Writing Great Software

Recipes App

- Your recipes app has gone viral.
- Now it has 1M users!
- Does it still work?
  - Oops. Only one server.
  - Database needs to be sharded across multiple disks
- Want to integrate with a *shopping list* app: what changes are needed?

Did You Miss Some Requirements?

- You *always* miss some requirements!

Quality Attributes

- Express "non-functional requirements"
- Not *what* the system should do, but *how* it should do it
- Examples: modifiability, maintainability, performance, robustness
- Good design *promotes* some quality attributes
  - Sometimes at the expense of others

A Key: Abstraction

- Software is composed of *abstractions* (you already know this)
- This slideshow is a sequence of *slides*
  - Each slide has *objects*, each of which can draw itself
  - Somewhere there's code that asks objects to draw themselves
  - But that code doesn't know what the objects are!

Modifiability

- Can create a new kind of object without changing code that draws slides
- Can change how one object draws without knowing how another object draws
- Conclusion: *separating concerns* promotes modifiability

Readability

- What does this do?

```
#!/usr/local/bin/perl -s
do 'bigint.pl';($_,$n)=@ARGV;s/^.(..)*$/0$&/;($k=unpack('B*',pack('H*',$_)))=~
s/^0*//;$x=0;$z=$n=~s/./$x=&badd(&bmul($x,16),hex$&)/ge;while(read(STDIN,$_,$w
=((2*$d-1+$z)&~1)/2)){$r=1;$_=substr($_."\0"x$w,$c=0,$w);s/.|\n/$c=&badd(&bmul
($c,256),ord$&)/ge;$_=$k;s/./$r=&bmod(&bmul($r,$r),$x),$&?$r=&bmod(&bmul($r,$c
),$x):0,""/ge;($r,$t)=&bdiv($r,256),$_=pack(C,$t).$_ while$w--+1-2*$d;print}
```

Source: http://www.cypherspace.org/rsa/pureperl.html

Try Again…

```
#!/usr/local/bin/perl -s
#Above: full path for perl (may need to be changed on local system).
#       -s switch enables simple switch processing, which sets $d to 1
#        if "-d" is on the command line (it also removes the switch from ARGV).
#        if -d is not given $d is undefined (acts like 0)

#Load the standard bigint library.  Unlike require, do will not complain if
#the library is not present.  The space between do and the quotes is required
#(ha ha) in 4.036.
do 'bigint.pl';
#Set $_ to the key (e or d), and $n to n.
($_,$n)=@ARGV;
#For $_ (the key), if there are an odd number of characters,
#then add a leading zero.  This is needed for the pack below.
s/^.(..)*$/0$&/;
#pack hex digits to 8-bit binary, then unpack to ASCII binary, store in $k
#The outer parens are needed for precedence.
($k=unpack('B*',pack('H*',$_)))
#remove any leading zeros from $k
        =~s/^0*//;
#Extract $x (bigint version of $n).
#   $x=0;    Initialize bigint (needed?)
#   $z=      result of search/replace--the number of characters
```

It's an RSA Implementation.

- Obviously this was obfuscated. But what makes code easy or hard to read?
- (you tell me.)
- Often, readability is more important than performance

Readability

- What promotes maintainability at a *low level?*
  - Good functional decomposition
  - Good identifier names
  - Good formatting
  - Avoiding repetition

SOLID Principles for Design

- Robert C. Martin proposed five principles of object-oriented design
- Conveniently, these apply to TypeScript as well!

Goals for Today

- **SOLID** principles:
  - SRP: Single Responsibility Principle
  - Open-closed principle
  - Liskov substitution principle
  - Interface segregation principle
  - Dependency inversion principle
- Also: DRY: Don't Repeat Yourself

Open-Closed Principle

- Classes should be open for extension but closed for modification
- i.e. enable extending class without modifying the class
  - Can do this via subclassing or via interfaces

Liskov Substitution Principle

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

LSP Violation

```
public interface Car {

    void turnOnEngine();
    void accelerate();
}
```

```
public class MotorCar implements Car {

    private Engine engine;

    //Constructors, getters + setters

    public void turnOnEngine() {
        //turn on the engine!
        engine.on();
    }

    public void accelerate() {
        //move forward!
        engine.powerOn(1000);
    }
}
```

```
public class ElectricCar implements Car {

    public void turnOnEngine() {
        throw new AssertionError("I don't have an engine!");
    }

    public void accelerate() {
        //this acceleration is crazy!
    }
}
```

Credit: https://www.baeldung.com/solid-principles

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

High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces).

Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

- Goal is to avoid tight coupling
- *n.b.* **not** the same as *dependency injection*
- Can be applied *too much*

Martin, Robert C. (2003). Agile Software Development, Principles, Patterns, and Practices

Dependency Inversion (Non)-Example

```
class OrderService {
  database: MySQLDatabase;

  public create(order: Order): void {
    this.database.create(order)
  }

  public update(order: Order): void {
    this.database.update
  }
}
```

```
class MySQLDatabase {
  public create(order: Order) {
    // create and insert to database
  }

  public update(order: Order) {
    // update database
  }
}
```

Changes in `MySQLDatabase` may propagate to `OrderService` 😕

With Dependency Inversion

```
interface Database {
  create(order: Order): void;
  update(order: Order): void;
}

class OrderService {
  database: Database;

  public create(order: Order): void {
    this.database.create(order);
  }

  public update(order: Order): void {
    this.database.update(order);
  }
}

class MySQLDatabase implements Database {
  public create(order: Order) {
    // create and insert to database
  }

  public update(order: Order) {
```

Database interface avoids dependency

For Rest of Today: DRY and SRP

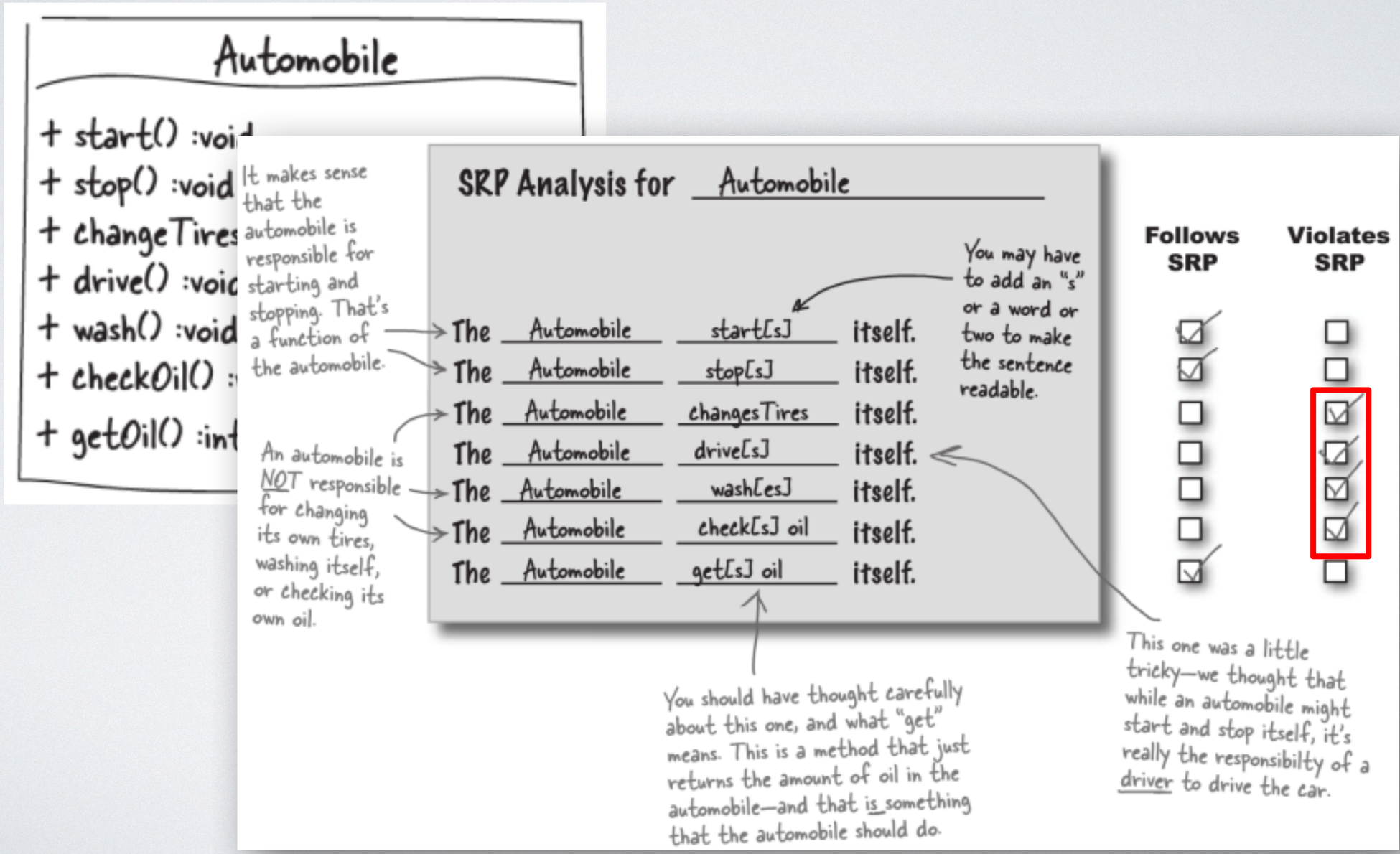
Thing-Ness Simplified: the **S**ingle **R**esponsibility **P**rinciple (**SRP**)

