Software Design: Design Patterns & Good Design

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

AGENDA

- Review
- Software Design
  - Design Patterns
  - Design Consideration
  - Cohesion and Coupling

AGENDA

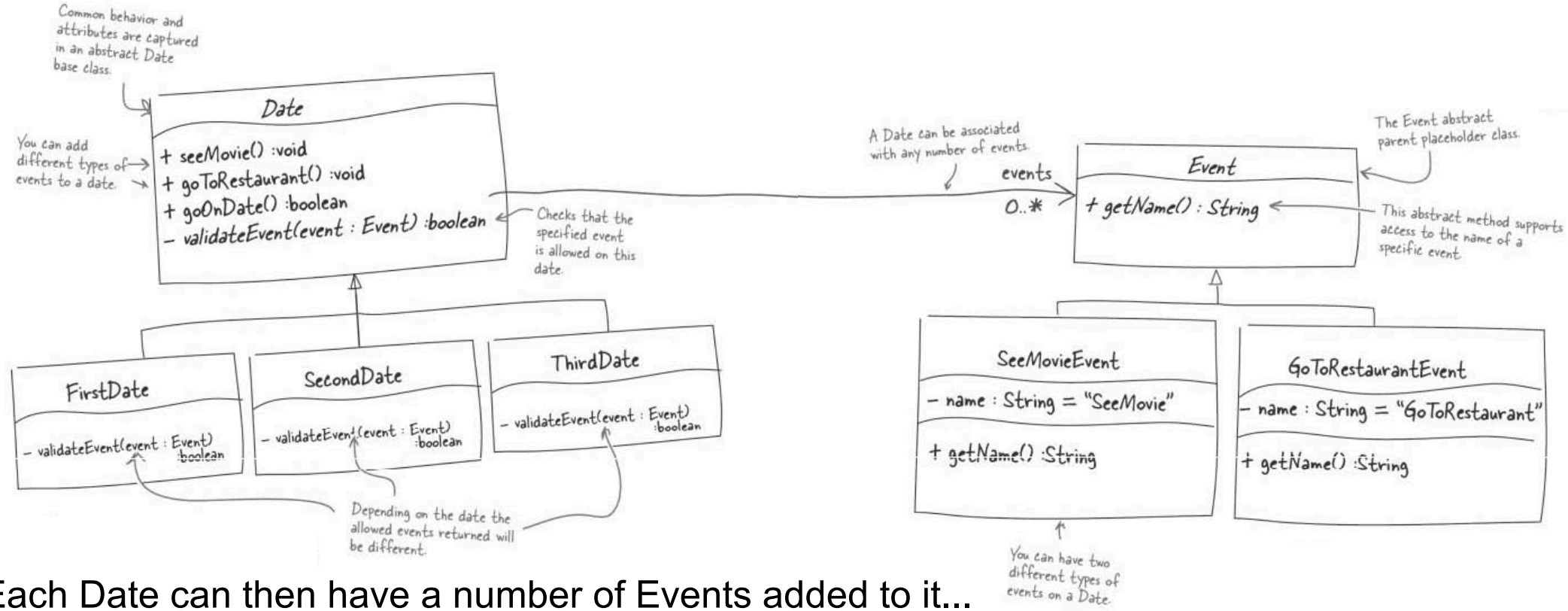
- Review
- Software Design
  - Design Patterns
  - Design Consideration
  - Cohesion and Coupling

Sequence Diagrams

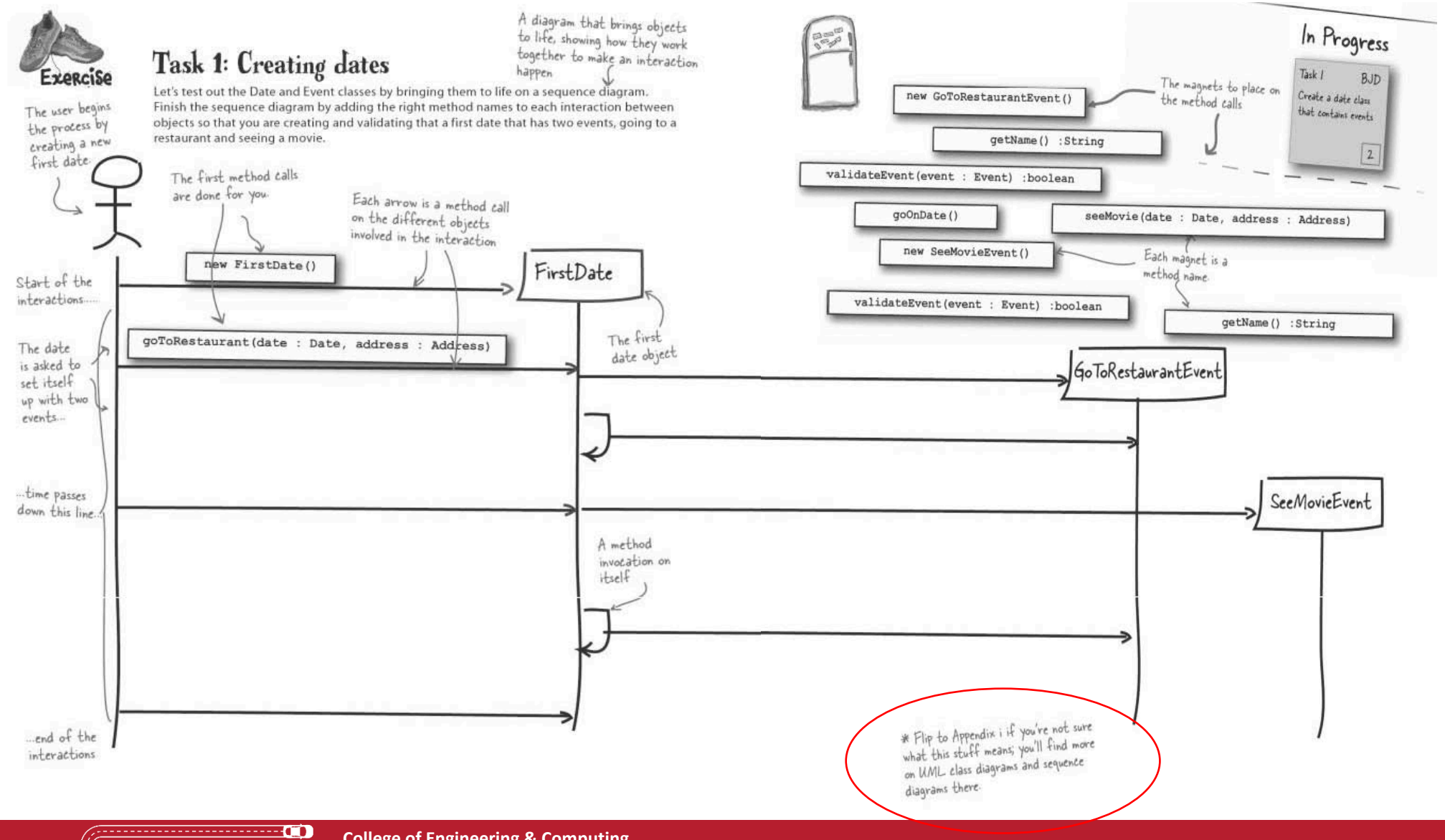
- A diagram that brings objects to life
  - Showing how they work together to make an interaction happen.

CREATE THE DATE CLASS

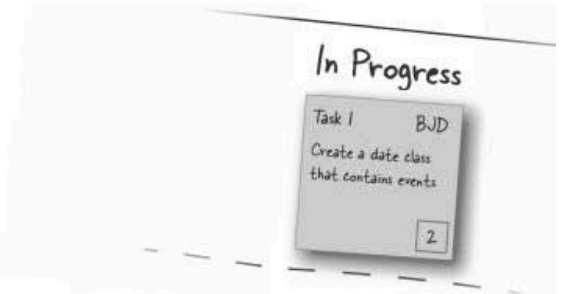
- The Date class is split into three classes, one class for each type of date...

