CSE203 Principles of Software Design and Development

Baby, somebody better call heaven, 'cause they're missing an angel tonight!
I must have heard that line a thousand times, but it works on me every time!

08. Design Principles

Originality is Overrated

it's not about "doing it your way"
this week is all about **doing it the smarter, faster way.**

1

Design principle roundup

So far, we've concentrated on the things you do before you start coding your application.

- Gathering requirements,
- analysis,
- writing out feature lists,
- and drawing use case diagrams.

*A **design principle** is a basic tool or technique that can be applied to designing or writing code to make that code more maintainable, flexible, or extensible.*

Of course, at some point you actually are going to have to write some code. And that's where design principles really come into play.

OO Principles

Encapsulate what varies.

Code to an interface rather than to an implementation.

Each class in your application should have only one reason to change.

Classes are about behavior and functionality.

We're going to look at several more key design principles, and how each one can improve the design and implementation of your code.

We'll even see that sometimes you'll have to choose between two design principles…

Using proven OO design principles results in more maintainable, flexible, and extensible software.

2

Principle #1: The Open-Closed Principle (OCP)

Our first design principle is the OCP, or the Open-Closed principle. The OCP is all about **allowing change**, but doing it **without requiring you to modify existing code**.

Open-Closed Principle

Classes should be open for extension, and closed for modification.

Closed for modification...

Suppose you have a class with a particular behavior, and you've got that behavior coded up just the way you want it. Make sure that nobody can change your class's code, and you've made that particular piece of behavior *closed for modification*.

You close classes by not allowing anyone to touch your working code.

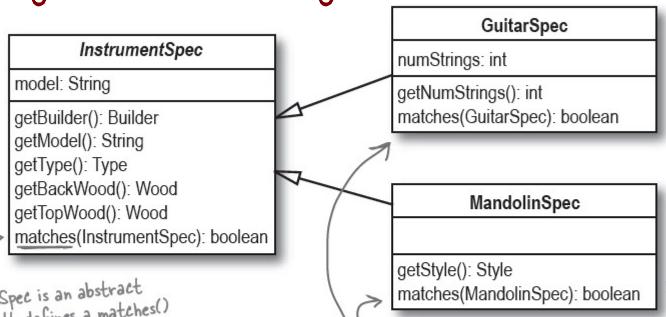
...but open for extension

But let them subclass your class, and then they can override your method to work like they want it to. So even though they didn't mess with your working code, you still left your class *open for extension*.

You open classes by allowing them to be subclassed and extended.

3

Remember working on Rick's Stringed Instruments?



InstrumentSpec
model: String
getBuilder(): Builder
getModel(): String
getType(): Type
getBackWood(): Wood
getTopWood(): Wood
matches(InstrumentSpec): boolean

GuitarSpec
numStrings: int
getNumStrings(): int
matches(GuitarSpec): boolean

MandolinSpec
getStyle(): Style
matches(MandolinSpec): boolean

InstrumentSpec is an abstract base class. It defines a matches() method that has a basic spec-matching implementation

Each of the instrument-specific subclasses extend the matches() method... they use the base version from InstrumentSpec, but then add some extra matching detail specific to the instrument they work with.

The parts of matches() that change across subclasses are encapsulated away from the InstrumentSpec base class.

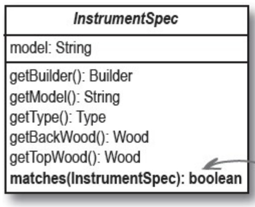
`InstrumentSpec` is closed for modification; the `matches()` method is defined in the base class and doesn't change.

But it's open for extension, because all of the subclasses can change the behavior of `matches()`.

4

The OCP, step-by-step

1. We coded **matches()** in InstrumentSpec.java, and closed it for modification.

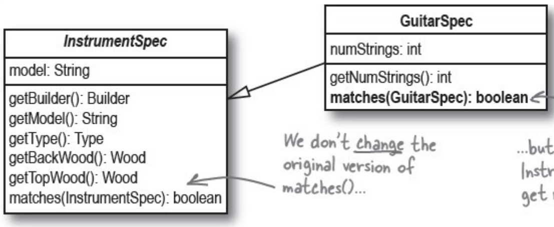


InstrumentSpec
model: String
getBuilder(): Builder getModel(): String getType(): Type getBackWood(): Wood getTopWood(): Wood **matches(InstrumentSpec): boolean**

This method works fine, so we don't want anyone else touching it.

2. But we needed to modify **matches()** to work with instrument-specific spec classes.

3. So we extended InstrumentSpec, and overrode **matches()** to change its behavior.



InstrumentSpec
model: String
getBuilder(): Builder getModel(): String getType(): Type getBackWood(): Wood getTopWood(): Wood matches(InstrumentSpec): boolean

GuitarSpec
numStrings: int
getNumStrings(): int **matches(GuitarSpec): boolean**

We don't change the original version of matches()...

...but we can extend InstrumentSpec, and still get new behavior.

Gee, inheritance is powerful. Really, this is supposed to be some sort of great design principle? Come on.

The OCP is about flexibility, and goes beyond just inheritance.

Principle #2: The Don't Repeat Yourself Principle (DRY)

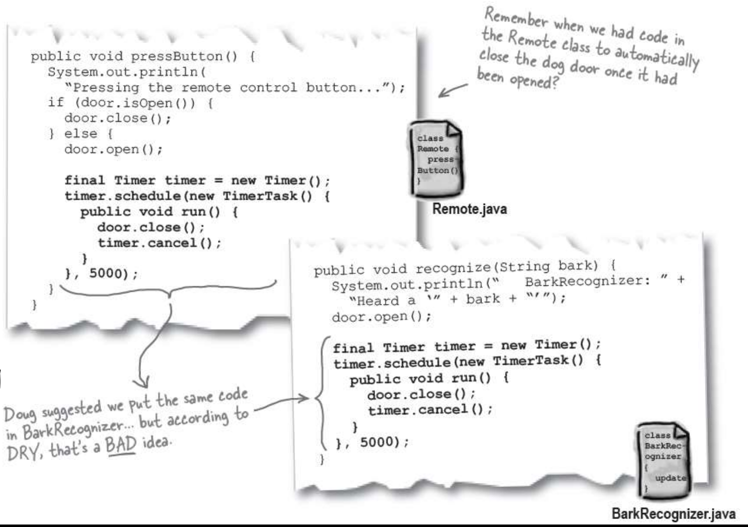
This is another principle that looks pretty simple, but turns out to be critical in writing code that's easy to maintain and reuse.

