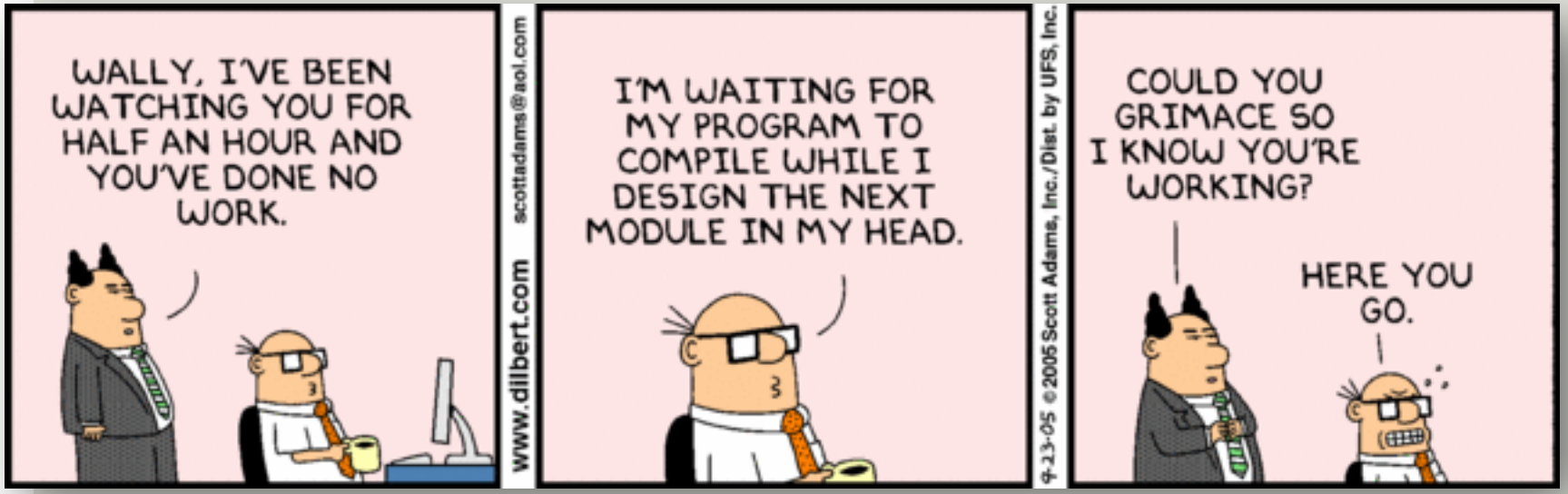
Object-Oriented Design



Slide credit: William Griswold

Object-Oriented Design

It's still about tradeoffs.

Some things we'd like to be true

- My teammate and I can each **add a feature** in parallel without us colliding or stopping to talk
- When I test my code, nobody else's code needs to work or even be written

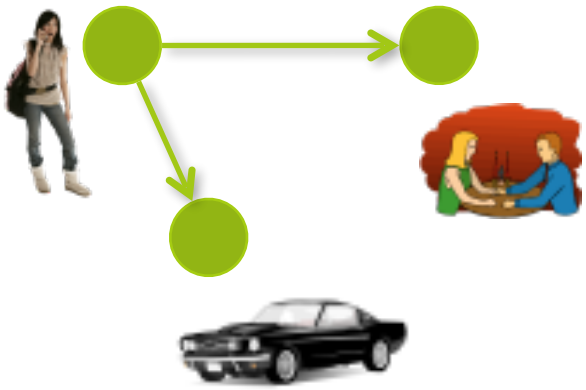
Modularity is about Teamwork!

- Easy to find **where to add code** to add a feature
- **Mostly adding code** and not modifying code
- Easy to understand the class I do have to **change**