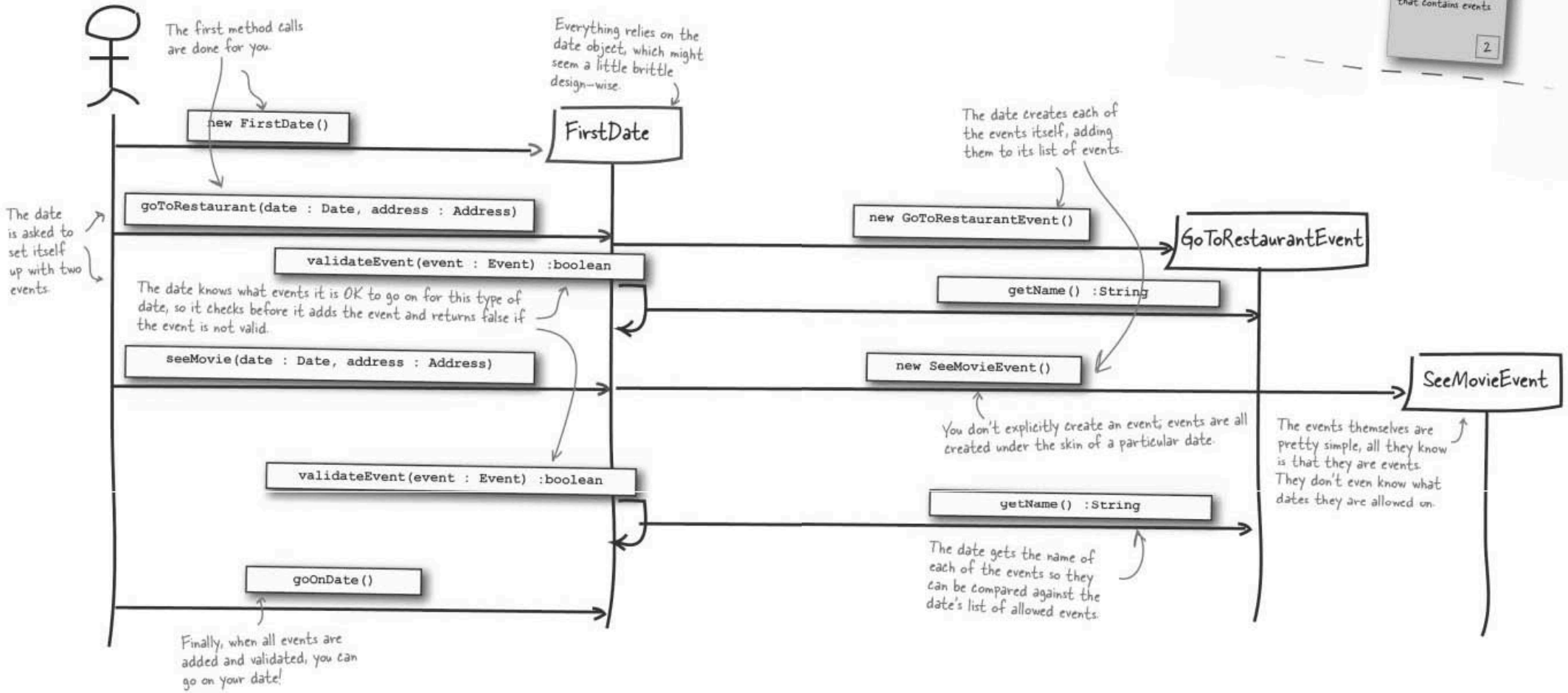


- Each Date can then have a number of Events added to it...


Exercise
Task 1: Creating dates
A diagram that brings objects to life, showing how they work together to make an interaction happen
Let's test out the Date and Event classes by bringing them to life on a sequence diagram. Finish the sequence diagram by adding the right method names to each interaction between objects so that you are creating and validating that a first date that has two events, going to a restaurant and seeing a movie.
The user begins the process by creating a new first date.
The first method calls are done for you.
Each arrow is a method call on the different objects involved in the interaction
Start of the interactions.....
new FirstDate()
FirstDate
The first date object
The date is asked to set itself up with two events...
goToRestaurant(date : Date, address : Address)
GoToRestaurantEvent
...time passes down this line...
SeeMovieEvent
A method invocation on itself
...end of the interactions
* Flip to Appendix i if you're not sure what this stuff means; you'll find more on UML class diagrams and sequence diagrams there.
new GoToRestaurantEvent()
The magnets to place on the method calls
getName() :String
validateEvent(event : Event) :boolean
goOnDate()
seeMovie(date : Date, address : Address)
new SeeMovieEvent()
Each magnet is a method name.
validateEvent(event : Event) :boolean
getName() :String
In Progress
Task 1 BJD
Create a date class that contains events
2

Your job was to test out the Date and Event classes by bringing them to life on a sequence diagram. You should have finished the sequence diagram so that you plan and go on a first date with two events, going to a restaurant and seeing a movie.





AGENDA

- Review
- Software Design
  - Design Patterns
  - Design Consideration
  - Cohesion and Coupling

Software Design is a Craft

- Software design is a craft; the more you do it, the better you get
  - Observe other software artifacts
  - Try to mimic how others break down problems and formulate solutions

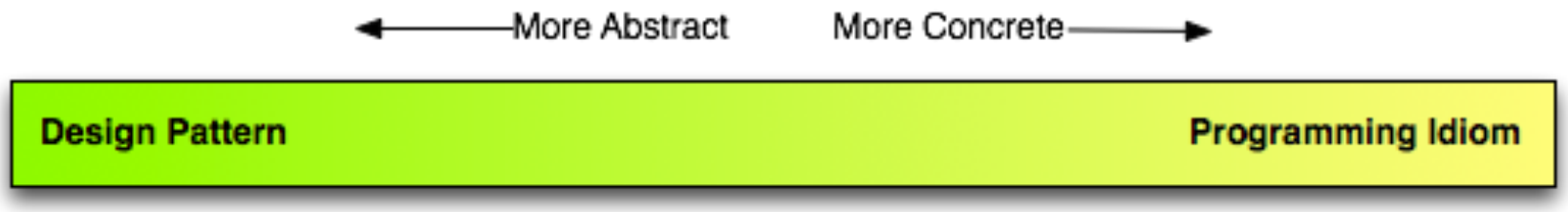
DON'T REINVENT THE SOLUTION

- As with any craft, programming contains an undeniable element of experience.
- We achieve mastery through long practice in solving the problems that inevitably arise in trying to apply technology to actual problem situations.

Common Programming Solutions

- Programming idioms (?)
- Design patterns

Difference Between Idiom and Pattern



* the distinction is sometimes blurry

Programming Idioms

- Often programming language-specific (like spoken language idioms)

Idiom	Meaning
i++	i=i+1
"holy cow"	"wow"

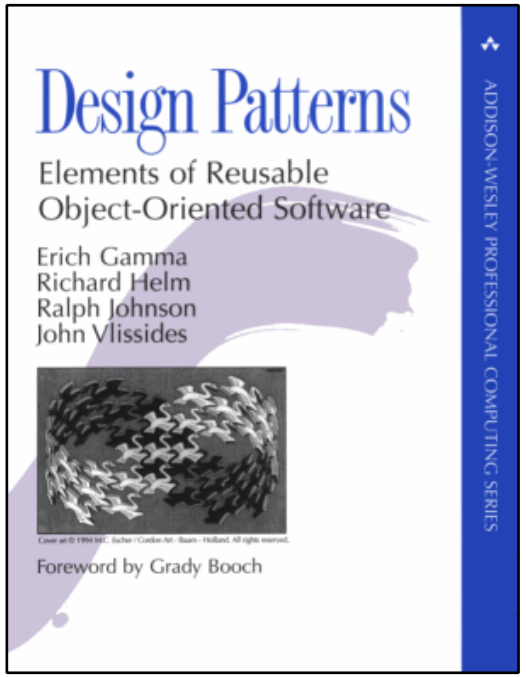
```
int i;
int a[10];

//Fortran programmer writing C
for (i=1; i<=10; i=i+1)
    a[i] = 0;

//C programmer
for (i=0; i<10; i++)
    a[i] = 0;
```

Design Patterns

- ***Design Pattern*** : "A description of the problem and the essence of its solution to enable the solution to be reused in different settings" [GLOSSARY]

- Created/inspired by architect Christopher Alexander in 1977
- Common problems have common optimal solutions
  - Solutions are heavily influenced by practical experience
  - Refined over time

GoF – Gang of Four



Design Patterns: Elements of Reusable Object-Oriented Software [GOF]
By: Erich Gamma, Richard Helm, RalphJohnson, and John Vlissides (the **Gang Of Four**)