Don't Repeat Yourself

Avoid duplicate code by abstracting out things that are common and placing those things in a single location.

A prime place to apply DRY...

You've seen the DRY principle in action, even if you didn't realize it. We used DRY when Todd and Gina wanted us to close the dog door automatically after it had been opened.



```java
public void pressButton() {
  System.out.println(
    "Pressing the remote control button...");
  if (door.isOpen()) {
    door.close();
  } else {
    door.open();

    final Timer timer = new Timer();
    timer.schedule(new TimerTask() {
      public void run() {
        door.close();
        timer.cancel();
      }
    }, 5000);
  }
}
```

Remote.java

Remember when we had code in the Remote class to automatically close the dog door once it had been opened?

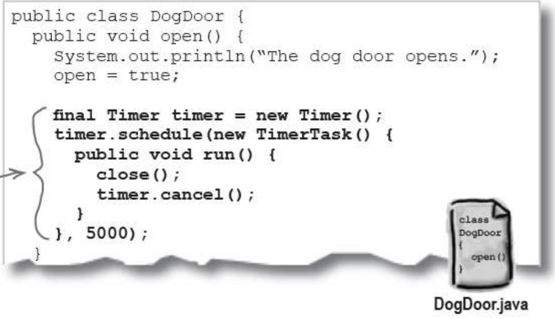
```java
public void recognize(String bark) {
  System.out.println("   BarkRecognizer: " +
    "Heard a '" + bark + "'");
  door.open();

  final Timer timer = new Timer();
  timer.schedule(new TimerTask() {
    public void run() {
      door.close();
      timer.cancel();
    }
  }, 5000);
}
```

BarkRecognizer.java

Doug suggested we put the same code in BarkRecognizer... but according to DRY, that's a BAD idea.

1. Let's abstract out the common code.

Using DRY, we first need to take the code that's common between **Remote** and **BarkRecognizer**, and put it in a single place. We figured out the best place for it was in the **DogDoor** class:

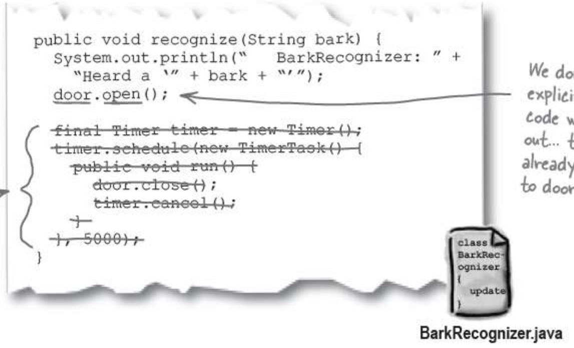


```java
public class DogDoor {
  public void open() {
    System.out.println("The dog door opens.");
    open = true;

    final Timer timer = new Timer();
    timer.schedule(new TimerTask() {
      public void run() {
        close();
        timer.cancel();
      }
    }, 5000);
  }
```

DogDoor.java

Using DRY, we pull out all this code from Remote and BarkRecognizer, and put it in ONE place: the DogDoor class. So no more duplicate code, no more maintenance nightmares.

2. Now remove code from other locations and reference the code from Step #1.



```java
public void recognize(String bark) {
  System.out.println("   BarkRecognizer: " +
    "Heard a '" + bark + "'");
  door.open();
```

~~final Timer timer = new Timer();~~
~~timer.schedule(new TimerTask() {~~
~~public void run() {~~
~~door.close();~~
~~timer.cancel();~~
~~}~~
~~}, 5000);~~

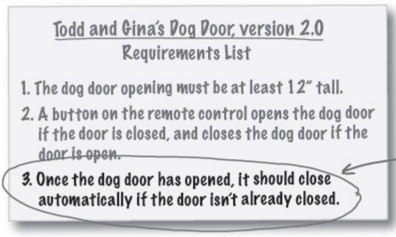
```java
}
```

BarkRecognizer.java

First, we got rid of this code... it's all in DogDoor's open() method now.

We don't have to explicitly call the code we abstracted out... that's handled already by our call to door.open().

DRY is really about ONE requirement in ONE place

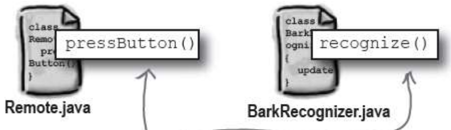

Todd and Gina's Dog Door, version 2.0
Requirements List

1. The dog door opening must be at least 12" tall.
2. A button on the remote control opens the dog door if the door is closed, and closes the dog door if the door is open.
3. Once the dog door has opened, it should close automatically if the door isn't already closed.

Here's the single requirement we're focusing on here.

Abstracting out duplicate code is a good start to using DRY, but there's more to it than just that. When you're trying to avoid duplicate code, you're really trying to make sure that you only implement each feature and requirement in our application one single time.

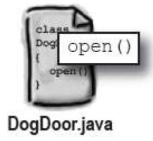
Originally, though, we implemented that single feature in *two* places: **`Remote.java`** and **`BarkRecognizer.java`**.



BOTH of these methods have code that closes the dog door.

DRY is about having each piece of information and behavior in your system in a <u>single, sensible place.</u>

By using DRY, we removed the duplicate code. But more importantly, we moved the implementation of this requirement, automatically closing the door, into *one* place, instead of *two* places:



Now there is just ONE place we automatically close the door: open(), in DogDoor.

Design Puzzle

DRY is about a lot more than just finding duplicate code in your system. It also applies to your features and requirements. It's time to put DRY into action on your own now, and to do it in more than just code.

These are the requirements you've already seen...

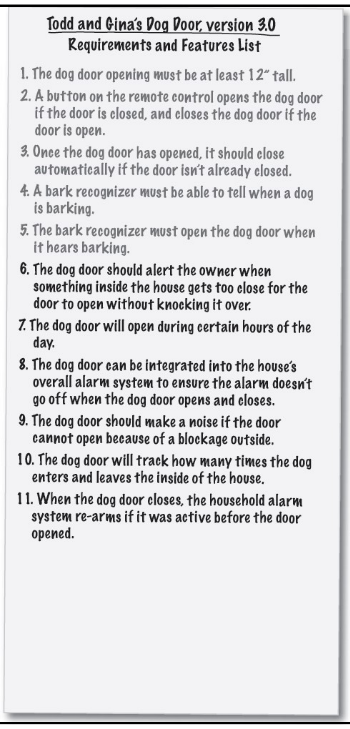
...and these are the new features and requirements.