Good software design gets us close to these ideals

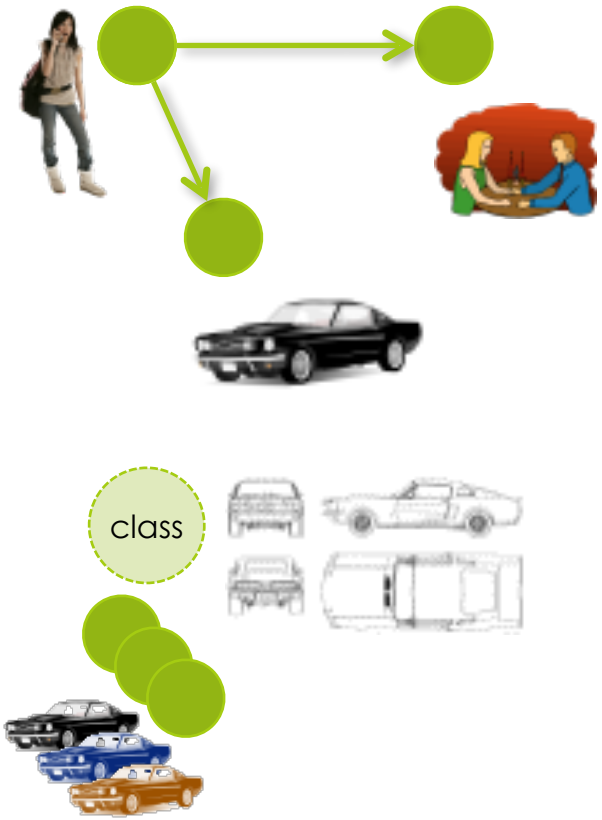
And yes, we'll be making a big project feel small

A Concise Theory of Object-Oriented



- Object represents a "thing"
  - *person, car, date, …*
  - (not two things, not ½ a thing)
- Object responds to *messages*
  - (method calls)
  - *Things it does to itself*
  - That is, other objects ask the object to do something to itself, with msg
- Objects are "opaque"
  - Can't see each others' data/vars
  - Messages (calls) are only way to get things done

A Concise Theory of Object-Oriented, II



- Because object is completely opaque, others don't need to know what's really inside it
  - Each *car* object could be implemented with its own unique code
- If two cars behave the same, then really *should* have same code
  - Otherwise a huge amount of coding
  - Each one would have to be tested
  - Creates a maintenance nightmare
- 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

Goals for today

- See how to use these two OOP principles to improve your designs:
  - SRP: Single Responsibility Principle
  - DRY: Don't Repeat Yourself
- Other principles from SOLID:
  - Open-closed principle
  - Liskove substitution principle
  - Interface segregation principle
  - Dependency inversion principle

Open-Closed principle

- Objects should be open for extension but closed for modification
- i.e. enable extending class without modifying the class

Liskov Substitution Principle

- Properties of a class should hold of subclasses
- i.e. anyone expecting a Super should be OK when receiving a Sub

Interface Segregation Principle

- Clients shouldn't have to implement interfaces they don't use
- Clients shouldn't have to depend on methods they don't use
- ShapeInterface includes area()
- But 3D shapes also include volume()
- Don't add volume() to ShapeInterface