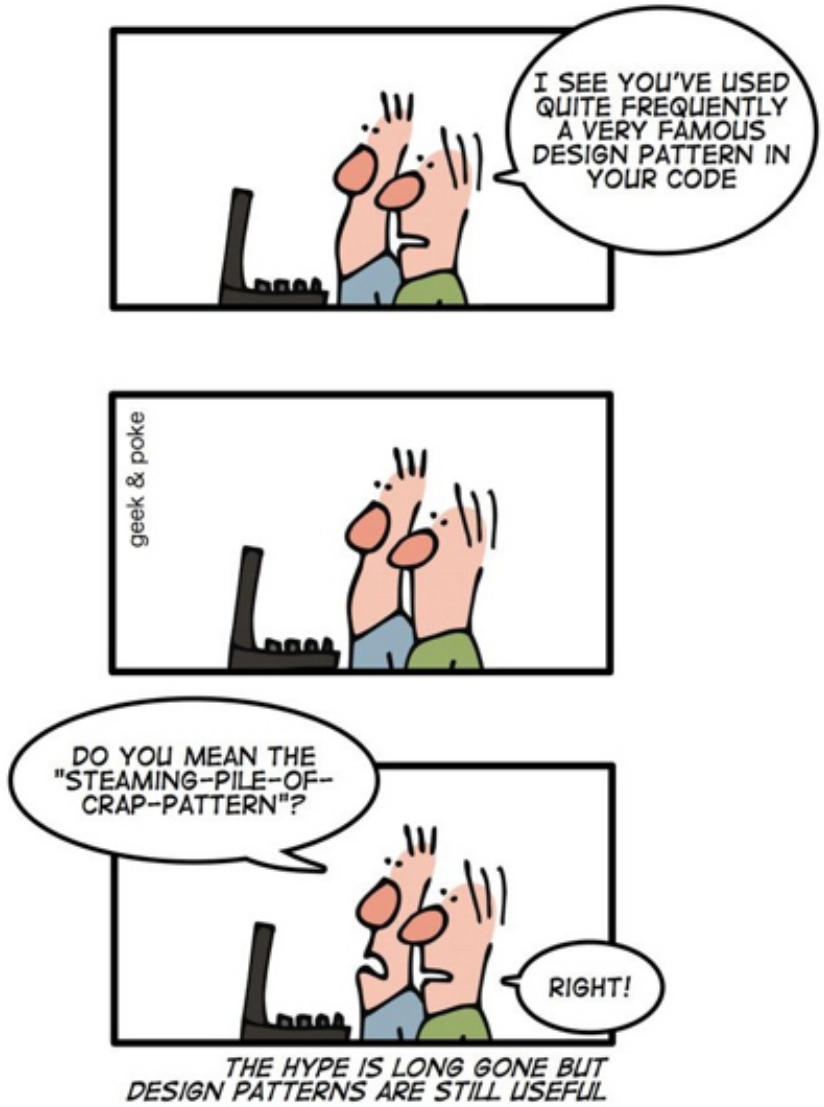
Many patterns described on the following wiki:

http://c2.com/cgi/wiki?GangOfFour

- GoF describes 23 design patterns
  - They aren't the only design patterns
  - Some of them aren't often used

Popular Patterns

- Facade
- Adapter
- Observer
- Builder
- Singleton
- Iterator...



ITERATOR PATTERN

- Iterator Pattern: "Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation." [GOF]

JAVA ITERATOR EXAMPLE

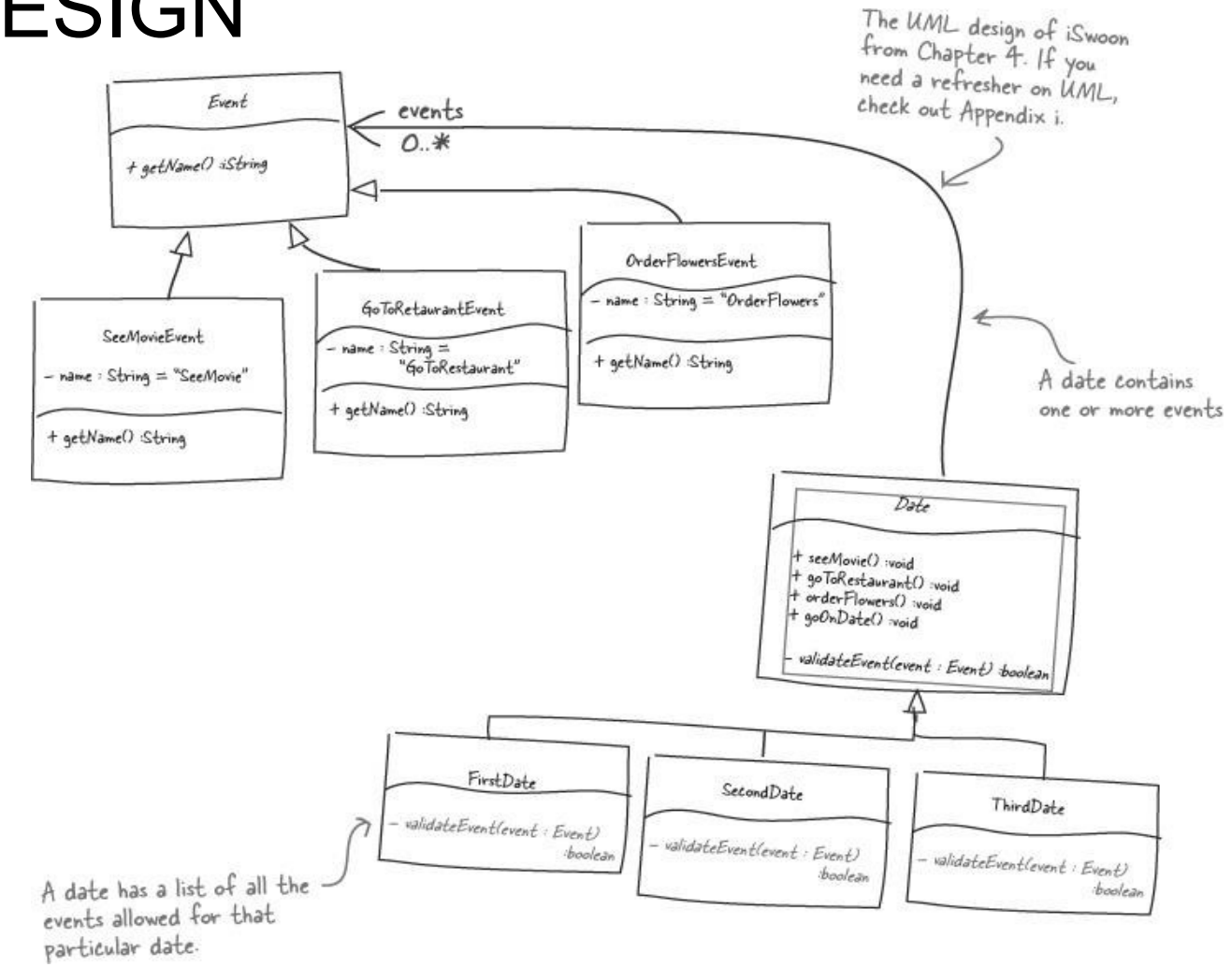
- Java has the Iterator Pattern built into the language
- The **aggregate** object is a collection of items:
  - `LinkedList<Integer> foo;`
- The aggregate produces an **iterator** object:
  - `i = foo.listIterator();`
- A java Iterator is an interface that can be implemented by any aggregate and has the following methods:
  - hasNext();
  - next();
  - remove();

```java
import java.util.LinkedList;
import java.util.ListIterator;

public class ExampleIterator {
    public static void main(String[] args) {

        LinkedList<Integer> foo = new LinkedList<Integer>();
        Integer tmp;

        foo.add(1);
        foo.add(2);

        ListIterator<Integer> i = foo.listIterator();
        while (i.hasNext()) {
            tmp = i.next();
            System.out.println(tmp);
        }
    }
}
```