The problem:

Todd and Gina have come up with yet more features for their dog door. It's your job to make sure the feature list we've assembled doesn't have any duplication issues, and that each feature is handled once and only once in the system you're designing for them.

Your task:

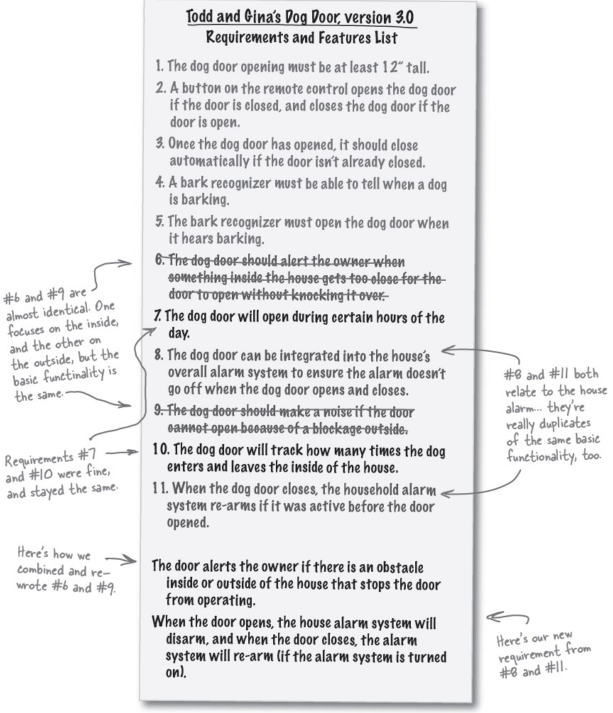
1. Read through the requirements and features list on the right. We've bolded the requirements and features that have been added since you last worked on the dog door.
2. Look through the new features and requirements, and see if you see any possible duplication in the new things you'd need to build.
3. Annotate the requirements and features list indicating what you think has been duplicated.
4. Rewrite the duplicate requirements at the bottom of the list so that there is no more duplication.
5. Write a new definition for the DRY principle in the space below, and make sure you talk about more than just duplicate code.


Todd and Gina's Dog Door, version 3.0
Requirements and Features List

1. The dog door opening must be at least 12" tall.
2. A button on the remote control opens the dog door if the door is closed, and closes the dog door if the door is open.
3. Once the dog door has opened, it should close automatically if the door isn't already closed.
4. A bark recognizer must be able to tell when a dog is barking.
5. The bark recognizer must open the dog door when it hears barking.
6. **The dog door should alert the owner when something inside the house gets too close for the door to open without knocking it over.**
7. **The dog door will open during certain hours of the day.**
8. **The dog door can be integrated into the house's overall alarm system to ensure the alarm doesn't go off when the dog door opens and closes.**
9. **The dog door should make a noise if the door cannot open because of a blockage outside.**
10. **The dog door will track how many times the dog enters and leaves the inside of the house.**
11. **When the dog door closes, the household alarm system re-arms if it was active before the door opened.**

Write any new or updated requirements, without duplication, here at the bottom of the list.

Design Puzzle Solution



Todd and Gina's Dog Door, version 3.0

Requirements and Features List

1. The dog door opening must be at least 12" tall.
2. A button on the remote control opens the dog door if the door is closed, and closes the dog door if the door is open.
3. Once the dog door has opened, it should close automatically if the door isn't already closed.
4. A bark recognizer must be able to tell when a dog is barking.
5. The bark recognizer must open the dog door when it hears barking.
6. ~~The dog door should alert the owner when something inside the house gets too close for the door to open without knocking it over.~~
7. The dog door will open during certain hours of the day.
8. The dog door can be integrated into the house's overall alarm system to ensure the alarm doesn't go off when the dog door opens and closes.
9. ~~The dog door should make a noise if the door cannot open because of a blockage outside.~~
10. The dog door will track how many times the dog enters and leaves the inside of the house.
11. When the dog door closes, the household alarm system re-arms if it was active before the door opened.

The door alerts the owner if there is an obstacle inside or outside of the house that stops the door from operating.

When the door opens, the house alarm system will disarm, and when the door closes, the alarm system will re-arm (if the alarm system is turned on).

#6 and #9 are almost identical. One focuses on the inside, and the other on the outside, but the basic functinality is the same.

Requirements #7 and #10 were fine, and stayed the same.

Here's how we combined and re-wrote #6 and #9.

#8 and #11 both relate to the house alarm... they're really duplicates of the same basic functionality, too.

Here's our new requirement from #8 and #11.

Principle #3: The Single Responsibility Principle (SRP)

The SRP is all about responsibility, and which objects in your system do what. You want each object that you design to have just one responsibility to focus on—and when something about that responsibility changes, you'll know exactly where to look to make those changes in your code.

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.

Hey, we've talked about this before... this is the same as a class having only one reason to change, isn't it?

You've implemented the Single Responsibility Principle correctly when each of your objects has <u>only one reason to change</u>.

Spotting multiple responsibilities

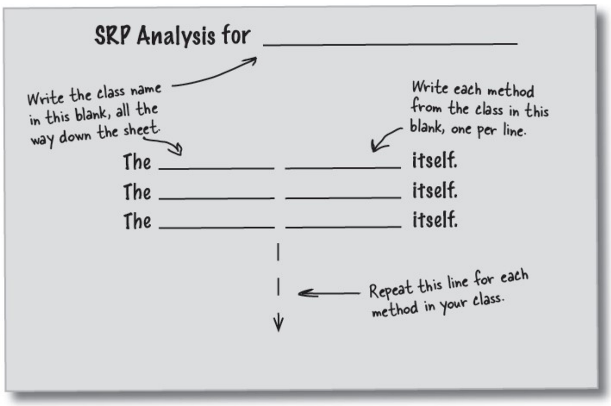
Most of the time, you can spot classes that aren't using the SRP with a simple test:

1. On a sheet of paper, write down a bunch of lines like this: The [blank] [blanks] itself. You should have a line like this for every method in the class you're testing for the SRP.