Dependency Inversion Principle

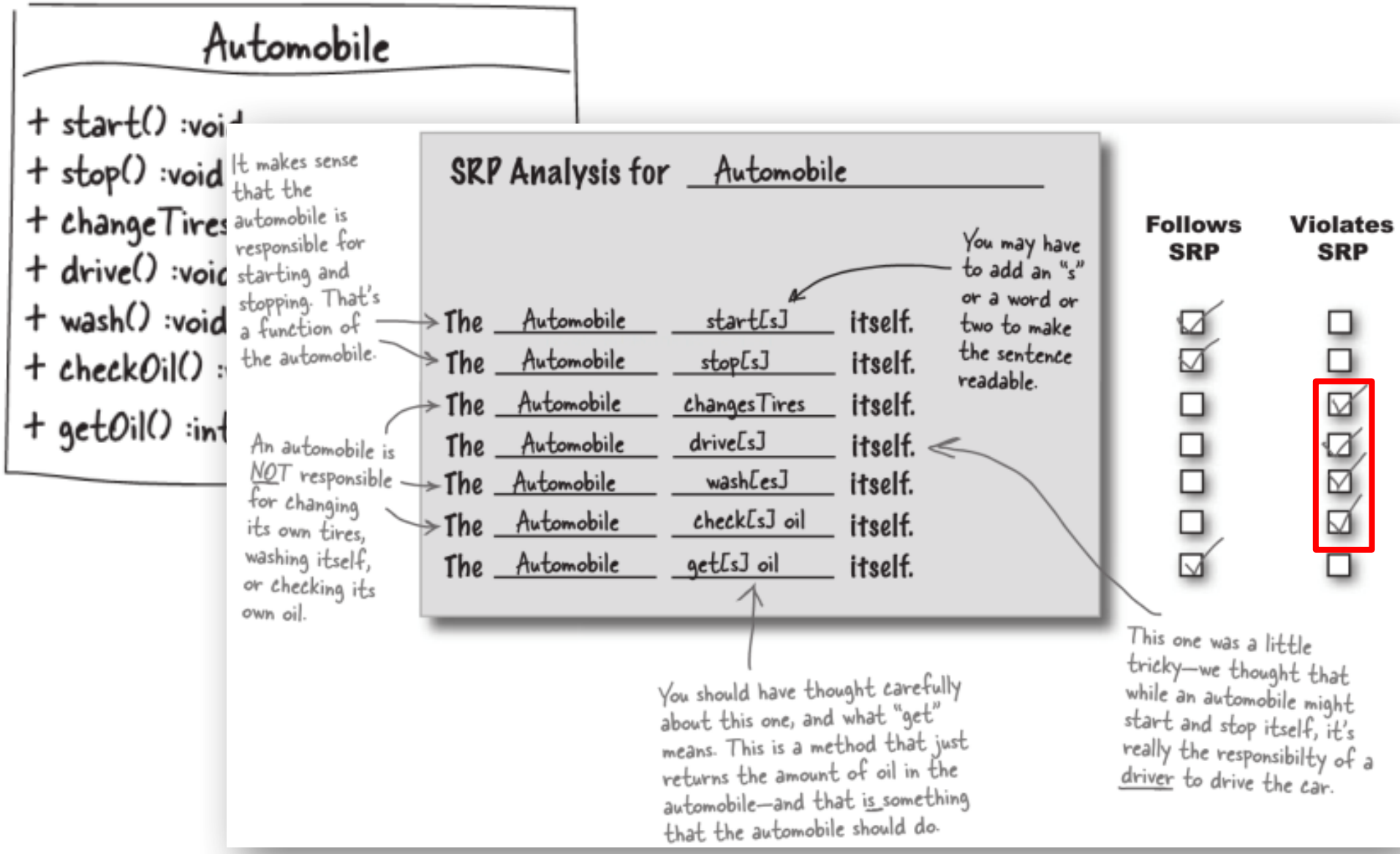
- Depend on abstractions (interfaces), not concrete implementations
- Goal is to avoid tight coupling

For rest of today: DRY and SRP

Thing-ness Simplified: The **S**ingle **R**esponsibility **P**rinciple (**SRP**)

- A class should be responsible for 1 thing (thing, capability, computation, etc.)
- Can phrase as "*mind your own business*"
  - object does its own calculations
  - object should not do calculations for another
- Easy to violate this because objects need to be connected to one another
  - If you want something done, you just do it (oops)

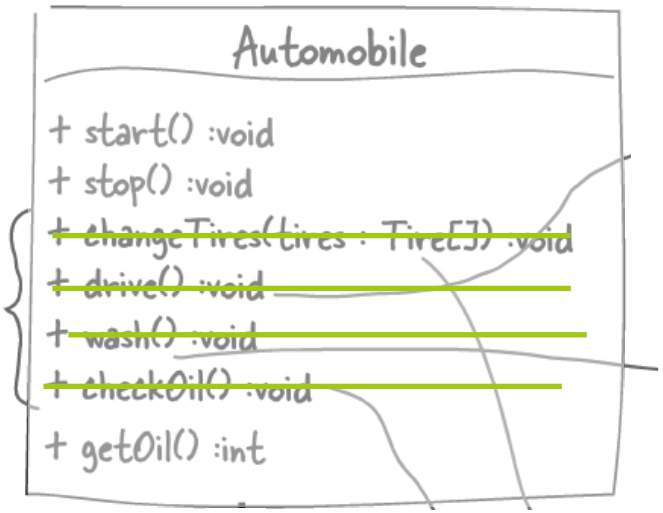
Un-thing-ness: cramming related functionality into a single class



SRP design has separate classes for "do-ers"

One big class into four smaller ones = making a big project act like a small one

The four misplaced methods



This is called *Refactoring*.