DON'T REINVENT THE PROBLEM TO MATCH SOLUTION

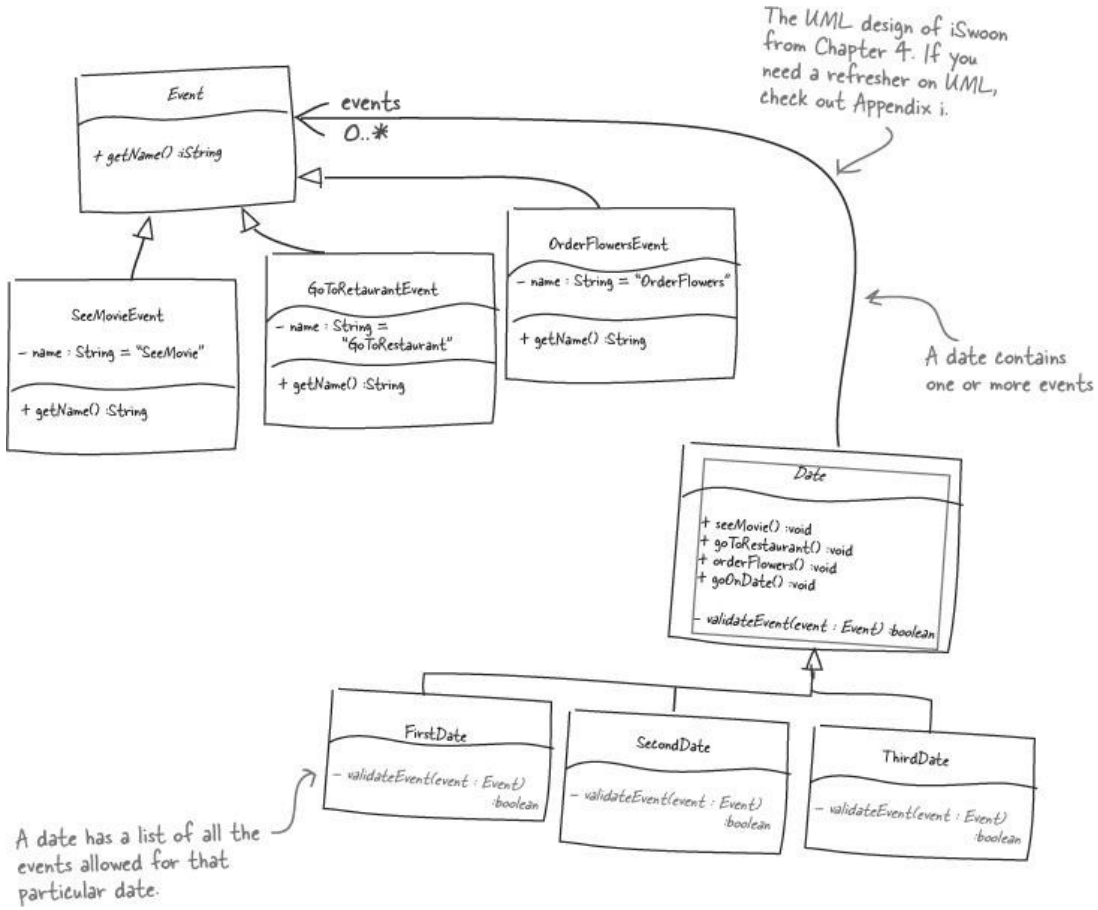
GOOD-ENOUGH DESIGN

- What's wrong with this design?



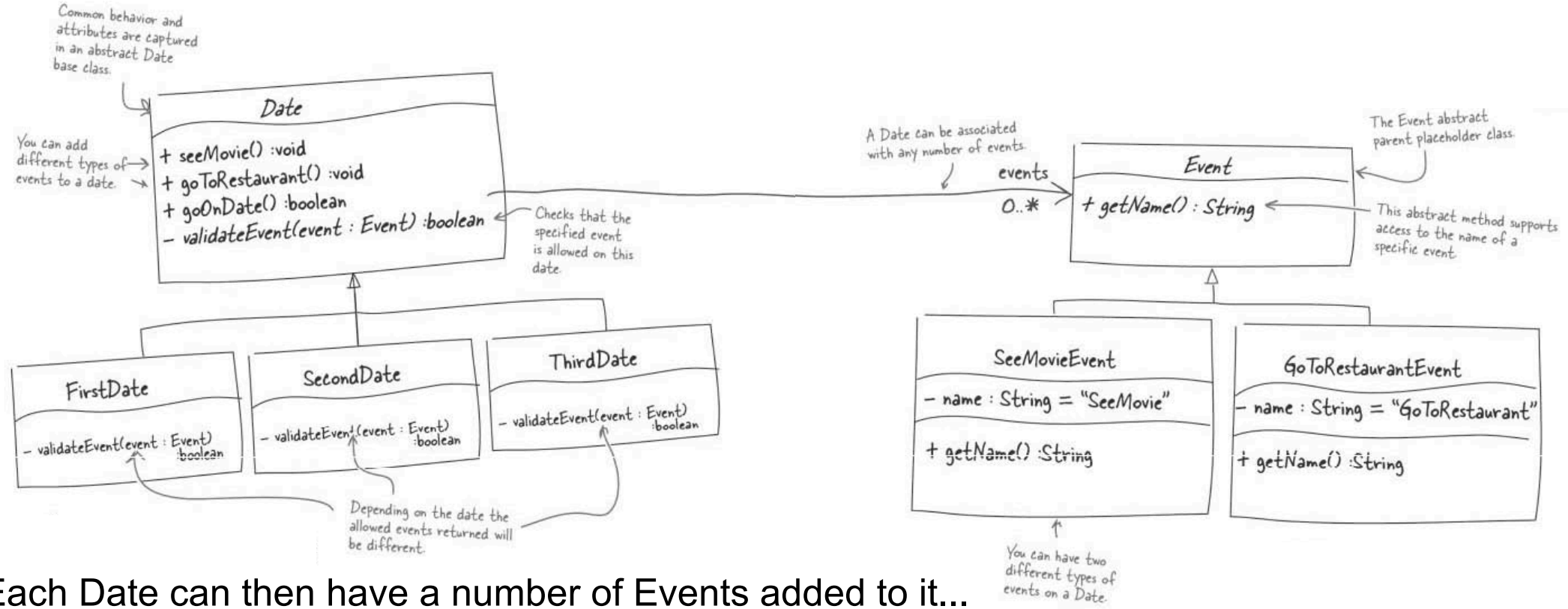
PROBLEMS WITH THAT DESIGN?

- To add three new event types?
  - We need a new event class for each of the new types.
  - We would need to add three new methods, one for each type of event to the abstract parent Date class.
  - Then, each of the date classes, will need to be updated to allow (or disallow) the three new types of event, depending on if the event is allowed for that date.
- E.G., adding a new event type called "Sleeping over" but that event was only allowed on the third date?
  - A new event class would be added, called something like "SleepingOverEvent."
  - Then a new method called "sleepOver" needs to be added to the Date class so the new event can be added to a date.
  - Finally, all three of the existing date classes would need to be updated in order to specify that only the third date allows a SleepingOverEvent to be specified.