- A class should be responsible for one 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)

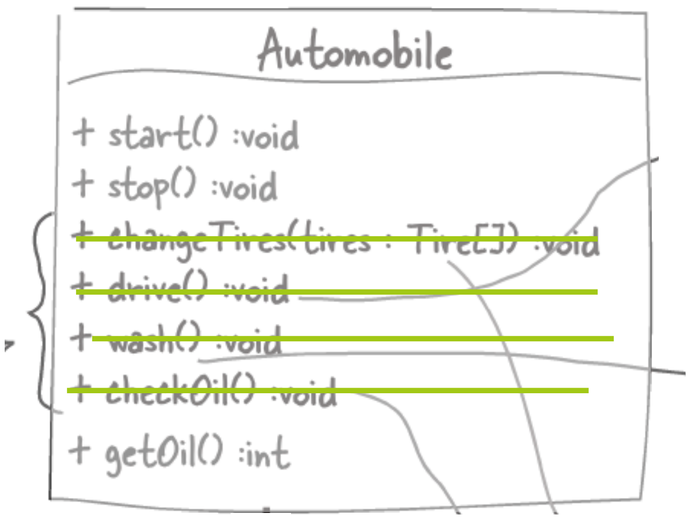
Oops: Cramming Related Functionality Into a Single Class



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**

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.)

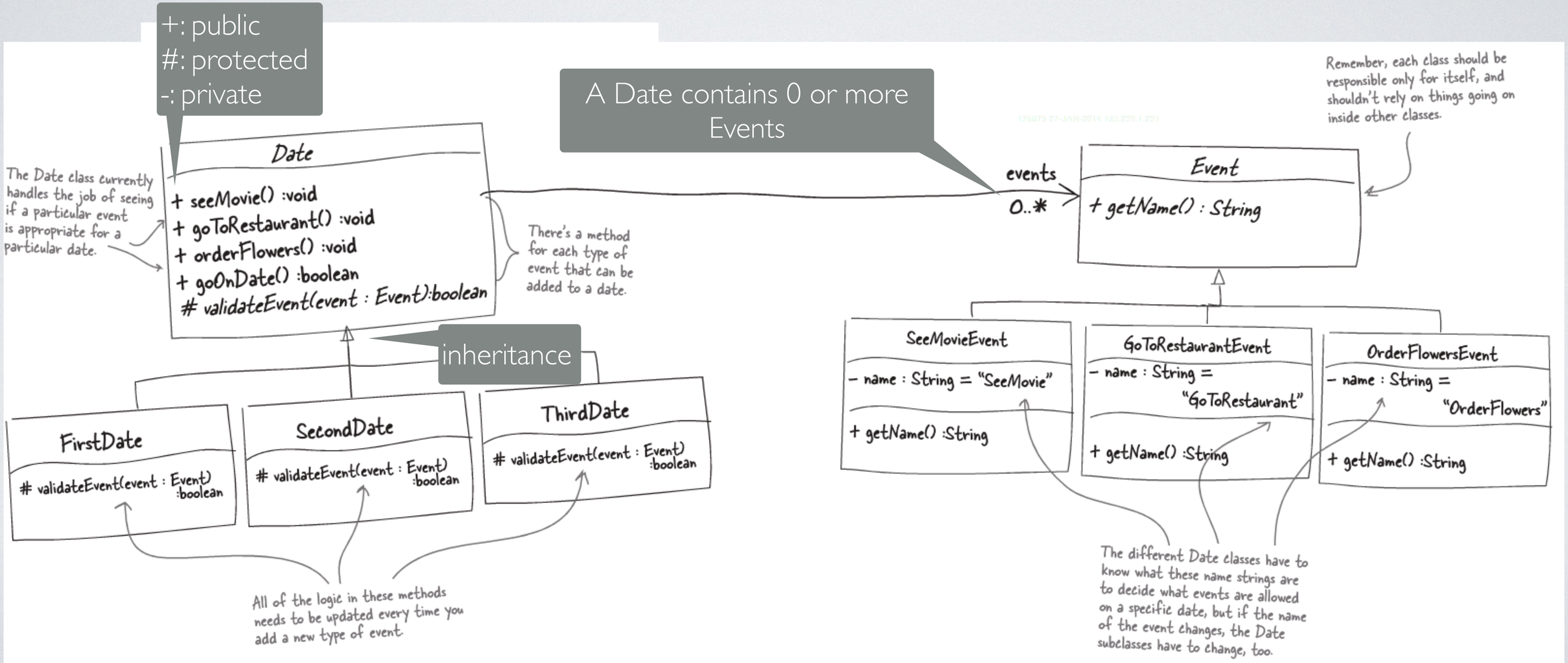
Thing-Ness Simplified:
Don't **R**epeat **Y**ourself (**DRY**)