2. In the first blank of each line, write down the class name; in the second blank, write down one of the methods in the class. Do this for each method in the class.

3. Read each line out loud (you may have to add a letter or word to get it to read normally). Does what you just said make any sense? Does your class really have the responsibility that the method indicates it does?

IF WHAT YOU'VE JUST SAID DOESN'T MAKE SENSE, THEN YOU'RE PROBABLY VIOLATING THE SRP WITH THAT METHOD. THE METHOD MIGHT BELONG ON A DIFFERENT CLASS... THINK ABOUT MOVING IT.



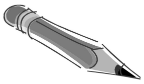
SRP Analysis for ____________

Write the class name in this blank, all the way down the sheet.

Write each method from the class in this blank, one per line.

The ________ ________ itself.
The ________ ________ itself.
The ________ ________ itself.

Repeat this line for each method in your class.

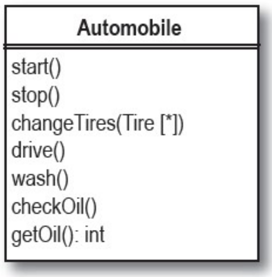


Sharpen your pencil

Apply the SRP to the Automobile class?

Do an SRP analysis on the Automobile class shown below. Fill out the sheet with the class name methods in Automobile, like we've described on the last slide. Then, decide if you think it makes sense for the Automobile class to have each method, and check the right box.

We looked at this class in CATASTROPHE



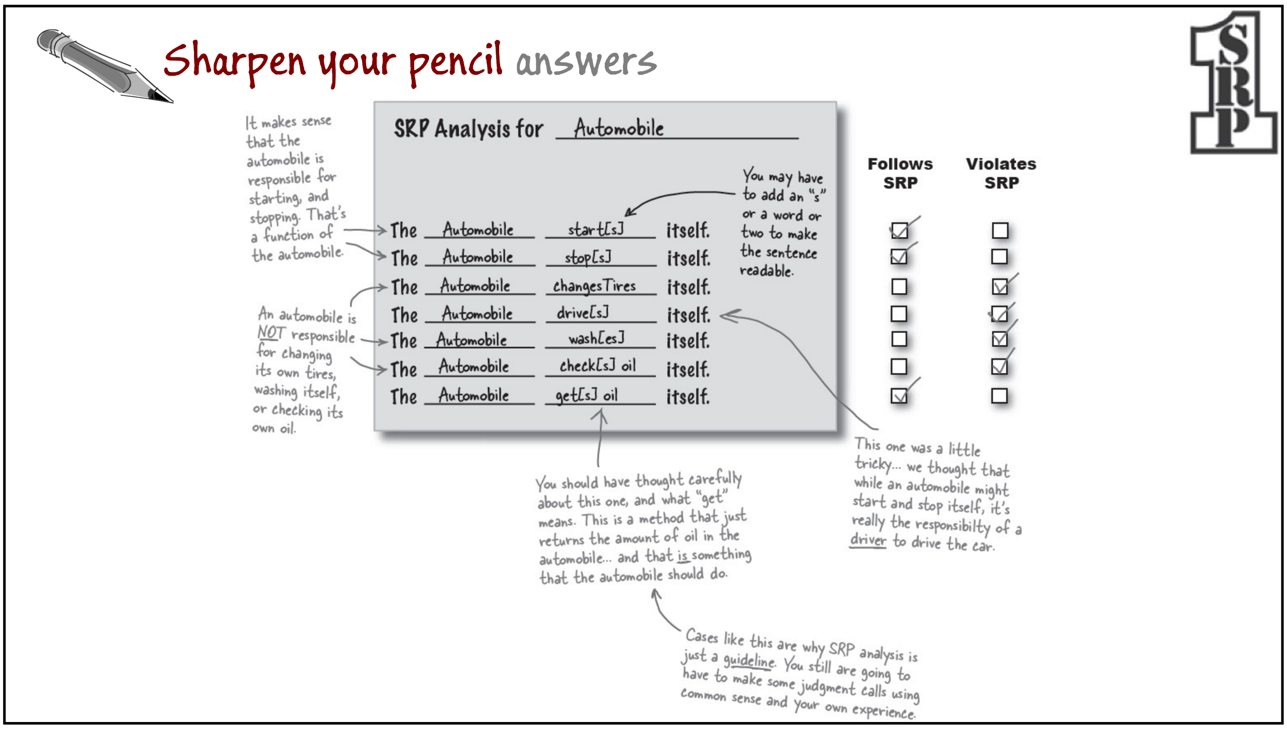
Automobile

start()
stop()
changeTires(Tire [*])
drive()
wash()
checkOil()
getOil(): int

SRP Analysis for Automobile

SRP Analysis for Automobile	Follows SRP	Violates SRP
The ________ ________ itself.	□	□
The ________ ________ itself.	□	□
The ________ ________ itself.	□	□
The ________ ________ itself.	□	□
The ________ ________ itself.	□	□
The ________ ________ itself.	□	□
The ________ ________ itself.	□	□

If what you read doesn't make sense, then the method on that line is probably violating the SRP.



Sharpen your pencil answers

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.