New Design is Better

- For change, you know **where to find** code
  - Changing Mechanic stuff? Look in Mechanic
  - In old design, could overlook Automobile, means bug
- Only **one locus of change**
  - Don't have to think about, or change, Automobile and Mechanic
  - Simpler change, fits on screen, less chance of bug
  - Can think of your big program as bunch of small ones
- Design matches world, so **easier to understand**
  - More later

People are Complicated

Consider this Java class, which is using good naming conventions to convey the meanings of the methods:

```
class Person {
  public void rainOn();
  public boolean isWet();
  public String getSpouseName();
  public boolean isLeftHanded();
}
```

Which methods are SRP?

A. rainOn(), isLeftHanded()

B. isWet(), getSpouseName()

C. isWet(), isLeftHanded()

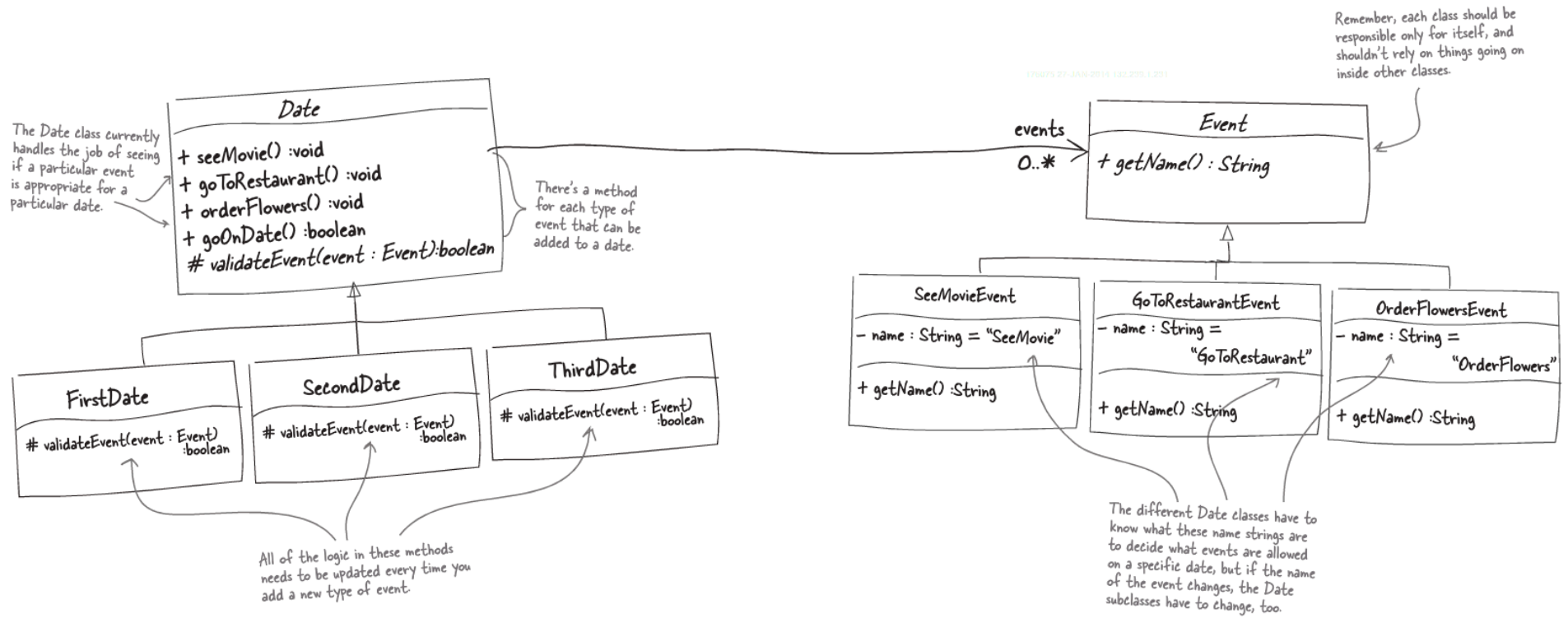
D. getSpouseName(), isLeftHanded()

D is tempting, but the fact that we're getting the name from the Spouse object is the give-away: the Spouse should be asked for its name directly. (Later we'll see that the spouse shouldn't be stored in the Person class at all.)

