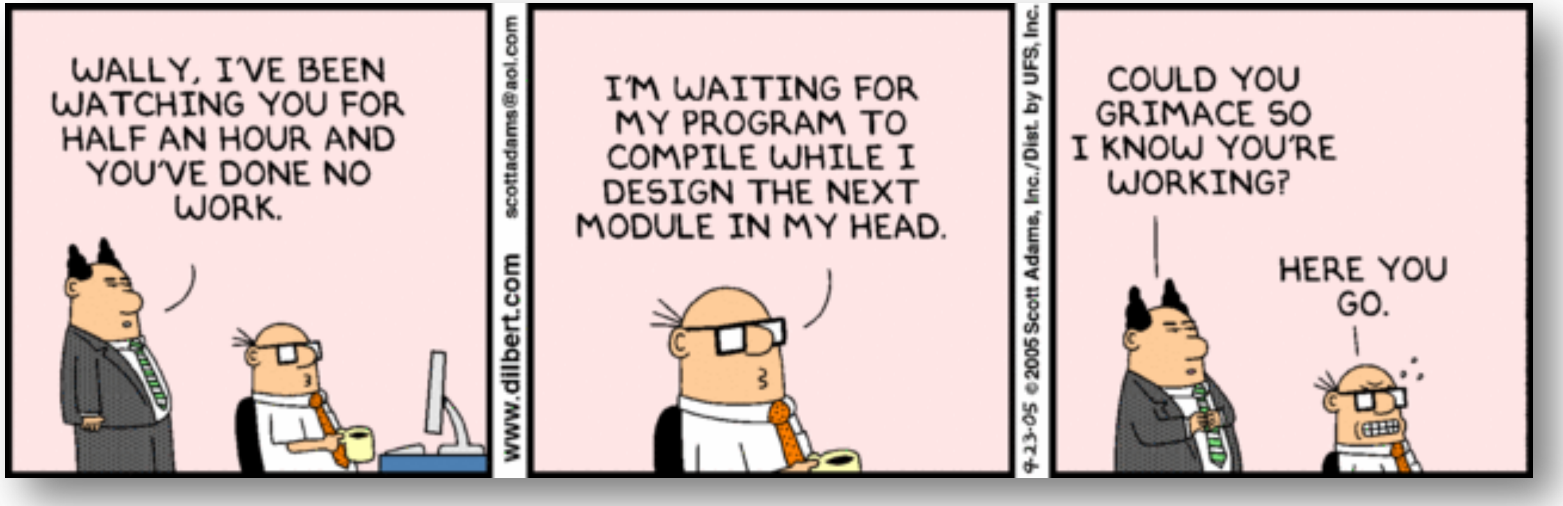
SOFTWARE DESIGN: DESIGN CONCEPTS

Content from Chapter 5 of “Head First Software Development”, Pilone et al.

Miami University Software Technology & Analysis Group (MUSTANG)
Computer Science & Software Engineering
Miami University, Oxford, Ohio, USA

Review

- In the “real world” we’re taught that design is *pretty.*
- Here we learn that design is about *productivity*.


WALLY, I'VE BEEN WATCHING YOU FOR HALF AN HOUR AND YOU'VE DONE NO WORK.
www.dilbert.com
scottadams@aol.com
I'M WAITING FOR MY PROGRAM TO COMPILE WHILE I DESIGN THE NEXT MODULE IN MY HEAD.
9-23-05 © 2005 Scott Adams, Inc./Dist. by UFS, Inc.
COULD YOU GRIMACE SO I KNOW YOU'RE WORKING?
HERE YOU GO.

LAST TIME

Wow, that's not good...a single change means we have to mess with a bunch of classes. Can't we do something about that in our design?

SOME THINGS WE'D LIKE TO BE TRUE

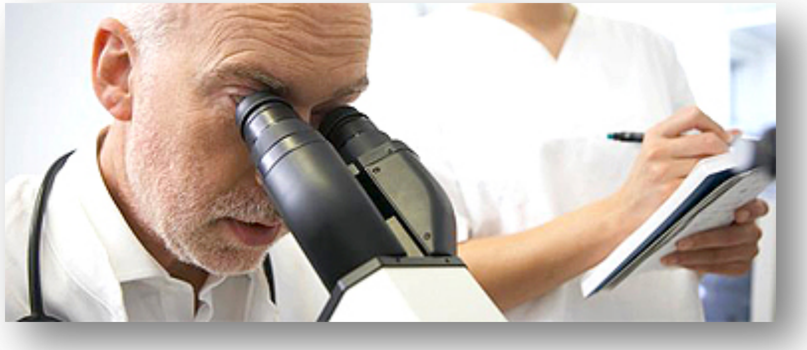
- **Mostly about easing CHANGE**
    - Easy to find what code to **modify** to add a feature
    - …I only have to **modify** one class (in addition to writing the new code)
    - …It's easy to understand the class I have to **change**
    - …My teammate can **add another feature** without us colliding 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**

SOME (MORE) THINGS WE'D LIKE TO BE TRUE

- I can drive the design from the user stories
- The code just “writes itself”
- My code will be easy to understand by the team
- My code will be SRP and DRY

FIRST, DIAGNOSIS

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

1. **S**ingle **R**esponsibility **P**rinciple (**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
2. **D**on't **R**epeat **Y**ourself (**DRY**)
   - Each computational idea should be expressed just once
   - Violations often the result of
     - cut-and-paste programming (code clones)
     - incomplete class (others have to do calculations for it, which also violates SRP)

1. Single Responsibility Principle (SRP)

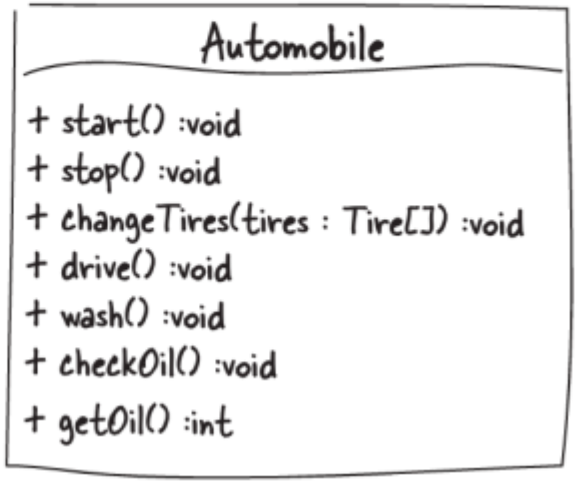
- Well-designed classes are singularly focused.
- Problem in this design is that for any particular behavior—like sending flowers—the logic for that behavior is spread out over a lot of different classes.
- So what seems like a simple change, turns into a multi-class mess of modifications.
- Every object in your system should have a **single responsibility**, and all the object’s services should be focused on carrying out that single responsibility.
  - You’ve implemented the single responsibility principle correctly when each of your objects has only one reason to change.
- Aka Cohesion: A class should represent a single concept only.