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

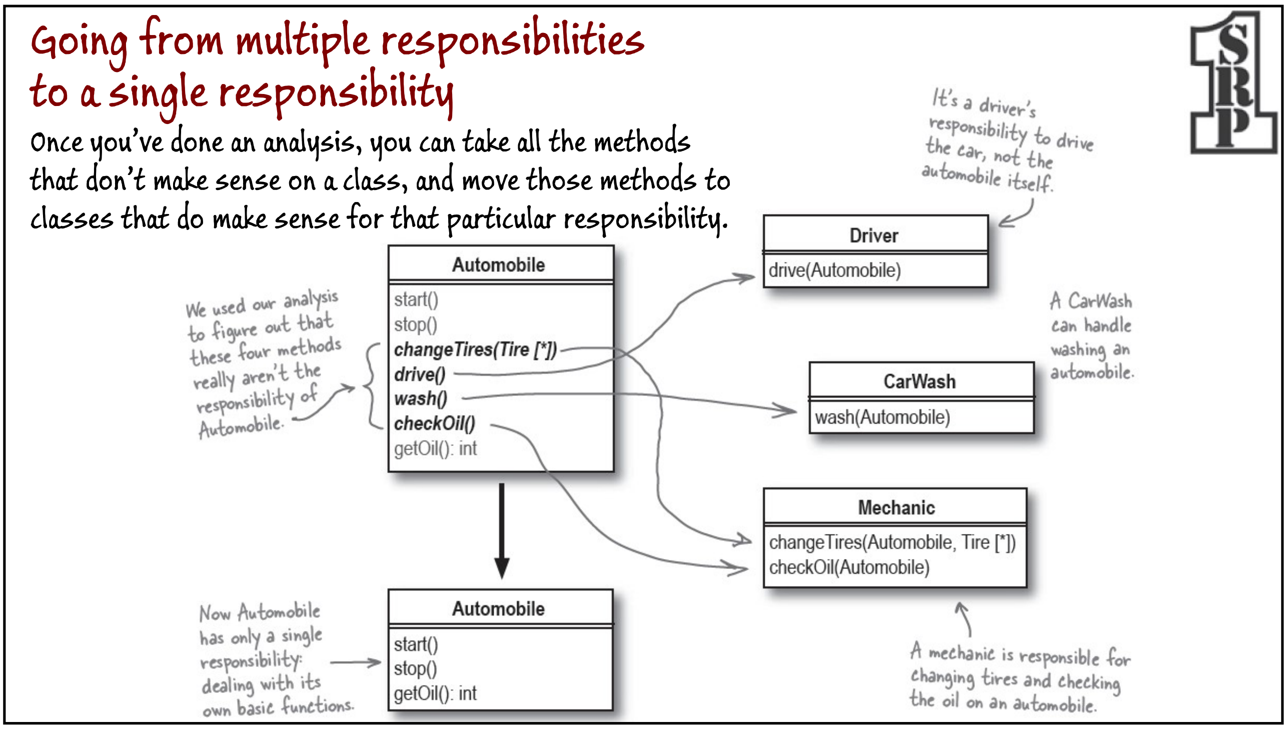
An automobile is NOT responsible for changing its own tires, washing itself, or checking its own oil.

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.

15



Going from multiple responsibilities to a single responsibility

Once you've done an analysis, you can take all the methods that don't make sense on a class, and move those methods to classes that do make sense for that particular responsibility.

Automobile
- start()
- stop()
- *changeTires(Tire [*])*
- *drive()*
- *wash()*
- *checkOil()*
- getOil(): int

We used our analysis to figure out that these four methods really aren't the responsibility of Automobile.

Driver
- drive(Automobile)

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

CarWash
- wash(Automobile)

A CarWash can handle washing an automobile.

Mechanic
- changeTires(Automobile, Tire [*])
- checkOil(Automobile)

A mechanic is responsible for changing tires and checking the oil on an automobile.

Automobile
- start()
- stop()
- getOil(): int

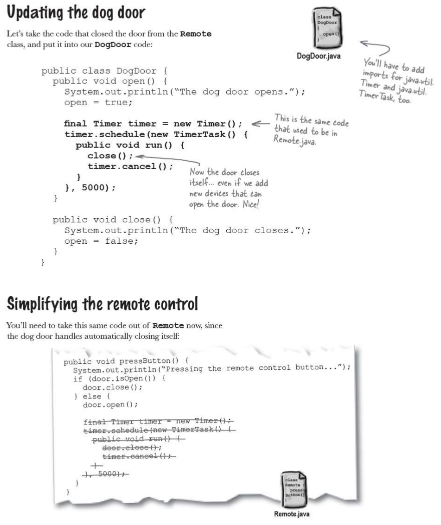
Now Automobile has only a single responsibility: dealing with its own basic functions.

16

SRP Sightings

How do you think the Single Responsibility Principle was used in Todd and Gina's dogdoor? Write your answer in the blanks below:

We moved the code to close the dog door out of Remote.java, and avoided duplicating the same code in the BarkRecognizer (DRY in effect there!). We also made sure that the DogDoor class handled all tasks relating to the operation of the dog door—it has that single responsibility.



Updating the dog door

Let's take the code that closed the door from the **Remote** class, and put it into our **DogDoor** code:

DogDoor.java

You'll have to add imports for java.util.Timer and java.util.TimerTask, too.

```
public class DogDoor {
  public void open() {
    System.out.println("The dog door opens.");
    open = true;

    final Timer timer = new Timer();
    timer.schedule(new TimerTask() {
      public void run() {
        close();
        timer.cancel();
      }
    }, 5000);
  }

  public void close() {
    System.out.println("The dog door closes.");
    open = false;
  }
}
```

This is the same code that used to be in Remote.java.

Now the door closes itself... even if we add new devices that can open the door. Nice!

Simplifying the remote control

You'll need to take this same code out of **Remote** now, since the dog door handles automatically closing itself:

```
public void pressButton() {
  System.out.println("Pressing the remote control button...");
  if (door.isOpen()) {
    door.close();
  } else {
    door.open();

    ~~final Timer timer = new Timer();~~
    ~~timer.schedule(new TimerTask() {~~
      ~~public void run() {~~
        ~~door.close();~~
        ~~timer.cancel();~~
      ~~}~~
    ~~}, 5000);~~
  }
}
```

Remote.java

Now see if you can find two more instances in the book's examples so far where we've used the SRP to make our design better and more flexible. You can find the SRP in the dog door, Rick's instrument inventory searcher, or Gary's game framework. Write down each instance you found, and how you think the SRP is being used.

First Instance

Example application: X Rick's Instruments ___ Doug's Dog Doors ___ Gary's Games

How SRP is being used:

We created a matches() method on InstrumentSpec, rather than leaving the code to compare instruments in the search() method of Inventory. So an InstrumentSpec handles everything related to an instrument's properties—that code isn't spread out over other classes. That's SRP in action.