TASK1: CREATE THE DATE CLASS

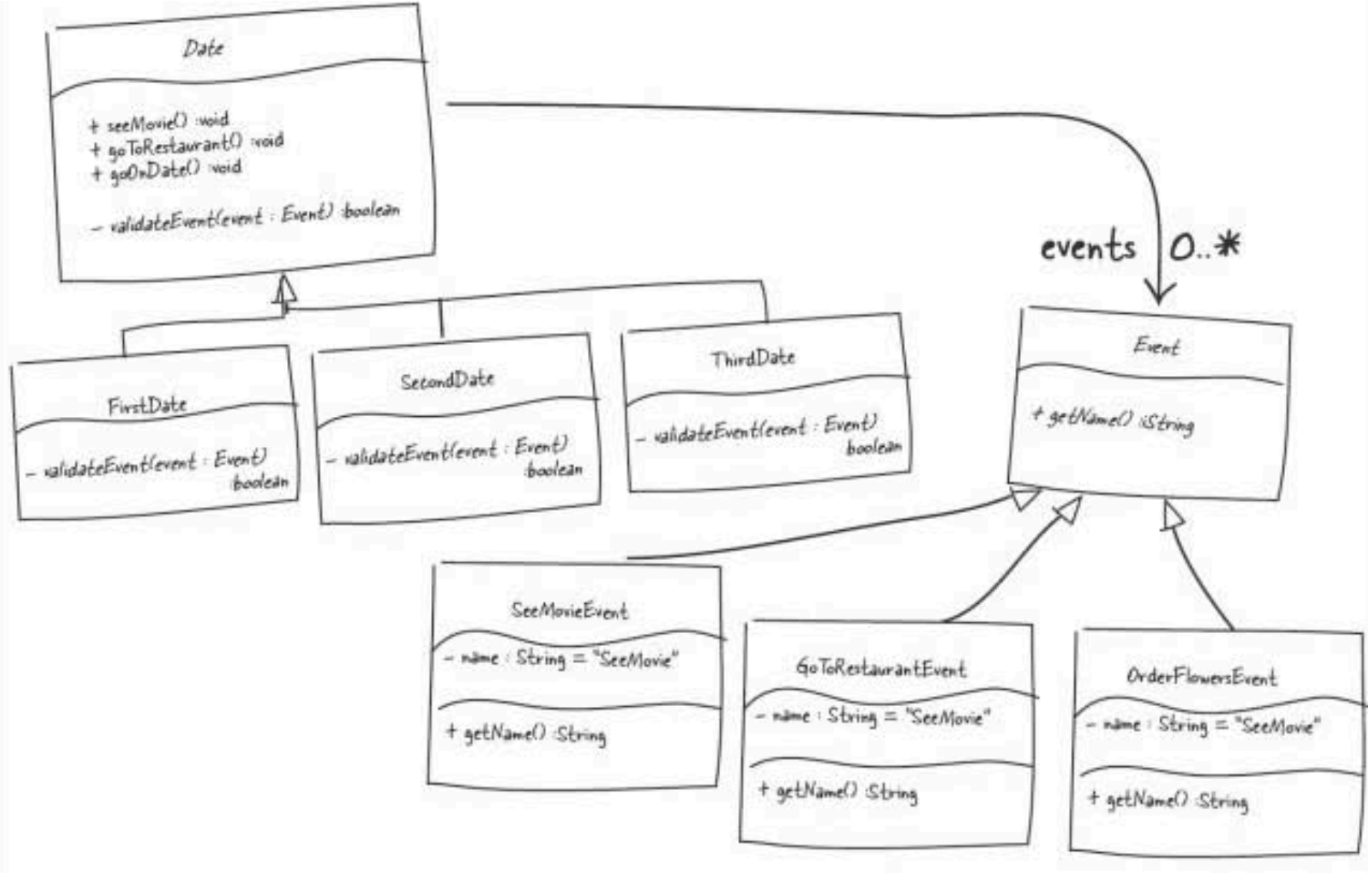
- The Date class is split into three classes, one class for each type of date...

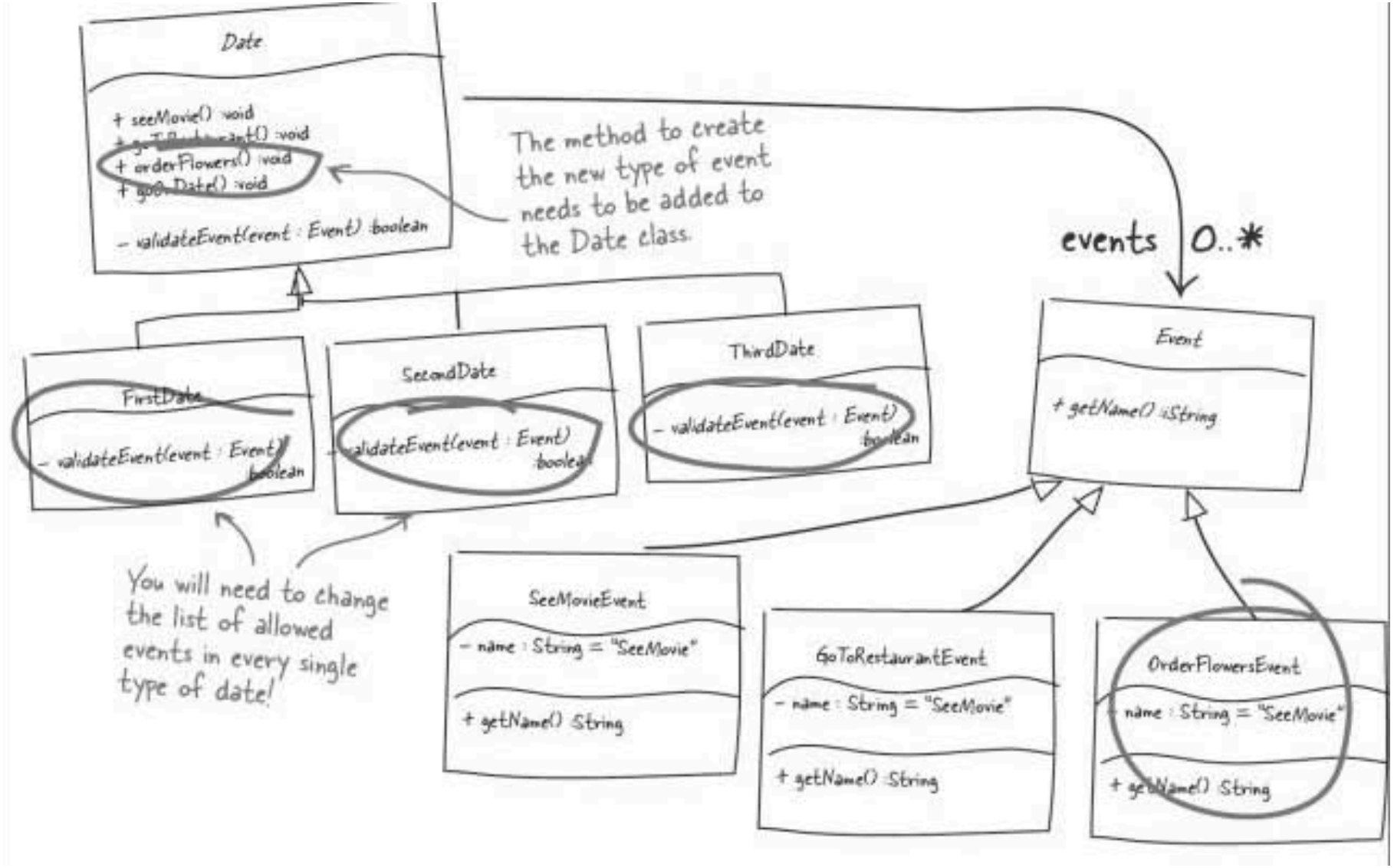


- Each Date can then have a number of Events added to it...

CHANGES SCENARIO

- **Bob**: well, if you treat someone ordering flowers as just another type of event, then we can add it straight into our current class tree, and that should save us some time in the long run.
  - What refactoring do you think Bob is talking about?
  - Take the class hierarchy below and circle all the things that you think will need to change to accommodate a new **OrderFlowers** event.
  - How many classes did you have to touch to make Bob's changes?
  - Are you happy with this design? Why or why not?


Date
+ seeMovie() :void
+ goToRestaurant() :void
+ goOnDate() :void
- validateEvent(event : Event) :boolean
events
0..*
Event
+ getName() :String
FirstDate
- validateEvent(event : Event) :boolean
SecondDate
- validateEvent(event : Event) :boolean
ThirdDate
- validateEvent(event : Event) :boolean
SeeMovieEvent
- name : String = "SeeMovie"
+ getName() :String
GoToRestaurantEvent
- name : String = "SeeMovie"
+ getName() :String
OrderFlowersEvent
- name : String = "SeeMovie"
+ getName() :String


Date
+ seeMovie() :void
+ goToRestaurant() :void
+ orderFlowers() :void
+ goOnDate() :void
- validateEvent(event : Event) :boolean
The method to create the new type of event needs to be added to the Date class.
events 0..*
Event
+ getName() :String
FirstDate
- validateEvent(event : Event) :boolean
SecondDate
- validateEvent(event : Event) :boolean
ThirdDate
- validateEvent(event : Event) :boolean
You will need to change the list of allowed events in every single type of date!
SeeMovieEvent
- name : String = "SeeMovie"
+ getName() :String
GoToRestaurantEvent
- name : String = "SeeMovie"
+ getName() :String
OrderFlowersEvent
- name : String = "SeeMovie"
+ getName() :String

EXPLANATION

- Five classes were changed or added to add just this one new type of event.
  - First the "**OrderFlowersEvent**" class needed to be added,
  - And then the method to order flowers on a date needed to be added to the Date class.
  - Finally I had to update each of the different types of date to allow, or reject, the new type of event depending on whether it's allowed on that date or not
- Five classes being changed seems like a LOT when all I'm adding is ONE new event.
- What happens when I have to add, say, a dozen new types of event; is it always going to involve this much work?

