for them