Second Instance

Example application: ___ Rick's Instruments ___ Doug's Dog Doors X Gary's Games

How SRP is being used:

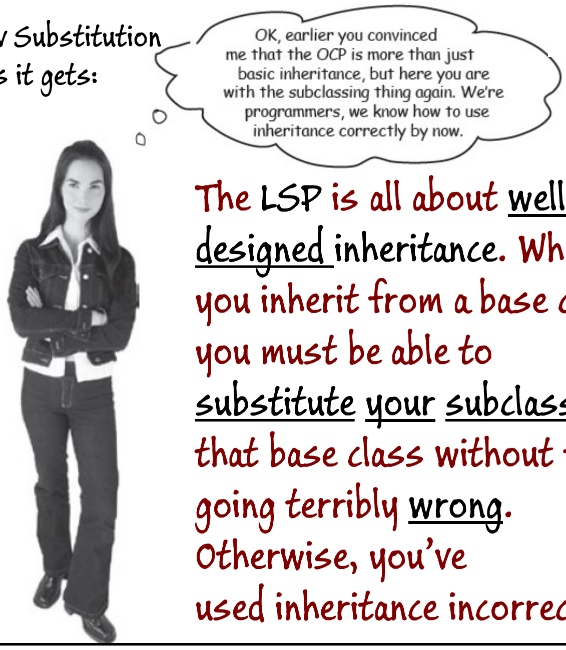
When we used a Map to store properties for all types of units in the Unit class, we were using the SRP. So instead of having game-specific Units have to deal with their properties, and still have the base Unit class dealing with a different set of properties, we moved all property-related functionality into the Unit class. So handling the properties feature is taken care of in ONE single place—the Unit class.

Contestant #4: The Liskov Substitution Principle (LSP)

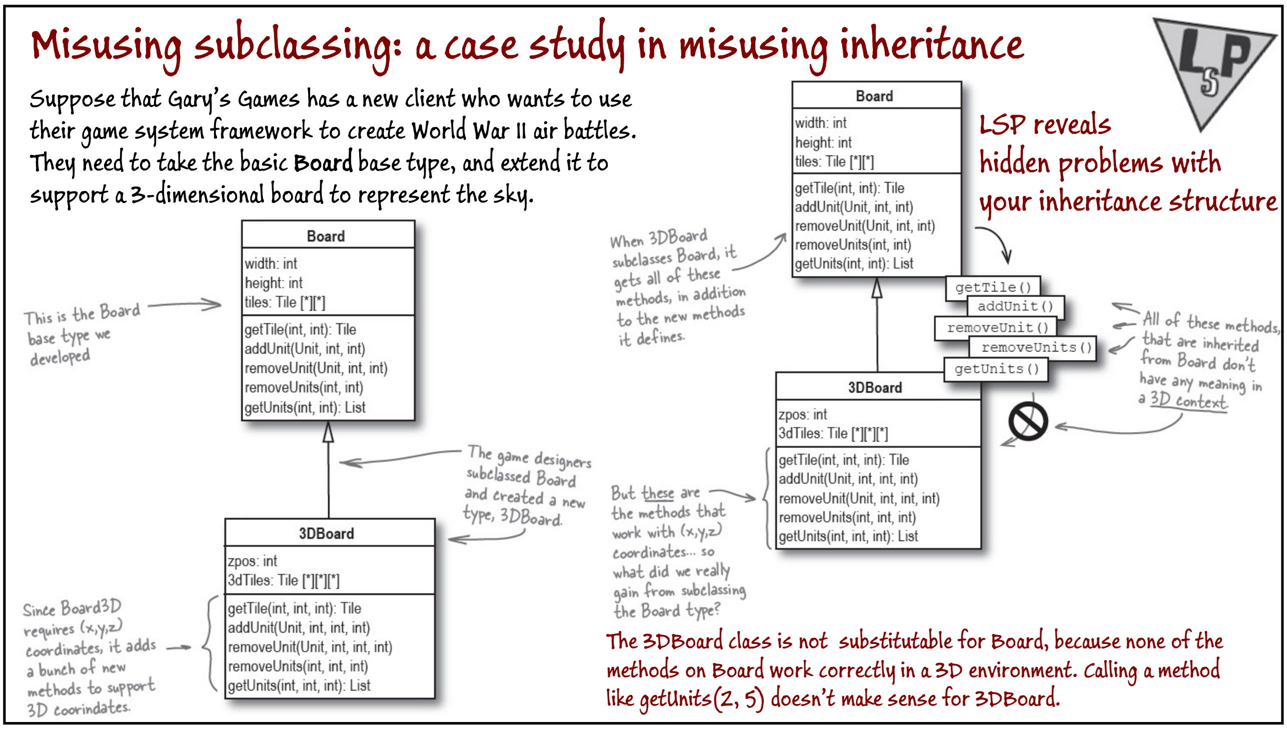
Next up in our design principle parade is the Liskov Substitution Principle, or the LSP. It's definition is as simple as it gets:

Liskov Substitution Principle

Subtypes must be substitutable for their base types.


OK, earlier you convinced me that the OCP is more than just basic inheritance, but here you are with the subclassing thing again. We're programmers, we know how to use inheritance correctly by now.

The LSP is all about well-designed inheritance. When you inherit from a base class, you must be able to substitute your subclass for that base class without things going terribly wrong. Otherwise, you've used inheritance incorrectly!



Misusing subclassing: a case study in misusing inheritance

Suppose that Gary's Games has a new client who wants to use their game system framework to create World War II air battles. They need to take the basic **Board** base type, and extend it to support a 3-dimensional board to represent the sky.

LSP reveals hidden problems with your inheritance structure

This is the Board base type we developed

Board
width: int
height: int
tiles: Tile [*][*]
getTile(int, int): Tile
addUnit(Unit, int, int)
removeUnit(Unit, int, int)
removeUnits(int, int)
getUnits(int, int): List

The game designers subclassed Board and created a new type, 3DBoard.

3DBoard
zpos: int
3dTiles: Tile [*][*][*]
getTile(int, int, int): Tile
addUnit(Unit, int, int, int)
removeUnit(Unit, int, int, int)
removeUnits(int, int, int)
getUnits(int, int, int): List

Since Board3D requires (x,y,z) coordinates, it adds a bunch of new methods to support 3D coorindates.

When 3DBoard subclasses Board, it gets all of these methods, in addition to the new methods it defines.