SINGLE RESPONSIBILITY PRINCIPLE

- 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.

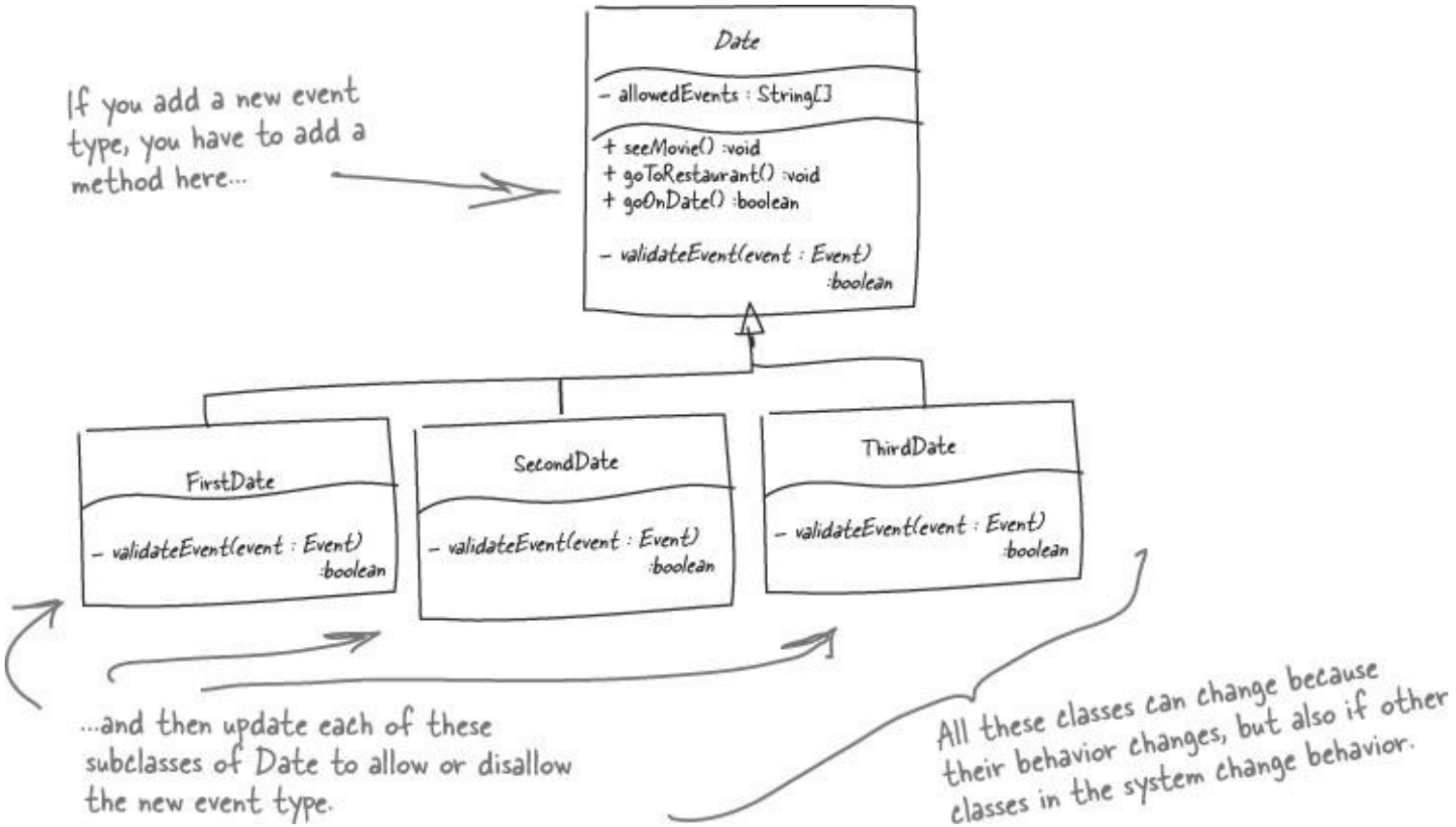
SINGLE RESPONSIBILITY PRINCIPLE

- 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.