Thing-ness Simplified: **D**on't **R**epeat **Y**ourself (**DRY**)

- **Each "thing" or computational idea should be expressed just once**
- Violations are 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 (implement object as a class)

Un-thing-ness: over-collaborating classes



~/documents/110/iSwoon/Original

```
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)

~/documents/110/iSwoon/Original

```
protected boolean validateEvent(Event event) {
  for (String eventName : allowedEvents)
    if (eventName.equals(event.getName())) return true;
  return false;
}
}


class FirstDate extends Date {

protected static ArrayList<String> allowedEvents =
  new ArrayList<String>(Arrays.asList("SeeMovie", "GoToMovie"));

public FirstDate() {}

}

class SecondDate extends Date {

protected static ArrayList<String> allowedEvents =
  new ArrayList<String>(Arrays.asList("SeeMovie", "GoToMovie", "OrderFlowers
"));

public SecondDate() {}

}

class ThirdDate extends Date {

protected static ArrayList<String> allowedEvents =
  new ArrayList<String>(Arrays.asList("SeeMovie", "GoToMovie", "OrderFlowers
"));
```

This code violates SRP. Why?

~/documents/110/iSwoon/Original

```
protected boolean validateEvent(Event event) {
  for (String eventName : allowedEvents)
    if (eventName.equals(event.getName())) return true;
  return false;
}
}


class FirstDate extends Date {

protected static ArrayList<String> allowedEvents =
  new ArrayList<String>(Arrays.asList("SeeMovie", "GoToMovie"));

public FirstDate() {}

}

class SecondDate extends Date {

protected static ArrayList<String> allowedEvents =
  new ArrayList<String>(Arrays.asList("SeeMovie", "GoToMovie", "OrderFlowers
"));

public SecondDate() {}

}

class ThirdDate extends Date {

protected static ArrayList<String> allowedEvents =
  new ArrayList<String>(Arrays.asList("SeeMovie", "GoToMovie", "OrderFlowers
"));
```

It's OK to call Event method, but not **calculating on event data to derive event property**

Responsibility for Events (violates SRP)

Also note that the only difference between subclasses is a constant data value

~/documents/110/iSwoon/Original

```
class Event {

protected static String name;

public String getName {
  return name;
}

class SeeMovieEvent extends Event {

protected static String name = "SeeMovie";

public SeeMovieEvent() {}

}

class GoToRestaurantEvent extends Event {

protected static String name = "GoToRestaurant";

public GoToRestaurantEvent() {}

}

class OrderFlowersEvent extends Event {

protected static String name = "OrderFlowers";

public OrderFlowersEvent() {}

}
~
```

Repetition (violates DRY)

Also note that only difference in subclasses is a constant

Refactored Date Class

~/documents/110/iSwoon/RefactoredForSRPandDRY