MIGRATING TO SINGLE RESPONSIBILITY

- Class Automobile
  - Methods:
    - **start(): void** // the start process of the car
    - **stop: void** // the car turning off process
    - **changeTires(tires : Tire[]) : void**
    - **drive() : void** //assume not an auto-driving Tesla
    - **wash (): void**
    - **checkOil() : void** //checking the physical oil
    - **getOil(): int** // get the numeric amount of oil left

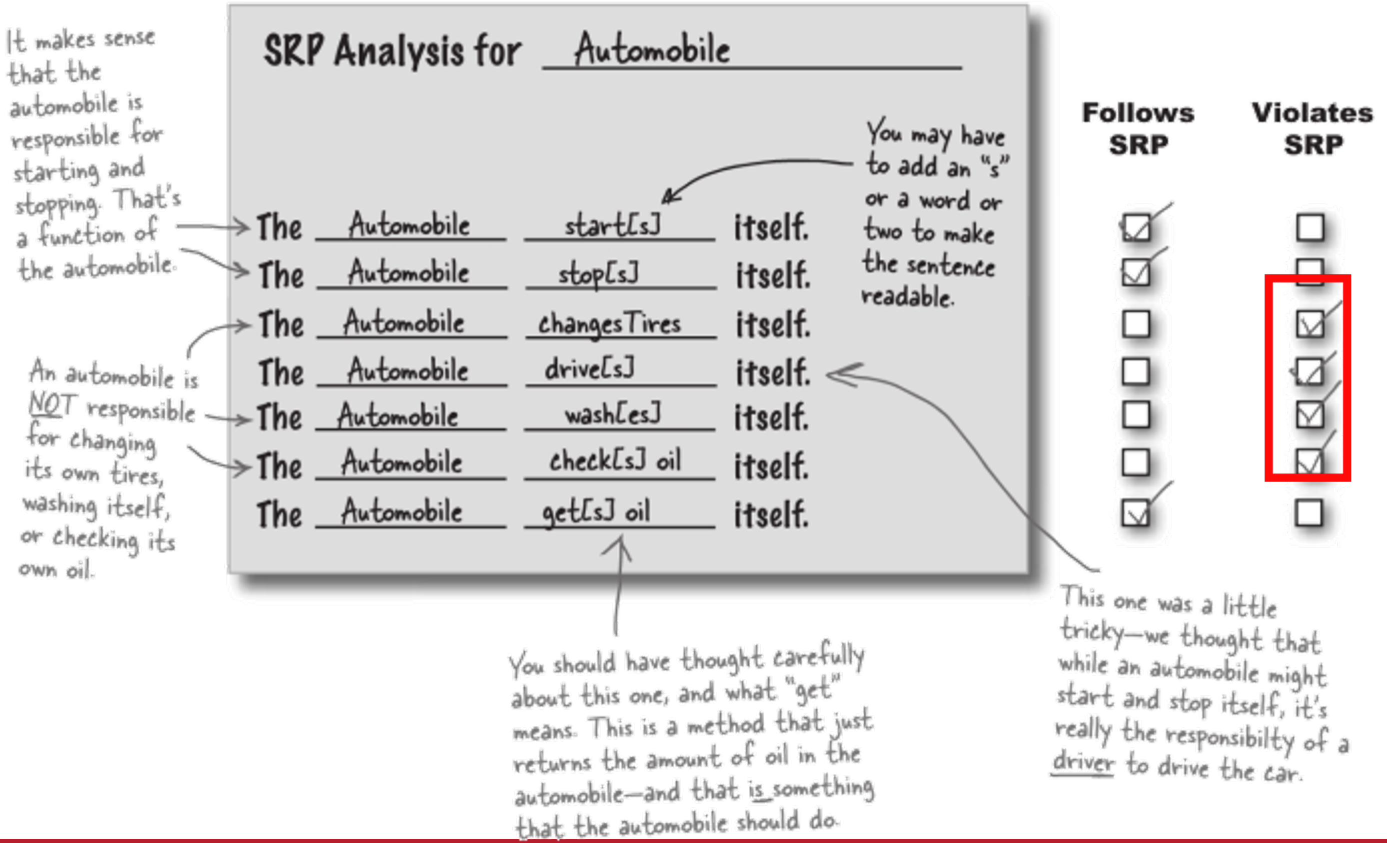
So OO Design is easy, right? Uh, no.

- The tendency is to cram “related” functionality into existing classes, rather than creating new ones



Automobile

```
+ start() :void
+ stop() :void
+ changeTires(tires : Tire[]) :void
+ drive() :void
+ wash() :void
+ checkOil() :void
+ getOil() :int
```



SRP Analysis for Automobile

				Follows SRP	Violates SRP
The	Automobile	start[s]	itself.	☑	☐
The	Automobile	stop[s]	itself.	☑	☐
The	Automobile	changesTires	itself.	☐	☑
The	Automobile	drive[s]	itself.	☐	☑
The	Automobile	wash[es]	itself.	☐	☑
The	Automobile	check[s] oil	itself.	☐	☑
The	Automobile	get[s] oil	itself.	☑	☐

It makes sense that the automobile is responsible for starting and stopping. That’s a function of the automobile.

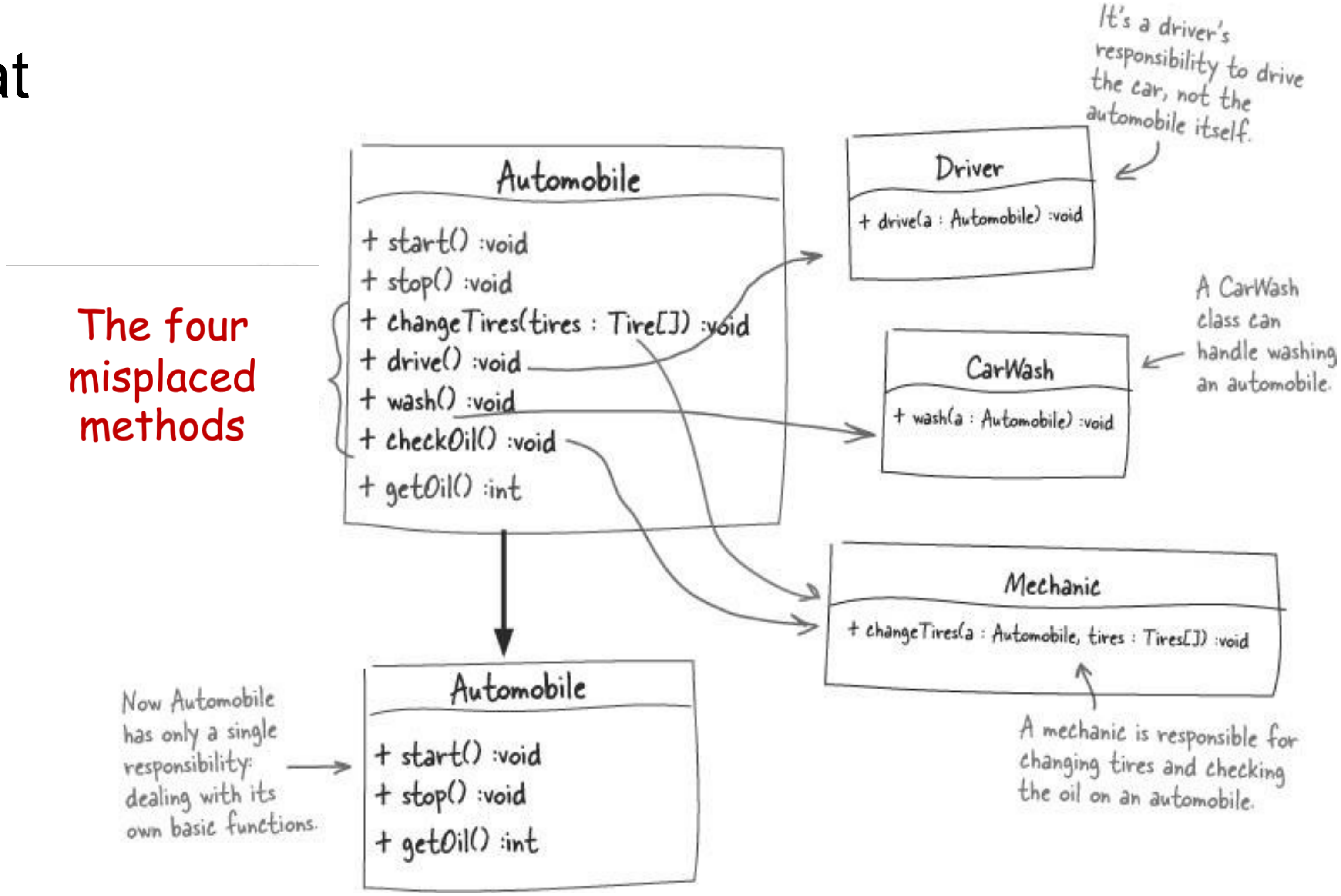
An automobile is NOT responsible for changing its own tires, washing itself, or checking its own oil.

You may have to add an “s” or a word or two to make the sentence readable.

You should have thought carefully about this one, and what “get” means. This is a method that just returns the amount of oil in the automobile—and that is something that the automobile should do.