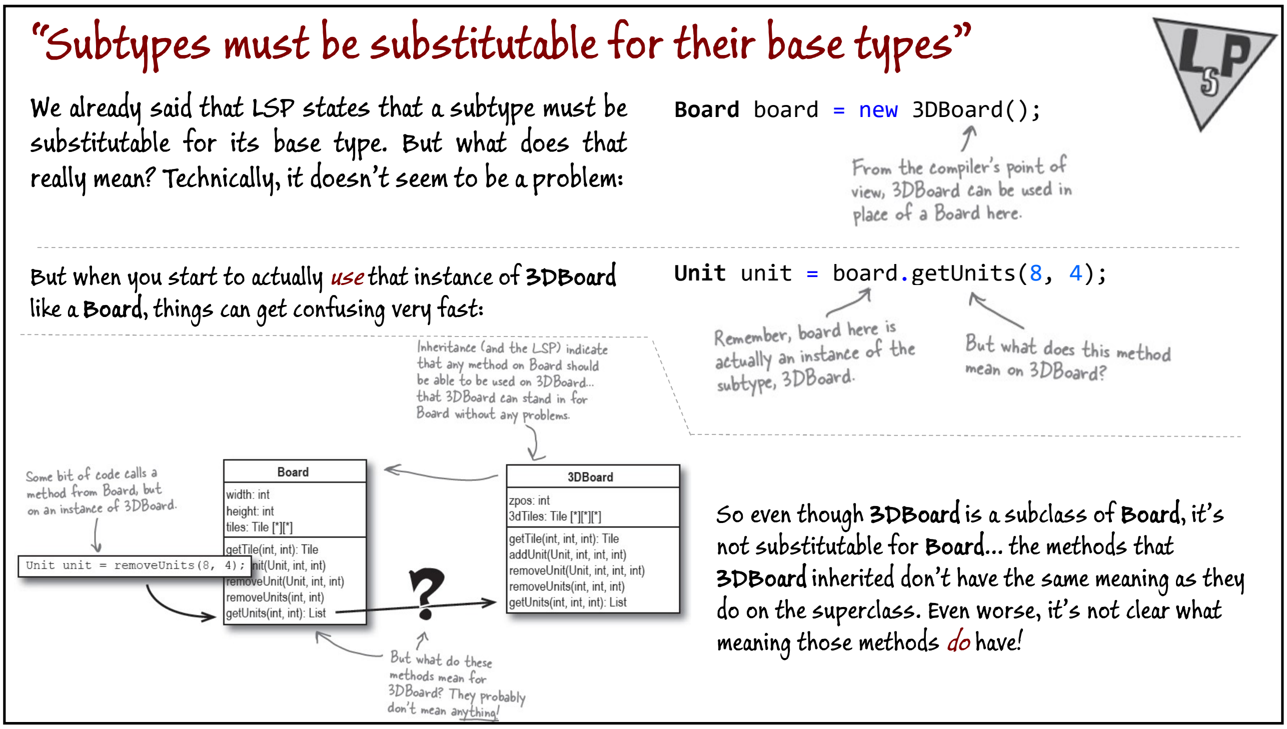
getTile()
addUnit()
removeUnit()
removeUnits()
getUnits()

All of these methods, that are inherited from Board don't have any meaning in a 3D context

But these are the methods that work with (x,y,z) coordinates... so what did we really gain from subclassing the Board type?

The 3DBoard class is not substitutable for Board, because none of the methods on Board work correctly in a 3D environment. Calling a method like getUnits(2, 5) doesn't make sense for 3DBoard.

19



"Subtypes must be substitutable for their base types"

We already said that LSP states that a subtype must be substitutable for its base type. But what does that really mean? Technically, it doesn't seem to be a problem:

```
Board board = new 3DBoard();
```

From the compiler's point of view, 3DBoard can be used in place of a Board here.

But when you start to actually *use* that instance of **3DBoard** like a **Board**, things can get confusing very fast:

```
Unit unit = board.getUnits(8, 4);
```

Remember, board here is actually an instance of the subtype, 3DBoard.

But what does this method mean on 3DBoard?

Inheritance (and the LSP) indicate that any method on Board should be able to be used on 3DBoard... that 3DBoard can stand in for Board without any problems.

Some bit of code calls a method from Board, but on an instance of 3DBoard.

```
Unit unit = removeUnits(8, 4);
```

Board
width: int
height: int
tiles: Tile [*][*]
getTile(int, int): Tile
nit(Unit, int, int)
removeUnit(Unit, int, int)
removeUnits(int, int)
getUnits(int, int): List

?

3DBoard
zpos: int
3dTiles: Tile [*][*][*]
getTile(int, int, int): Tile
addUnit(Unit, int, int, int)
removeUnit(Unit, int, int, int)
removeUnits(int, int, int)
getUnits(int, int, int): List

But what do these methods mean for 3DBoard? They probably don't mean anything!

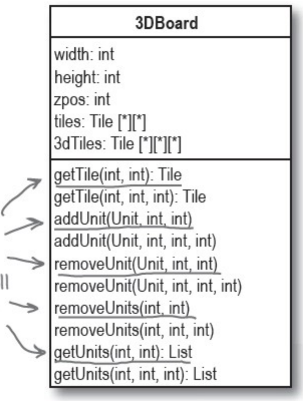
So even though **3DBoard** is a subclass of **Board**, it's not substitutable for **Board**... the methods that **3DBoard** inherited don't have the same meaning as they do on the superclass. Even worse, it's not clear what meaning those methods *do* have!

20

Violating the LSP makes for confusing code

It might seem like this isn't such a big deal, but code that violates LSP can be confusing, and a real nightmare to debug. Let's think a bit about someone who comes to use the badly designed **3DBoard** for the first time.
They probably start out by checking out the class's methods:

Hmm, I'm not sure which version of getTile() and addUnit() to use. Maybe those methods take an X- and Y-coordinate for the current board... I'm just not sure.


3DBoard

width: int
height: int
zpos: int
tiles: Tile [*][*]
3dTiles: Tile [*][*][*]

getTile(int, int): Tile
getTile(int, int, int): Tile
addUnit(Unit, int, int)
addUnit(Unit, int, int, int)
removeUnit(Unit, int, int)
removeUnit(Unit, int, int, int)
removeUnits(int, int)
removeUnits(int, int, int)
getUnits(int, int): List
getUnits(int, int, int): List

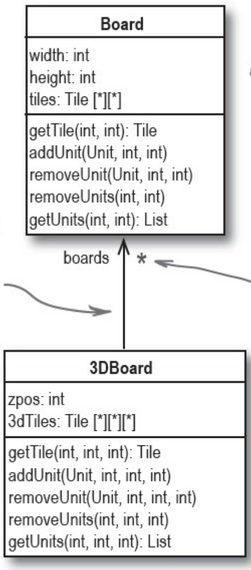
Even though some of these methods aren't defined on 3DBoard, they're all inherited from the base class, Board.