- **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



Credit: *Head First Software Development*

~/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?

Better phra
A.Date doe
B.Changes
type) req

~/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

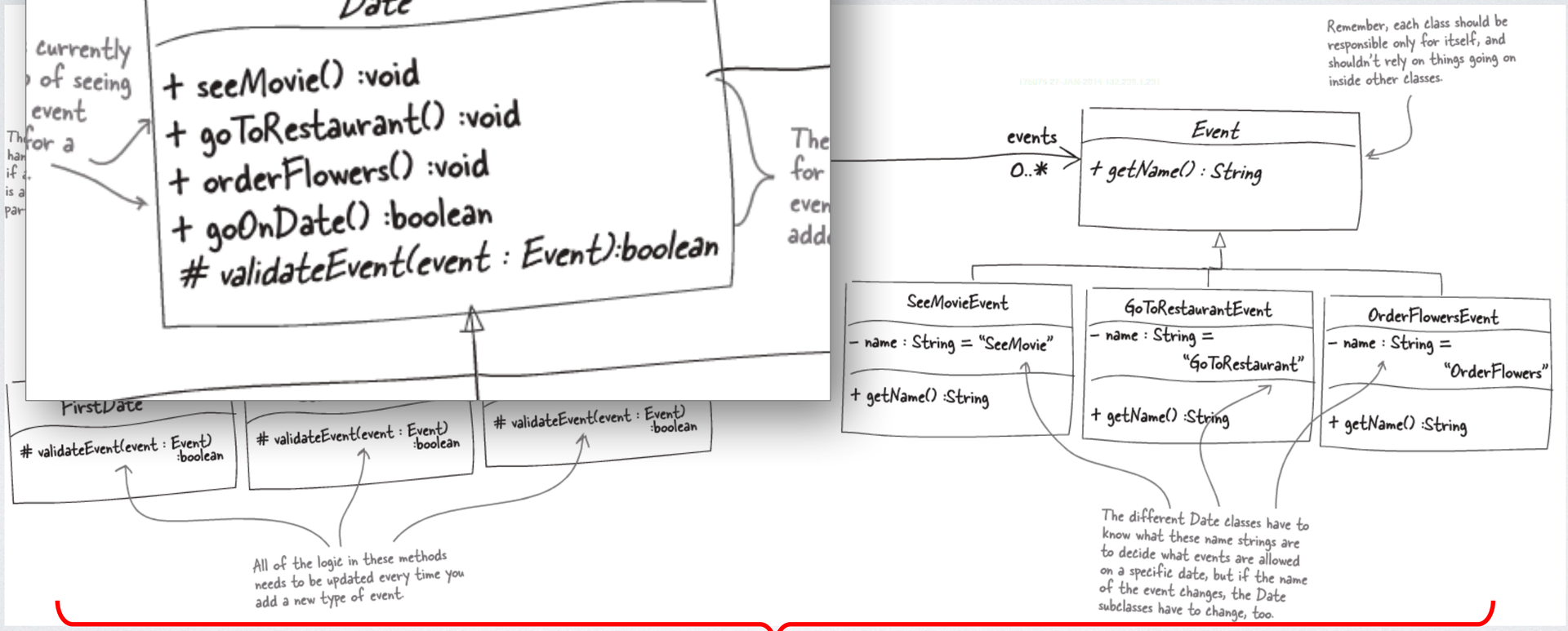
String, not class for each event!

```
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

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

- Possible to diagnose 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
- Often, **iteration** and **peer feedback** can help you improve your design
- Unfortunately, there are many kinds of design mistakes, and unique repairs for them