This one was a little tricky—we thought that while an automobile might start and stop itself, it’s really the responsibilty of a driver to drive the car.

- Take all the methods that don’t make sense on a class and move them to an appropriate class where it does make sense.



The four misplaced methods

Automobile
+ start() :void
+ stop() :void
+ changeTires(tires : Tire[]) :void
+ drive() :void
+ wash() :void
+ checkOil() :void
+ getOil() :int
Driver
+ drive(a : Automobile) :void
It's a driver's responsibility to drive the car, not the automobile itself.
CarWash
+ wash(a : Automobile) :void
A CarWash class can handle washing an automobile.
Mechanic
+ changeTires(a : Automobile, tires : Tires[]) :void
A mechanic is responsible for changing tires and checking the oil on an automobile.
Automobile
+ start() :void
+ stop() :void
+ getOil() :int
Now Automobile has only a single responsibility: dealing with its own basic functions.

2. Don’t Repeat Yourself (DRY)

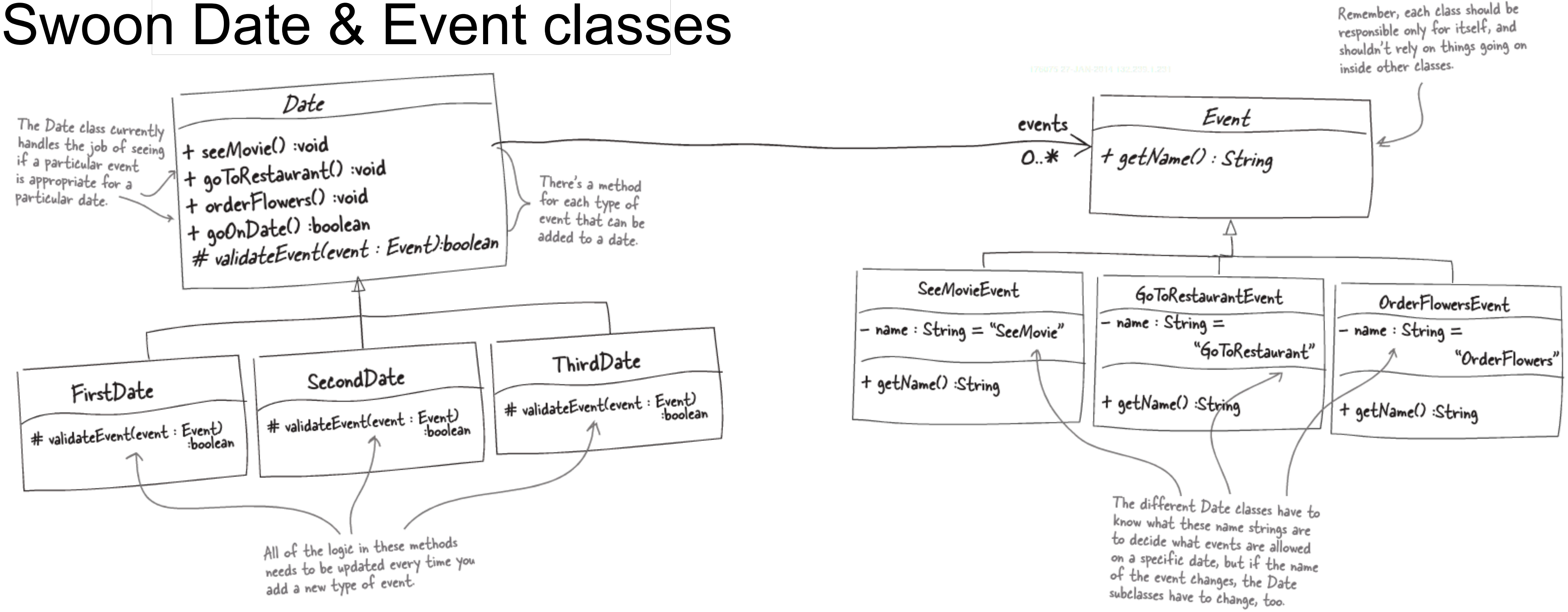
- Avoid duplicate code by abstracting or separating out things that are common and placing those things in a single location
- DRY is about having each piece of information and behavior in your system in a single, sensible place.

SRP vs. DRY

- SRP sounded a lot like DRY to me. Aren't both about a single class doing the one thing it's supposed to do?
  - They are related, and often appear together. DRY is about putting a piece of functionality in a single place, such as a class; SRP is about making sure that a class does only one thing, and that it does that one thing well. In well-designed applications, one class does one thing, and does it well, and no other classes share that behavior.

Sample Design Problem

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



Remember, each class should be responsible only for itself, and shouldn't rely on things going on inside other classes.

The Date class currently handles the job of seeing if a particular event is appropriate for a particular date.

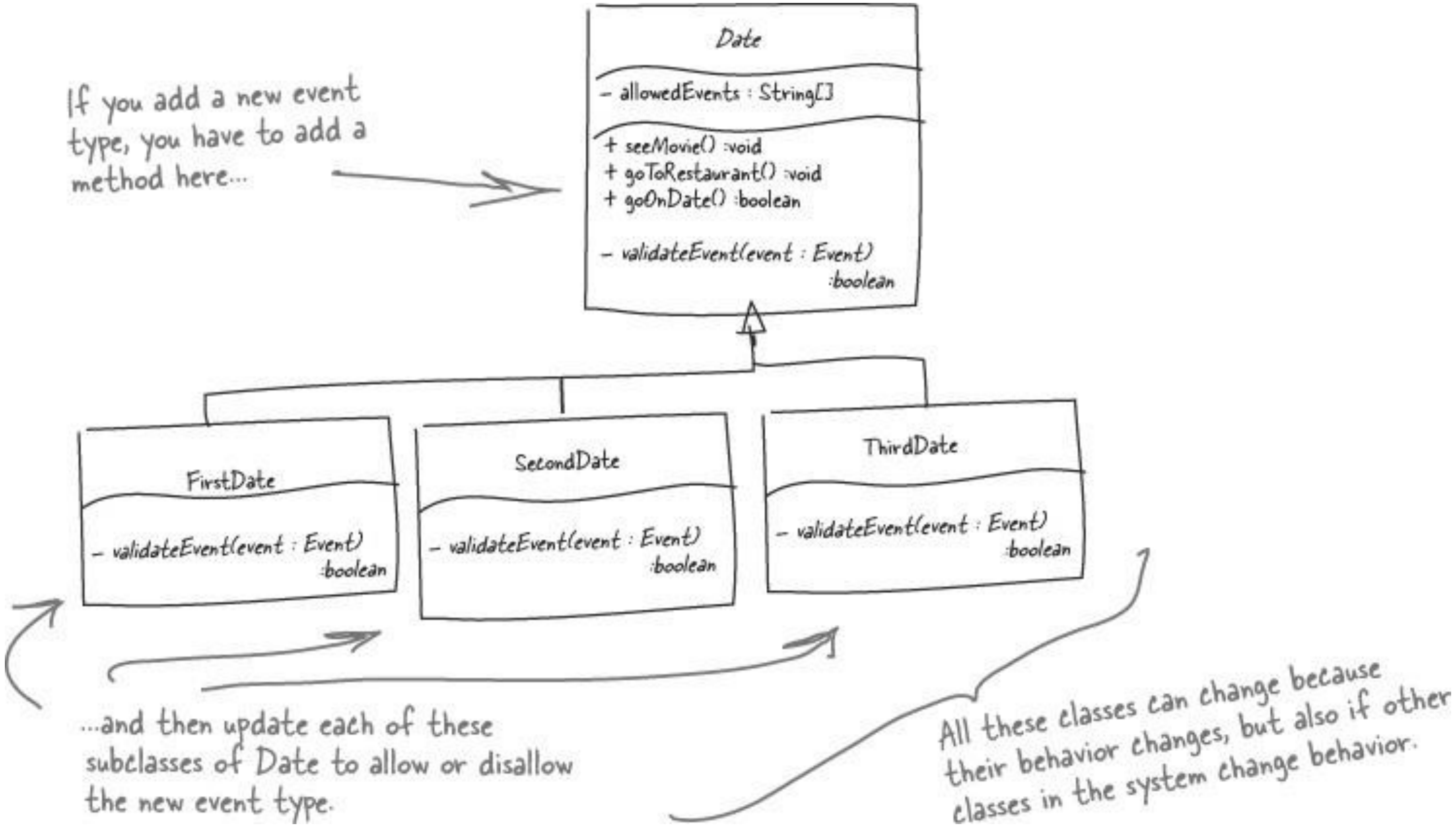
There's a method for each type of event that can be added to a date.