SAMPLE DESIGN PROBLEM

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



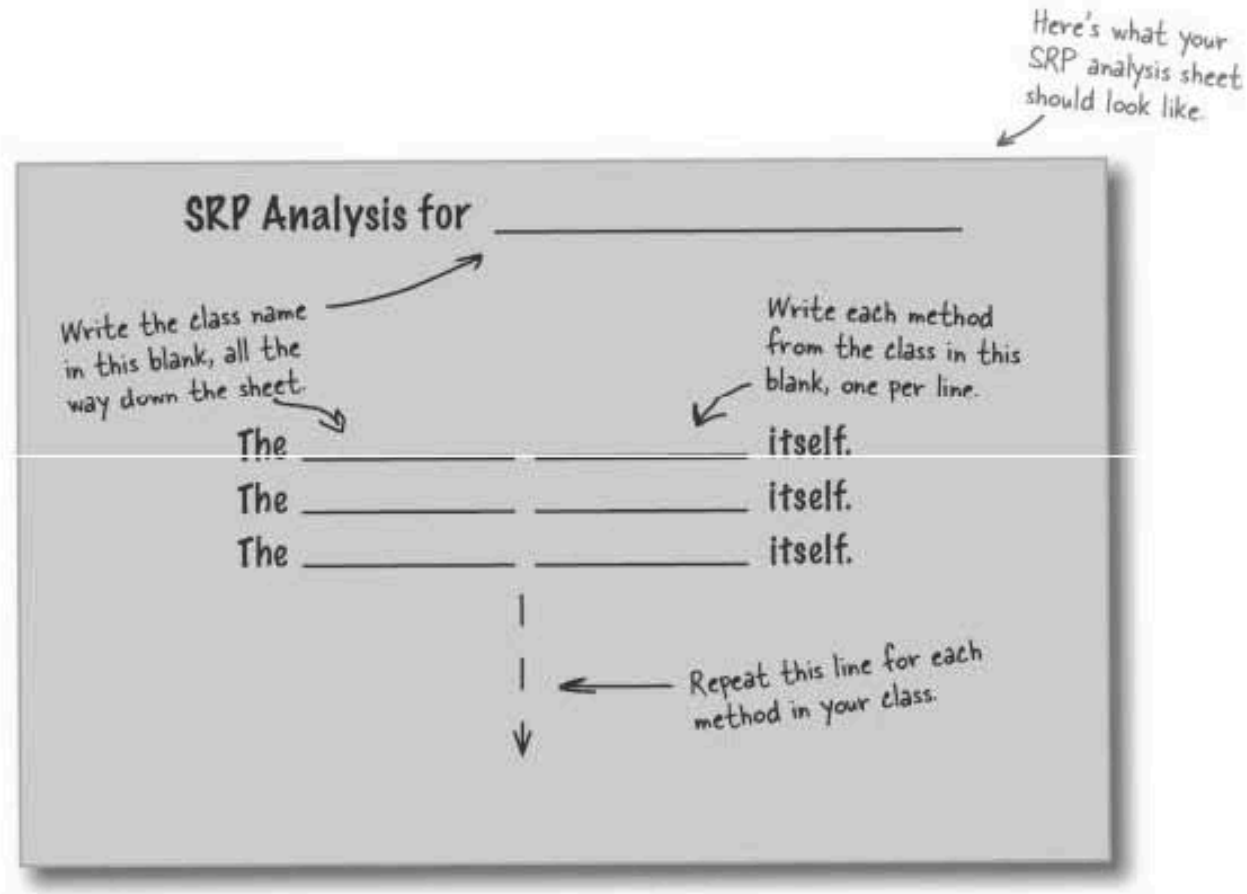
AKA COHESION

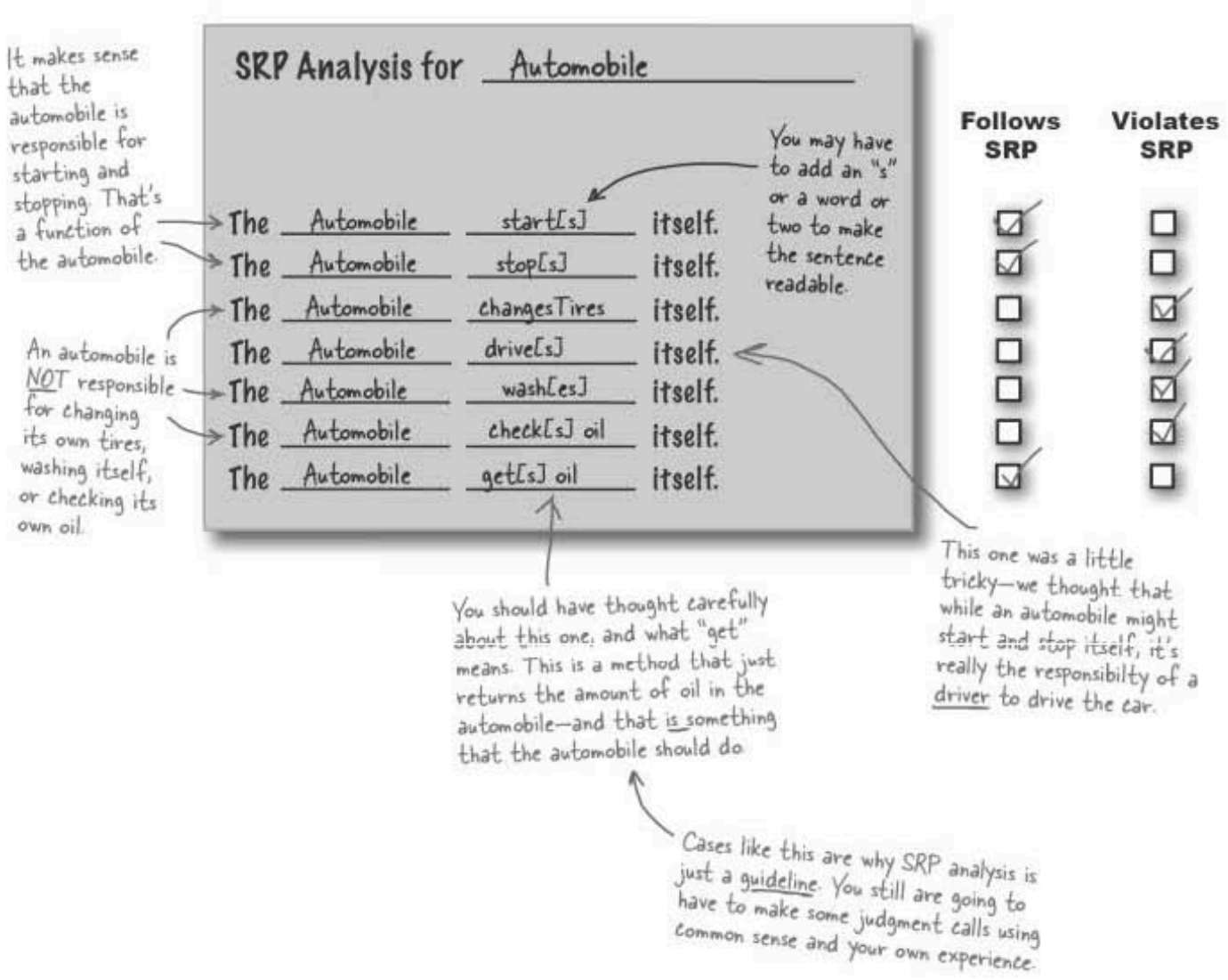
- A class should represent a single concept only.
- Translation:
  - Public methods and constants exposed by the public interface should be cohesive
  - All interface features should be closely related to the single concept

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

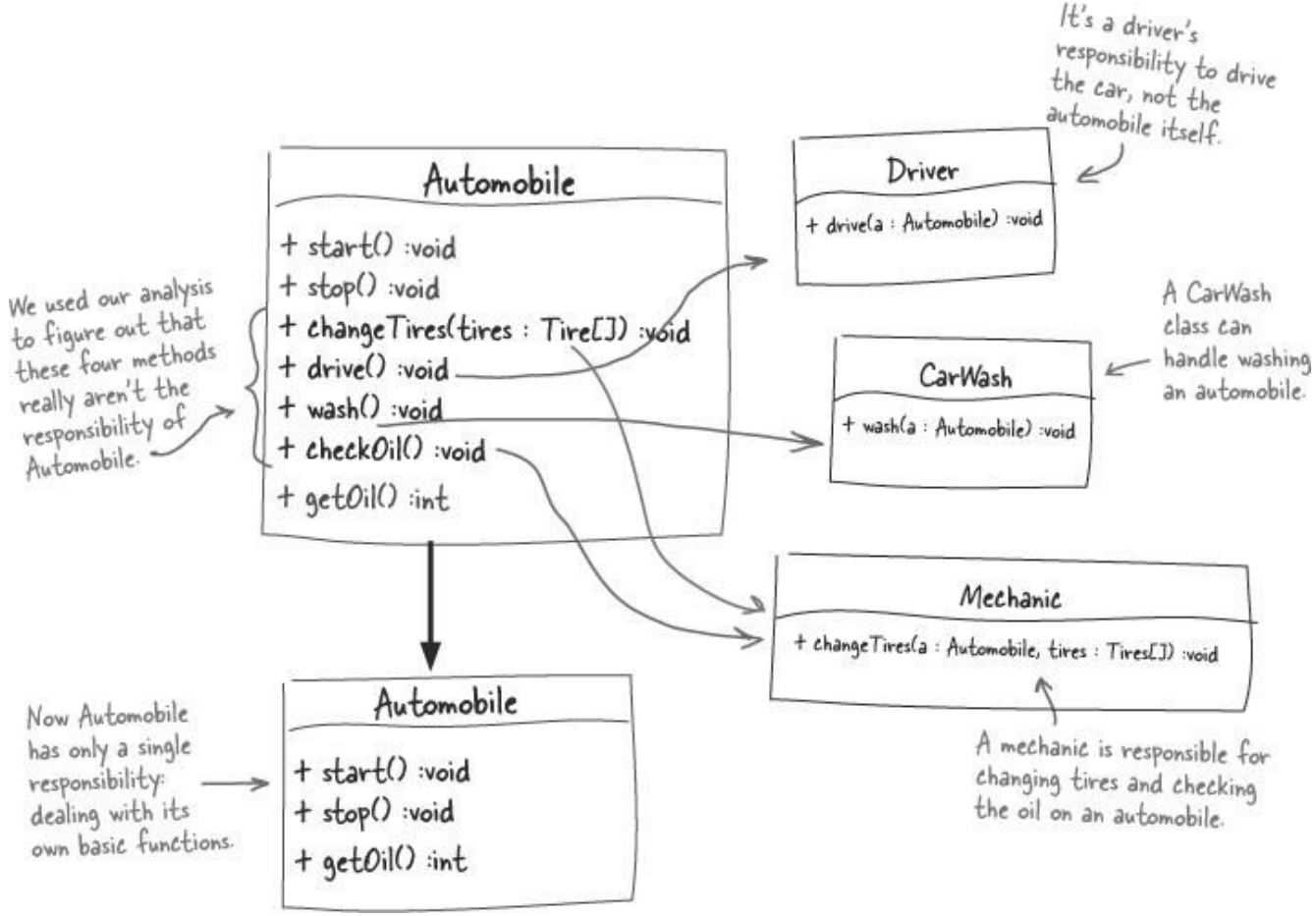
SPOTTING MULTIPLE RESPONSIBILITY IN YOUR DESIGN




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.
SRP Analysis for Automobile
You may have to add an "s" or a word or two to make the sentence readable.
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.
Follows SRP
Violates SRP
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.
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
Cases like this are why SRP analysis is just a guideline. You still are going to have to make some judgment calls using common sense and your own experience.