```
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 */ }

}
~
~
~
~
~
~
~
~
~
~
```

Number instead of class for each date!

Replaces 3 Event constructors

~/documents/110/iSwoon

Refactored Event

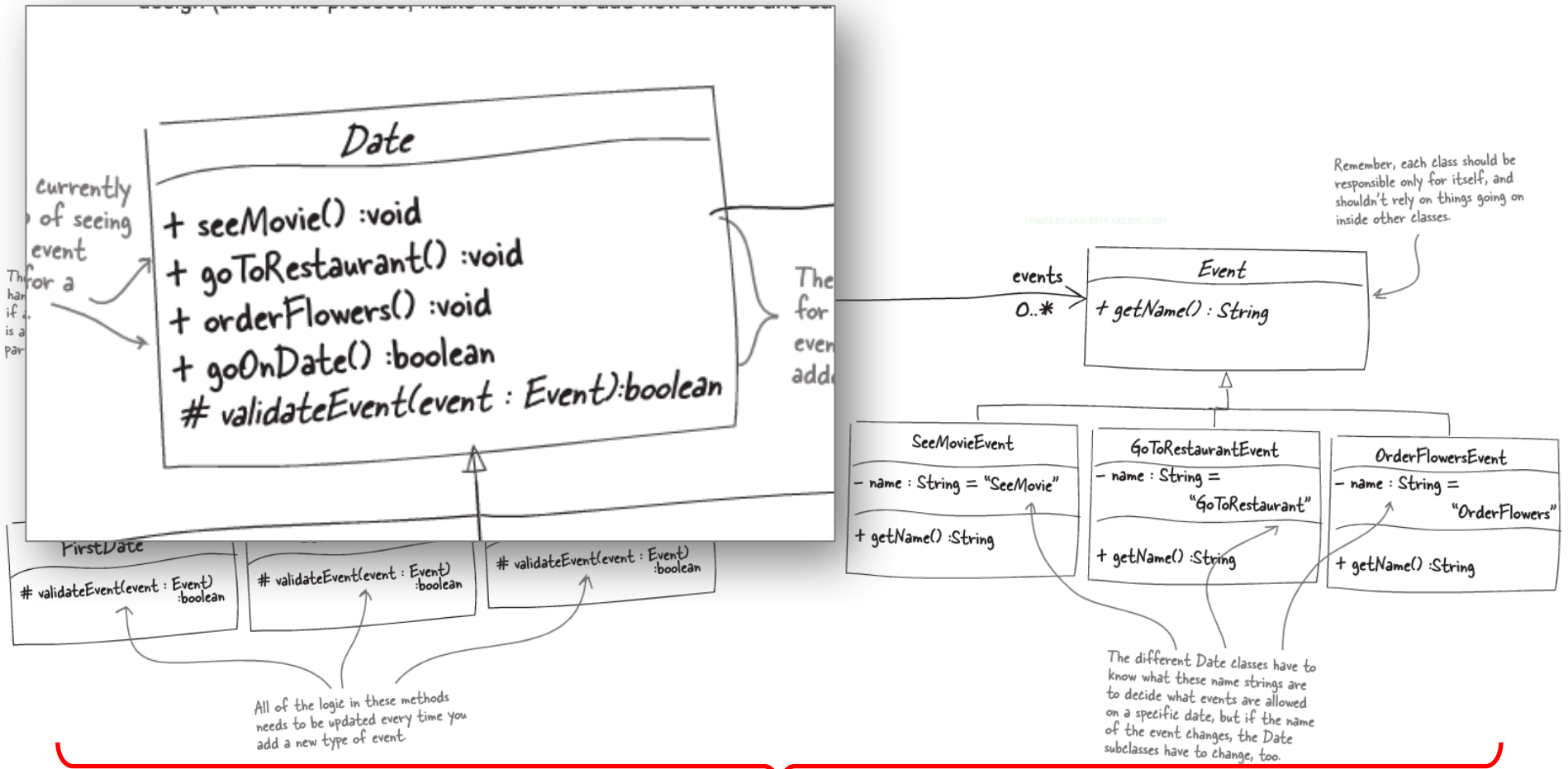
String, not class for each event!

```
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");
}

}
```

Moved from Date to get SRP.

"Factory" Methods keep Event details local

Rewind:
Now we can **see** symptoms in the UML



These classes sound like objects

~/documents/110/iSwoon/RefactoredForSRPandDRY

```
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;
}
```

But now **date** functionality here! Why OK?

Which of these is a wrong justification for dateSupported(int) is OK in Event, but validateEvent(Event) is not OK in Date?

A. The only thing that's going to use a Date is an Event

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

C. dateSupported is computing on an int, not a Date

D. You wouldn't have to change any code if you were to add another valid Event

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 class that has only one object (Date & Event)
- **SRP**: The Single Responsibility diagnostic
  - Do the "____ itself" test on methods
  - A change in one class causes change in another class
- **DRY**: The Don't Repeat Yourself diagnostic
  - Repetitive code
  - A "small" change requires many similar changes across methods or classes
- **Constant Classes**: Only diff. between classes is constants (same methods)

Design *Repair* Review

- For SRP-violating functionality
  - Create additional classes, move violations there (Automobile)
  - Move into existing classes (Date & Event)
- For DRY-violating functionality
  - Create new method out of repetitive code, call it
- For repetitive/constant classes
  - Merge repetitive, similar classes and encode differences with variables
  - static String name = “SeeMovie”; → String name;

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**: shared responsibility requires two classes to change together
  - **DRY**: duplicated code requires multiple methods/classes to change **[to be continued]**
- Unfortunately, there are many kinds of design mistakes, and unique repairs for them