All of the logic in these methods needs to be updated every time you add a new type of event.

The different Date classes have to know what these name strings are to decide what events are allowed on a specific date, but if the name of the event changes, the Date subclasses have to change, too.

Sample Design Problem – SRP violation

- Date and Event break SRP.
  - All we should have to do is add the new event class and be done.

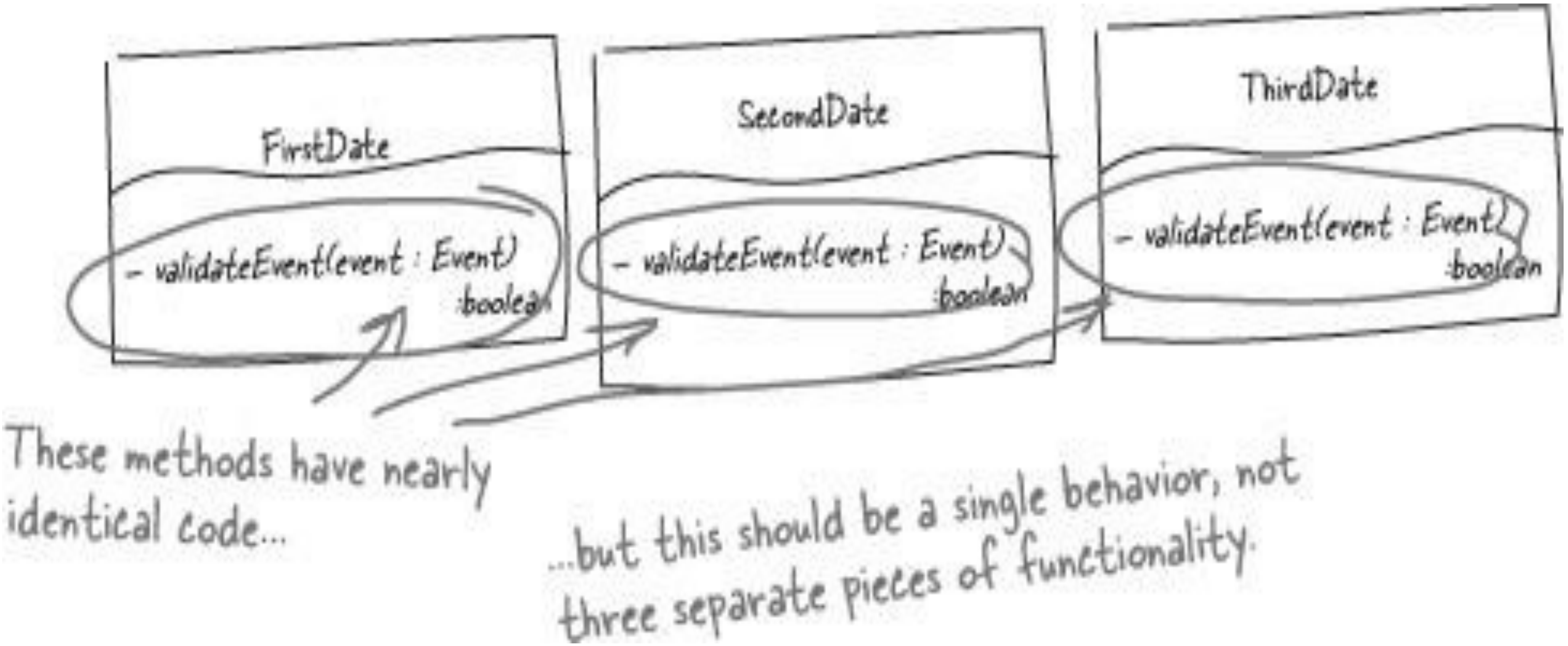

Date
- allowedEvents : String[]
+ seeMovie() :void
+ goToRestaurant() :void
+ goOnDate() :boolean
- validateEvent(event : Event) :boolean
FirstDate
- validateEvent(event : Event) :boolean
SecondDate
- validateEvent(event : Event) :boolean
ThirdDate
- validateEvent(event : Event) :boolean

If you add a new event type, you have to add a method here...

...and then update each of these subclasses of Date to allow or disallow the new event type.

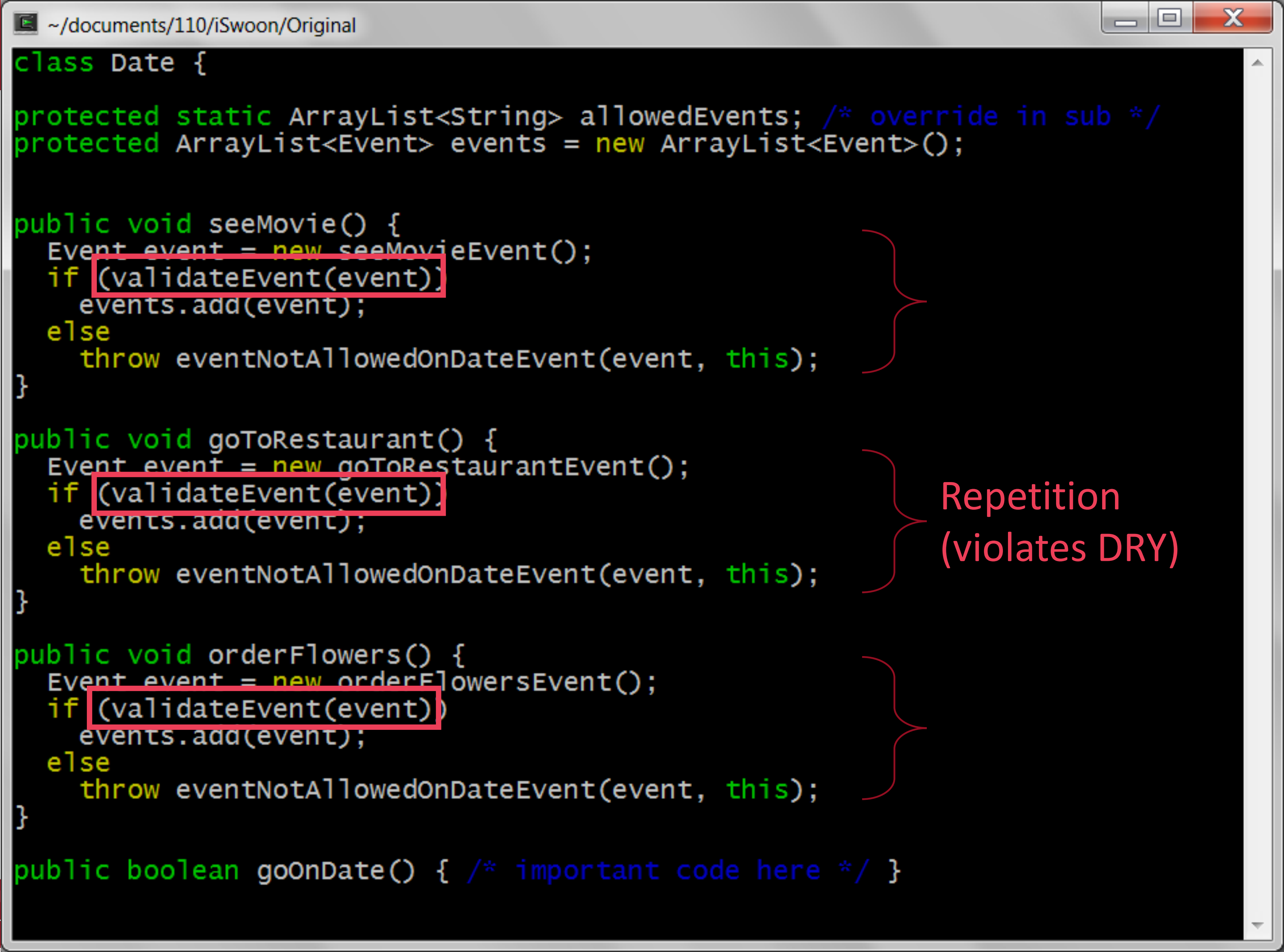
All these classes can change because their behavior changes, but also if other classes in the system change behavior.