MIGRATING TO SINGLE RESPONSIBILITY

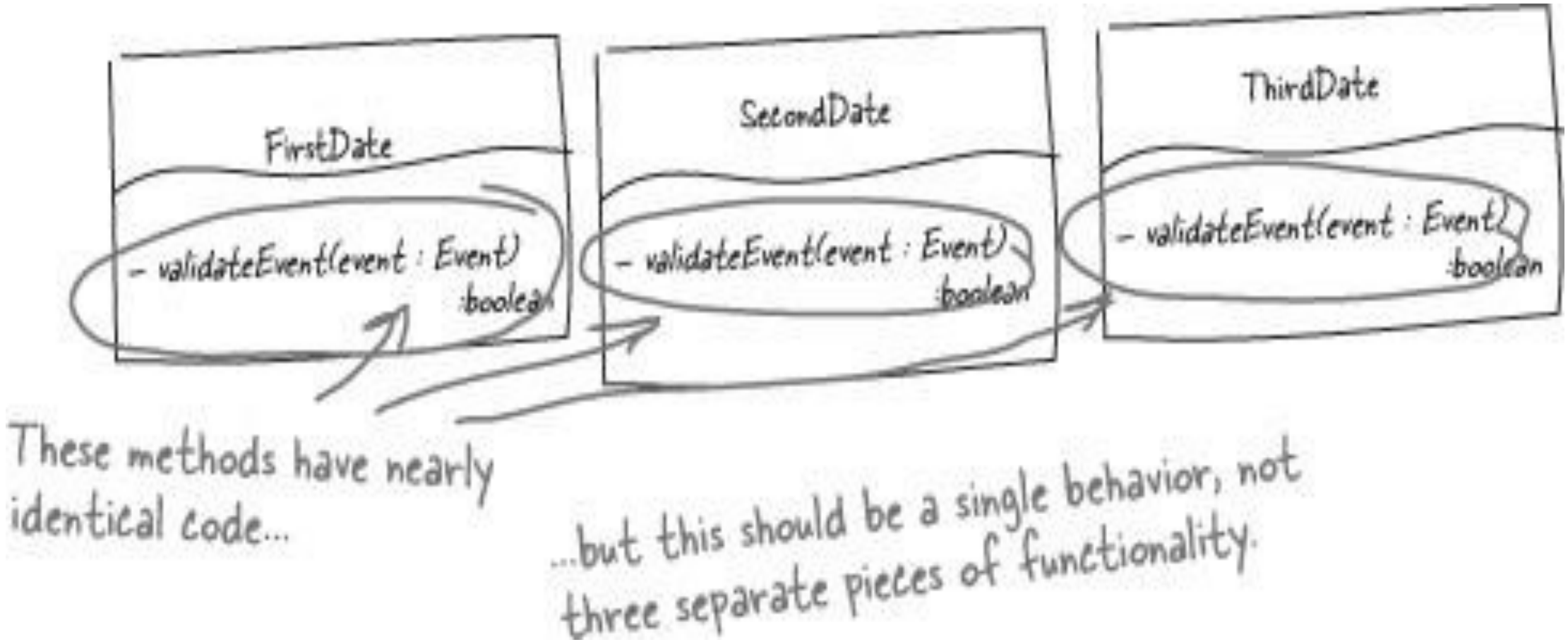
- Take all the methods that don't make sense on a class and move them to an appropriate class where it does make sense.



DON'T REPEAT YOURSELF

- 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.

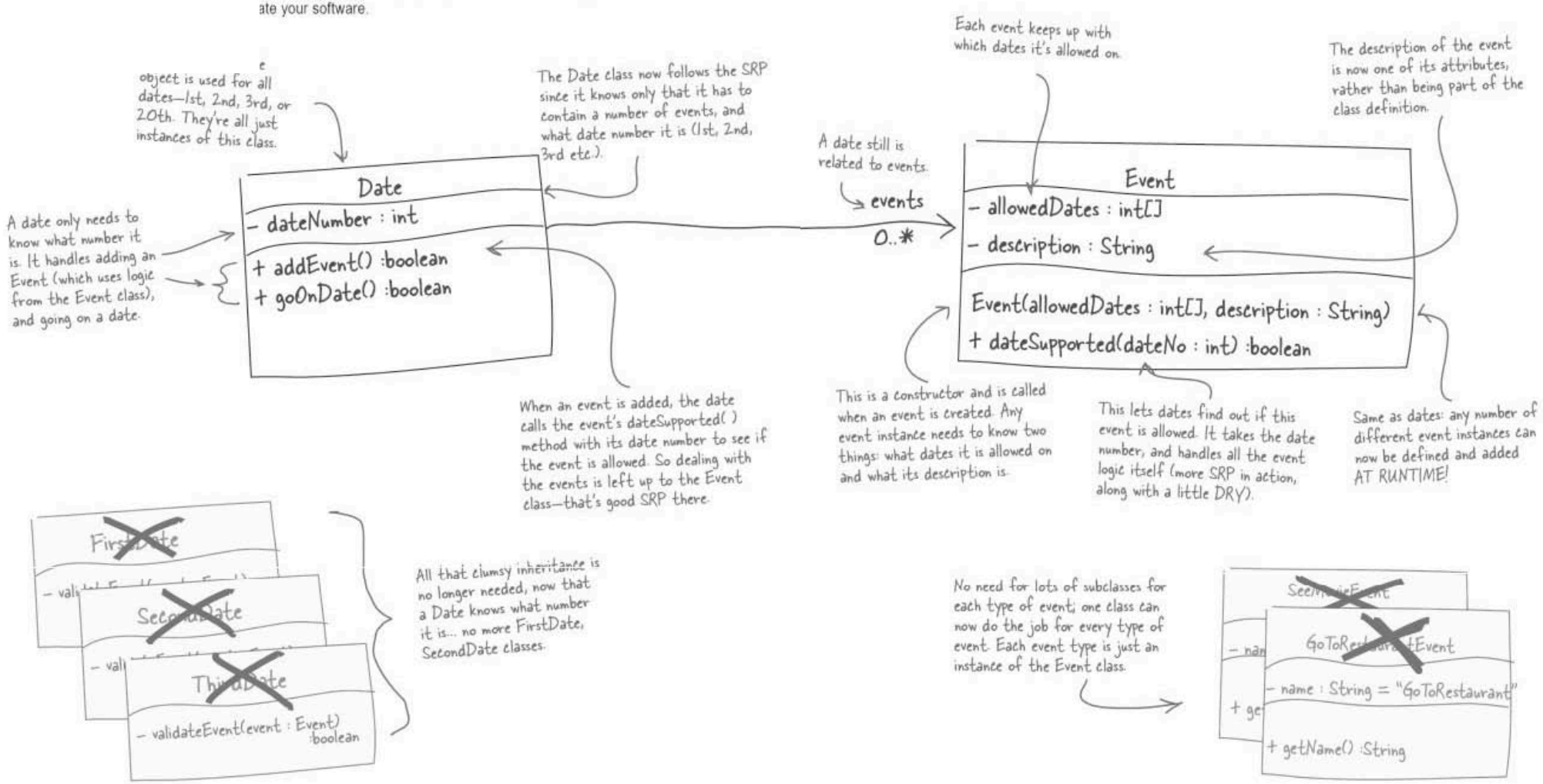
DRY EXAMPLE


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.

DRY Example

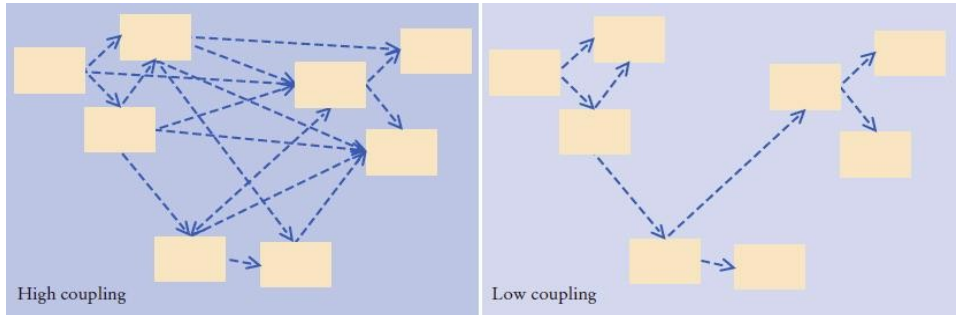
- SRP sounded a lot like DRY to me. Aren't both about a single class doing the one thing it's supposed to do?
  - They are related, and often appear together. DRY is about putting a piece of functionality in a single place, such as a class; SRP is about making sure that a class does only one thing, and 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.

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 ate your software.



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

- (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

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?