SAMPLE DESIGN PROBLEM - DRY VIOLATION


FirstDate
- validateEvent(event : Event) :boolean
SecondDate
- validateEvent(event : Event) :boolean
ThirdDate
- validateEvent(event : Event) :boolean
These methods have nearly identical code...
...but this should be a single behavior, not three separate pieces of functionality.

EXAMPLE: ISWOON

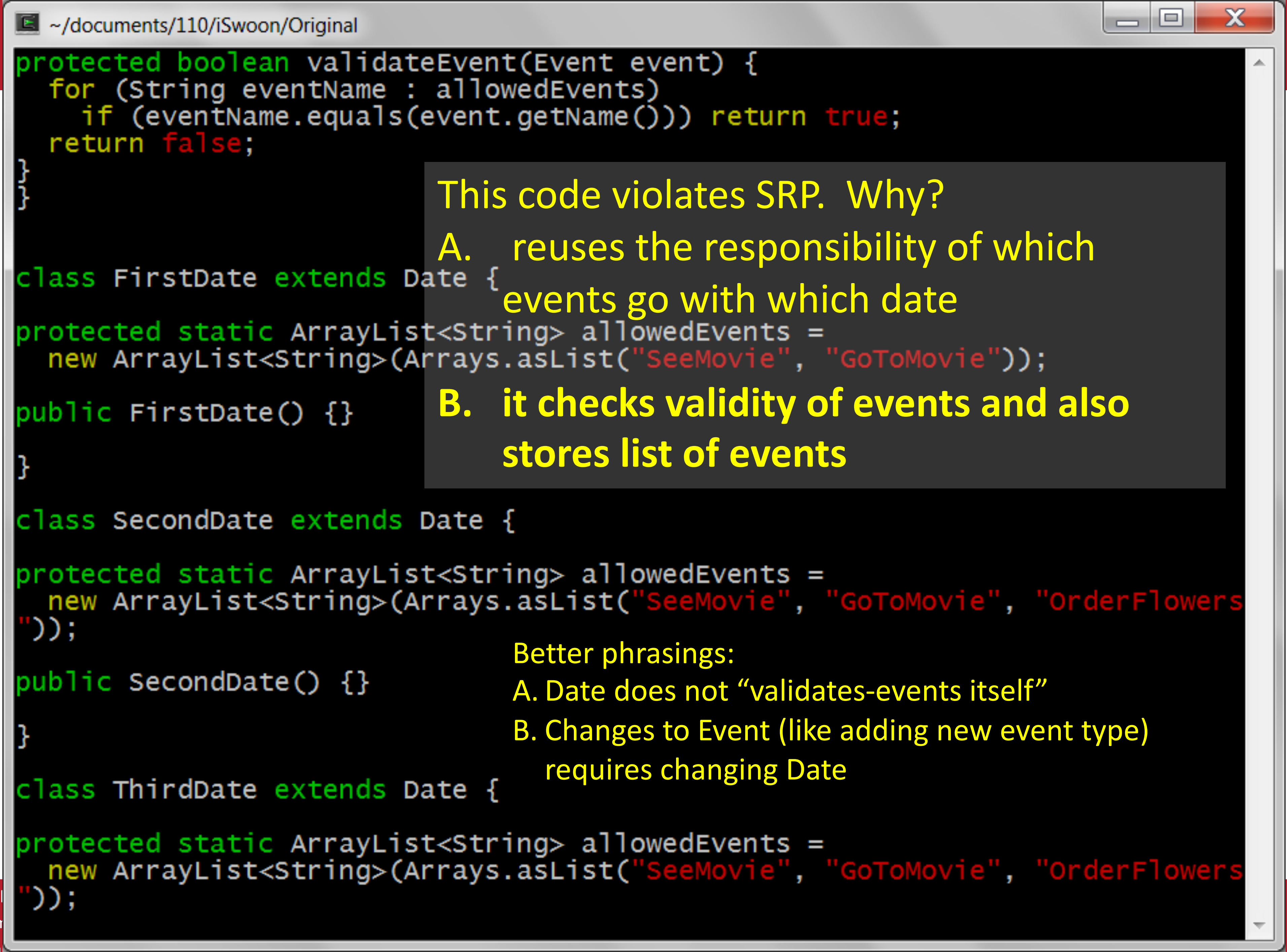
- Note that only difference is the class of the object being manufactured.
- **If one requires change, all do.** Easy to miss one or change one incorrectly.
- Validate is not only repetitive, **but also a violation of SRP, although it's hard to see here**.



```
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)

EXAMPLE: ISWOON (CONTINUED)



```
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?

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

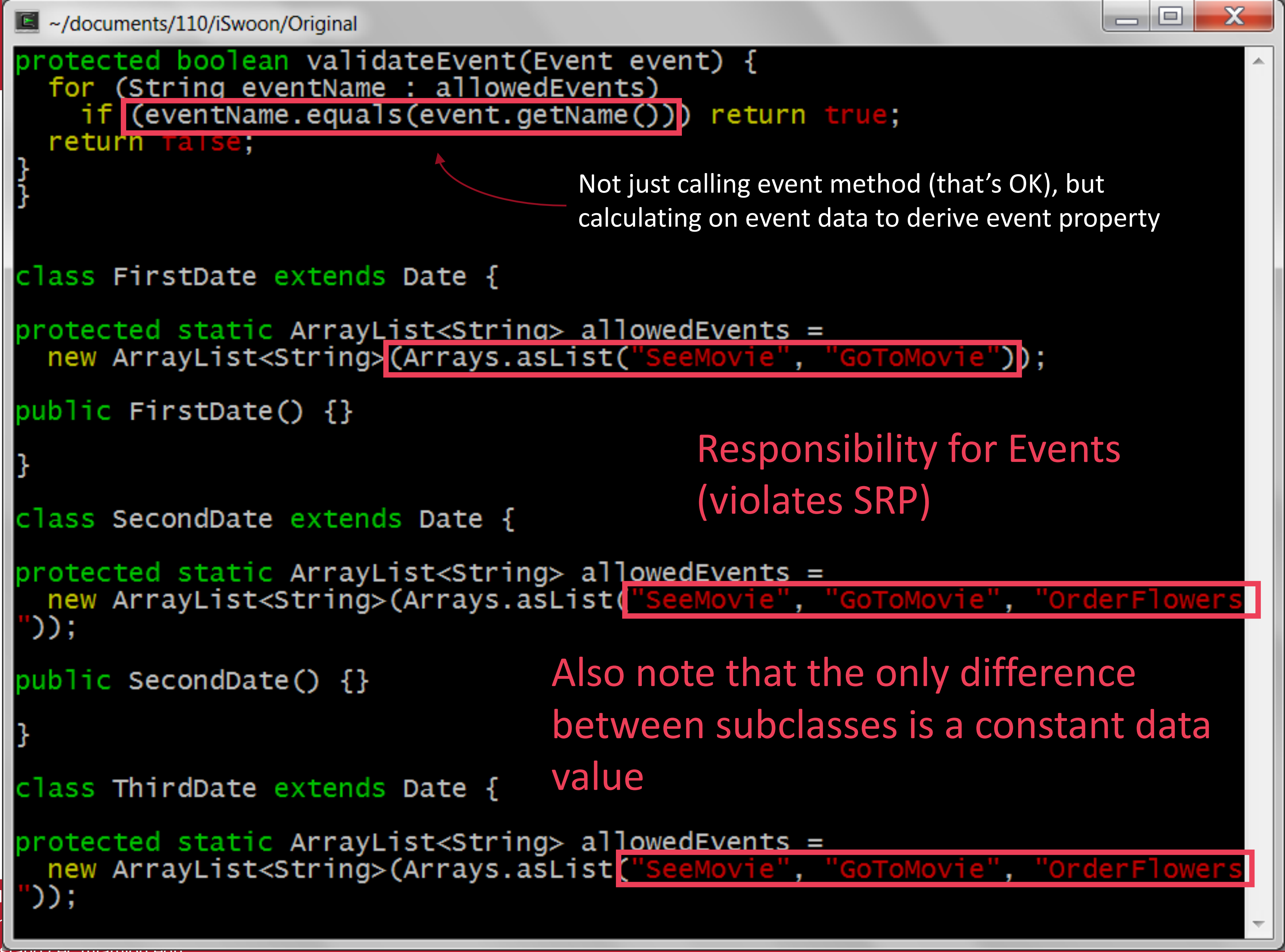
B. it checks validity of events and also stores list of events

Better phrasings:

A. Date does not “validates-events itself”

B. Changes to Event (like adding new event type) requires changing Date

EXAMPLE: ISWOON (CONTINUED)



```
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
"));
```

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

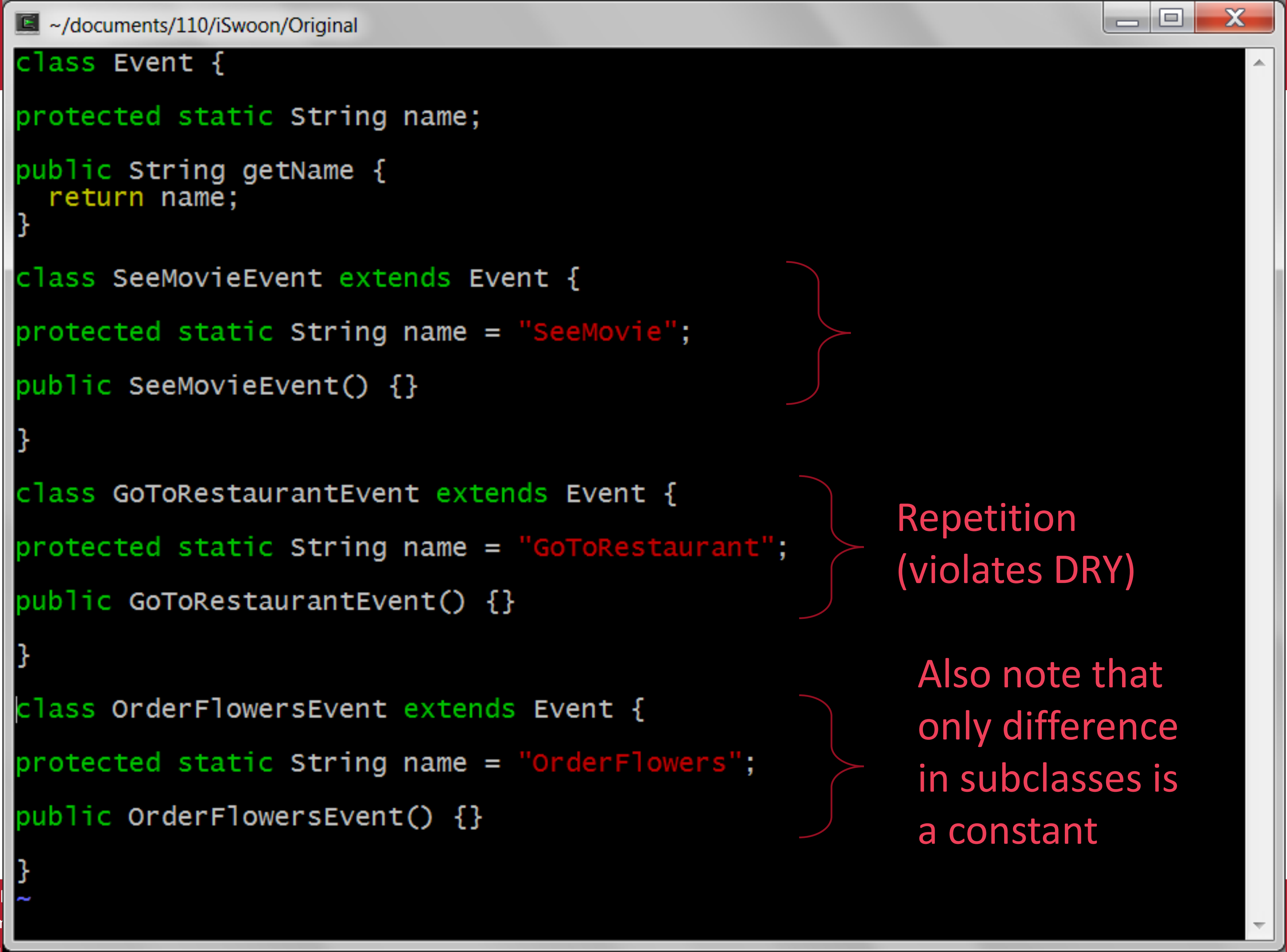
Responsibility for Events (violates SRP)

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

EXPLANATION

- Here we can see the **dual** responsibility.
  - We are comparing Event string names, not just calling Event methods.
  - This means that Date has to know what these strings mean, how to compare them (whole string vs. prefix), etc. KNOWS TOO MUCH ABOUT EVENTS.
  - Classes ideally interact purely through method calls. The fact that the comparison is outside the Event class is a red flag.
- Here we also again see repetitive, duplicated code.
  - Cut-and-paste with an edit of the array initializer.
- Having a class to represent variation in data is WRONG. This is what objects are for, not classes. Classes are for variation in computation (different methods). So we should collapse these into a single class to achieve DRY.

- More repetition



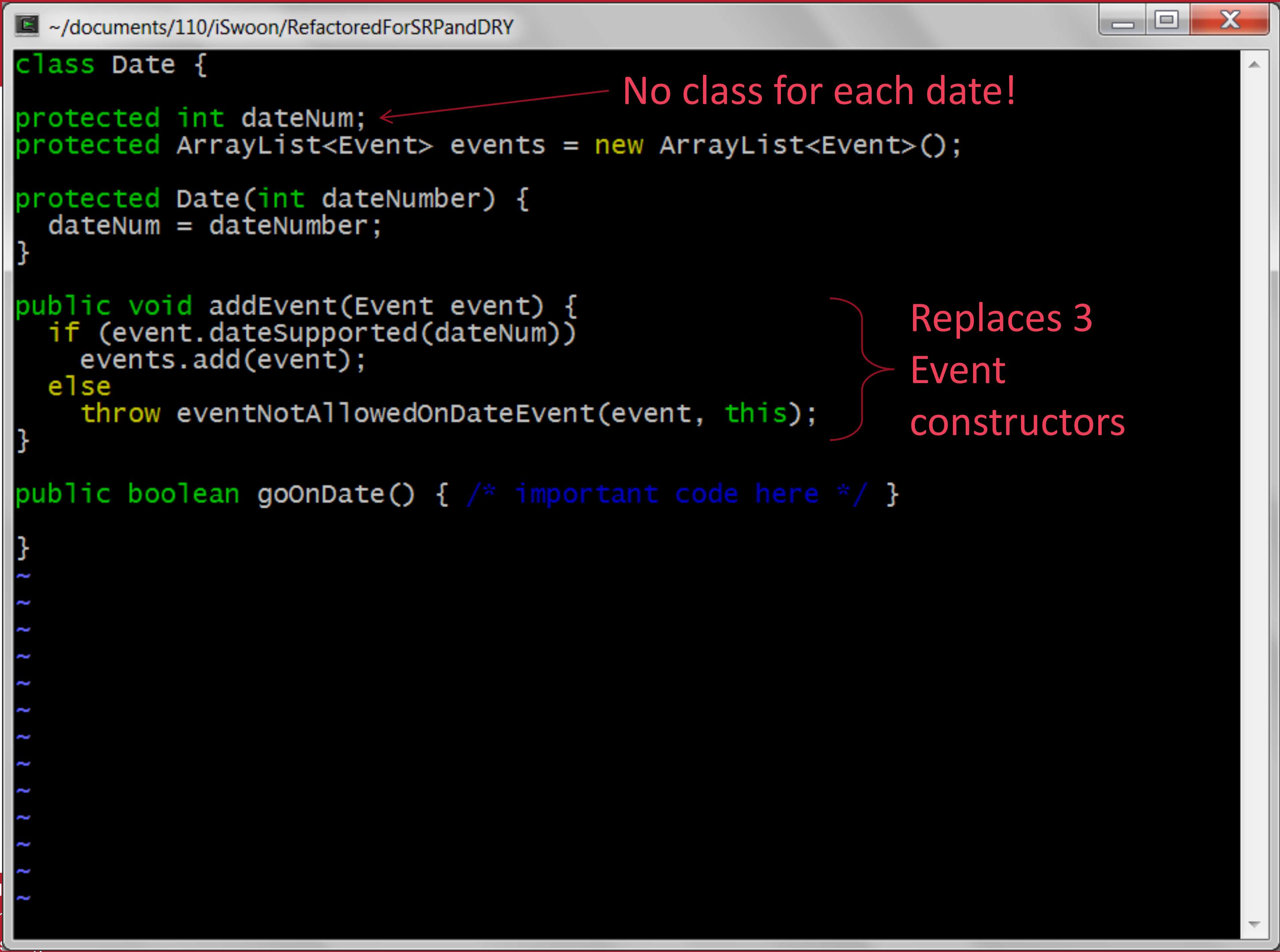
```
class Event {

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 iSwoon Design

- We realized that the number of dates could be quite large, and having a new class for each one is ridiculous.
- Note that addEvent replaces the three methods from before (seeMovie, goToRestaurant, orderFlowers).



~/documents/110/iSwoon/RefactoredForSRPandDRY

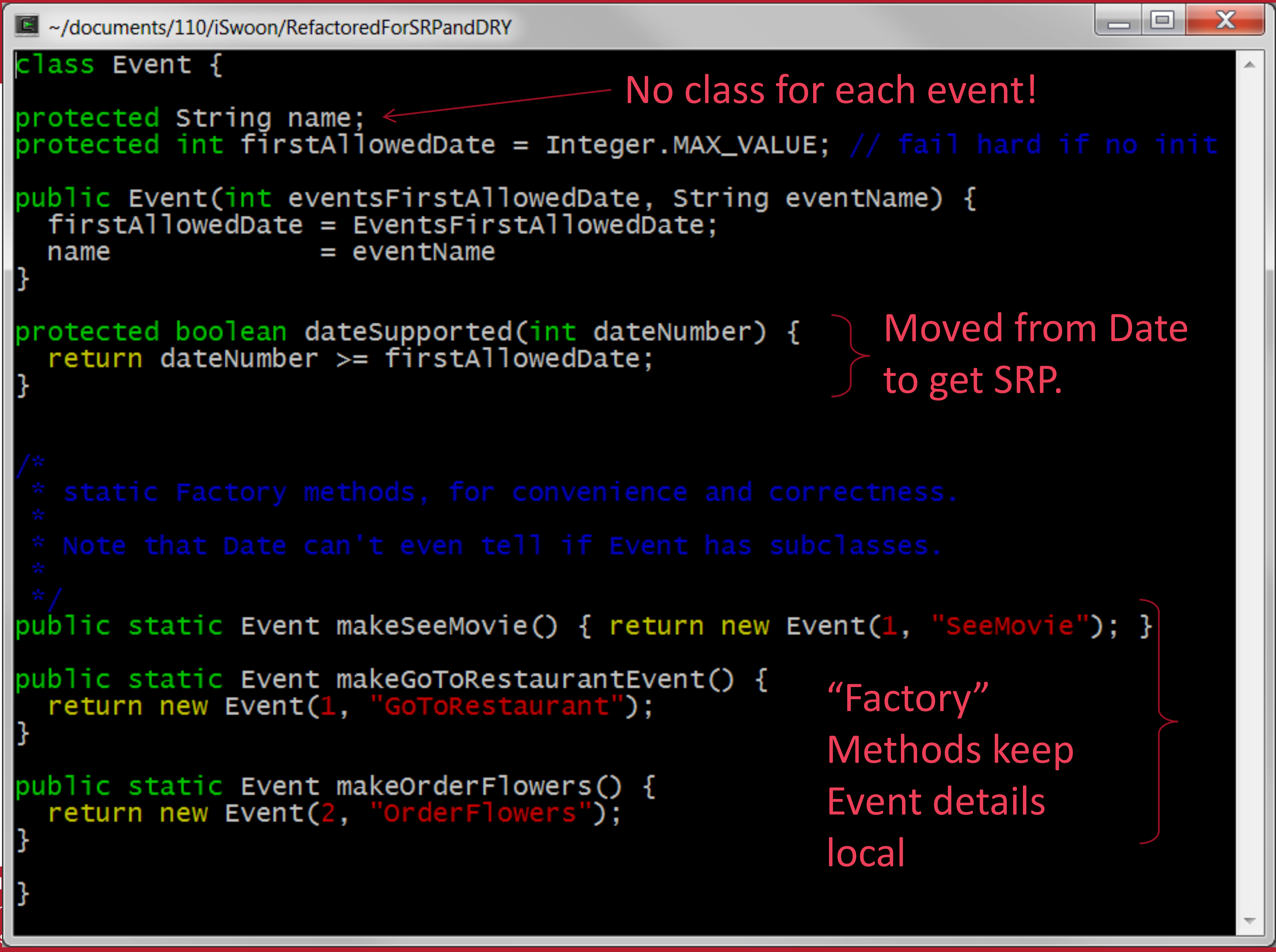
```
class Date {

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 */ }

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

No class for each date!

Replaces 3 Event constructors

Refactored iSwoon Design (cont'd)

- Again, we collapse to one class because we are now handling the data variation with objects, not classes.
- Note that dateSupported(int) is the old validate(Event) from the Date classes. No magic strings, and the whole comparison is handled inside the class.



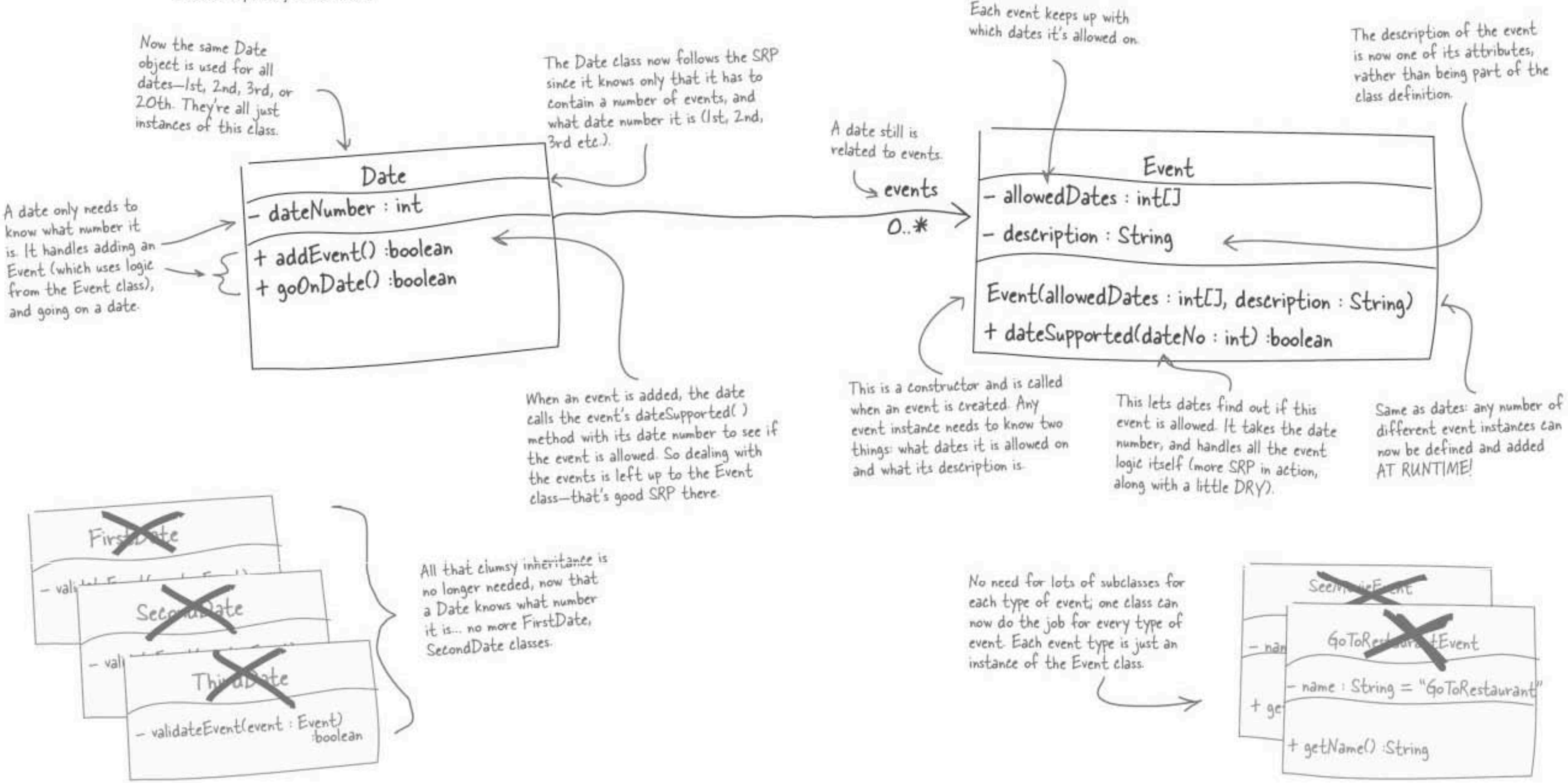
```
class Event {

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");
}

}
```

No class for each event!

Moved from Date to get SRP.

"Factory" Methods keep Event details local

You were asked to take a look at the current design and mark up what changes you'd make to apply the single responsibility principle to the iSwoon design to make it a breeze to update your software.



Now the same Date object is used for all dates—1st, 2nd, 3rd, or 20th. They're all just instances of this class.

The Date class now follows the SRP since it knows only that it has to contain a number of events, and what date number it is (1st, 2nd, 3rd etc.).

A date still is related to events.

Each event keeps up with which dates it's allowed on.

The description of the event is now one of its attributes, rather than being part of the class definition.

A date only needs to know what number it is. It handles adding an Event (which uses logic from the Event class), and going on a date.

When an event is added, the date calls the event's dateSupported( ) method with its date number to see if the event is allowed. So dealing with the events is left up to the Event class—that's good SRP there.

This is a constructor and is called when an event is created. Any event instance needs to know two things: what dates it is allowed on and what its description is.

This lets dates find out if this event is allowed. It takes the date number, and handles all the event logic itself (more SRP in action, along with a little DRY).

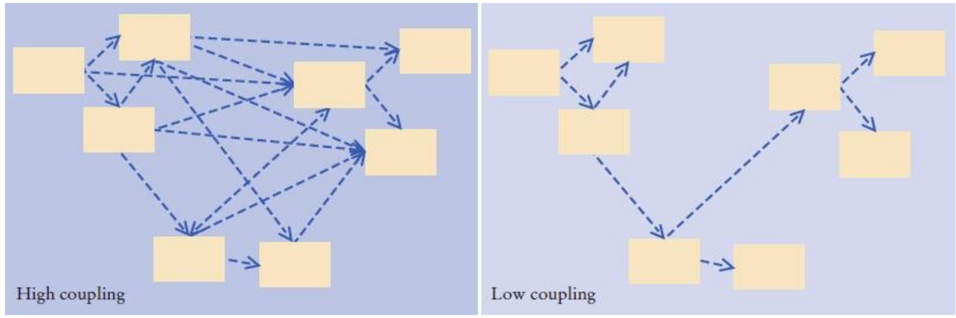
Same as dates: any number of different event instances can now be defined and added AT RUNTIME!

All that clumsy inheritance is no longer needed, now that a Date knows what number it is... no more FirstDate, SecondDate classes.

No need for lots of subclasses for each type of event; one class can now do the job for every type of event. Each event type is just an instance of the Event class.

DEPENDENCY & COUPLING

- If many classes depend on each other = High Coupling
- Few dependencies = Low Coupling
- Why does it matter?
  - Drastic change implies updates
  - If we would like to use a class in another program, we'd have to take with it all the classes it depends on.



Perfect Design vs. Good-Enough Design

Perfect Design

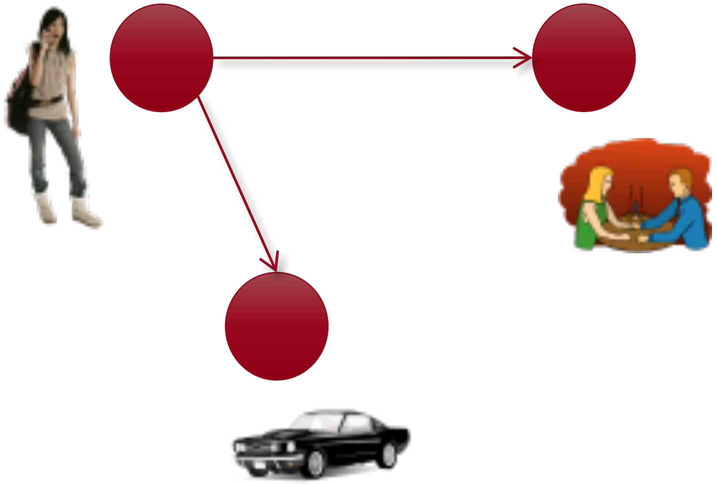
- (High or Low) Cohesion?
- (High or Low) Coupling?

Good-Enough Design

- Bad design will make you late
- Perfect design will make you late
- So make your design good enough

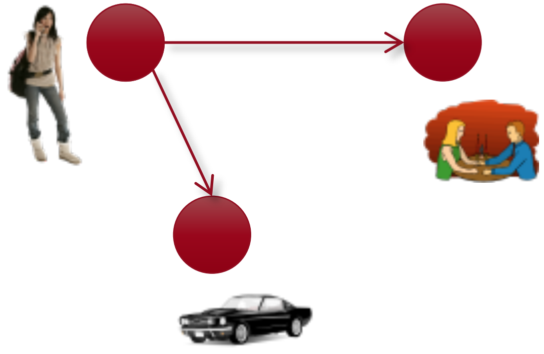
A Concise Theory of Object-Oriented

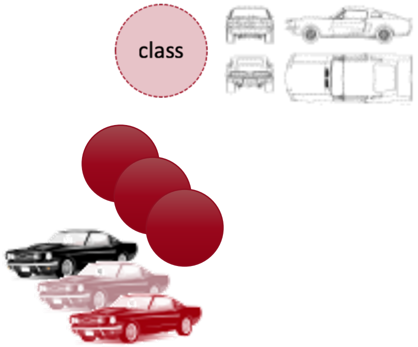
- Object represents a “thing”
  - *person, car, date*, …
  - (not two things, not ½ thing)
- Object responds to *messages*
  - (method calls)
  - *Things it does to itself* **(SRP)**
  - That is, other objects ask the object to do something to itself
- Objects are “opaque”
  - Can’t see each others’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



A Concise Theory of Object-Oriented

- Because objects are completely opaque, we don't need to know what's really inside them
  - Each *car* object <u>could</u> be implemented with its own unique code
- 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**





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:** a change in one class causes change in another class
  - **DRY**: repetitive code. A “small” change requires many similar changes across methods or classes
- 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

RETROSPECTIVE QUESTIONS

What are design patterns? Example?

What is the SRP?

What is the DRY principle?

What do we strive for in a good/perfect design?