IT'S HARD TO UNDERSTAND CODE THAT MISUSES INHERITANCE.

Solving the 3DBoard problem without using inheritance

Now we've got to figure out what we *should* have done. Let's look at the **Board** and **3DBoard** classes again, and see how we can create a 3-dimensional board without using inheritance.

SO WHAT OPTIONS ARE THERE BESIDES INHERITANCE ?


Board

width: int
height: int
tiles: Tile [*][*]

getTile(int, int): Tile
addUnit(Unit, int, int)
removeUnit(Unit, int, int)
removeUnits(int, int)
getUnits(int, int): List

boards *

3DBoard

zpos: int
3dTiles: Tile [*][*][*]

getTile(int, int, int): Tile
addUnit(Unit, int, int, int)
removeUnit(Unit, int, int, int)
removeUnits(int, int, int)
getUnits(int, int, int): List

The Board class has functionality that 3DBoard needs, but it's not the base type for 3DBoard.

Instead of extension, we're using an association. So 3DBoard can use the behavior of Board, without having to extend from it and violate the LSP.

3DBoard can store an array of Board objects, and end up with a 3D collection of boards.

This is a form of delegation. The 3DBoard class delegates a lot of its functionality to the individual Board instances.

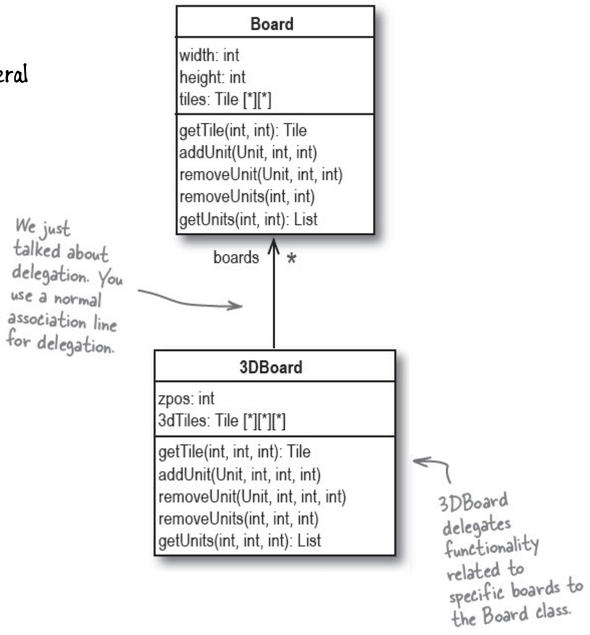
The 3DBoard methods use the zpos coordinate to figure out which Board instance in the array to use, and then delegates the (x,y) coords to that Board's functions.

These methods look a lot like the methods in Board, but they need to use the functionality in Board, rather than extend it. So inheritance isn't a good option here.

Delegate functionality to another class

You've already seen that delegation is when one class hands off the task of doing something to another class. It's also just one of several alternatives to inheritance.

Delegation *is when you hand over the responsibility for a particular task to another class or method.*



Board
width: int
height: int
tiles: Tile [*][*]
getTile(int, int): Tile
addUnit(Unit, int, int)
removeUnit(Unit, int, int)
removeUnits(int, int)
getUnits(int, int): List

boards *

3DBoard
zpos: int
3dTiles: Tile [*][*][*]
getTile(int, int, int): Tile
addUnit(Unit, int, int, int)
removeUnit(Unit, int, int, int)
removeUnits(int, int, int)
getUnits(int, int, int): List

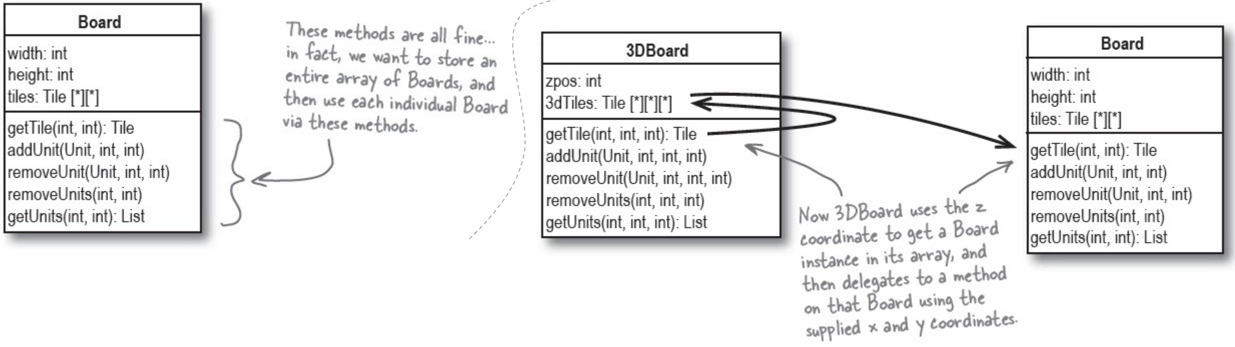
We just talked about delegation. You use a normal association line for delegation.

3DBoard delegates functionality related to specific boards to the Board class.

When to use delegation

Delegation is best used when you want to use another class's functionality, as is, without changing that behavior at all. In the case of **3DBoard**, we wanted to use the various methods in the **Board** class:

Since we don't want to change the existing behavior, but we do want to use it, we can simply create a delegation relationship between **3DBoard** and **Board**. **3DBoard** stores multiple instances of **Board** objects, and delegates handling each individual board-related task.



Board
width: int
height: int
tiles: Tile [*][*]
getTile(int, int): Tile
addUnit(Unit, int, int)
removeUnit(Unit, int, int)
removeUnits(int, int)
getUnits(int, int): List

These methods are all fine... in fact, we want to store an entire array of Boards, and then use each individual Board via these methods.

3DBoard
zpos: int
3dTiles: Tile [*][*][*]
getTile(int, int, int): Tile
addUnit(Unit, int, int, int)
removeUnit(Unit, int, int, int)
removeUnits(int, int, int)
getUnits(int, int, int): List

Board
width: int
height: int
tiles: Tile [*][*]
getTile(int, int): Tile
addUnit(Unit, int, int)
removeUnit(Unit, int, int)
removeUnits(int, int)
getUnits(int, int): List

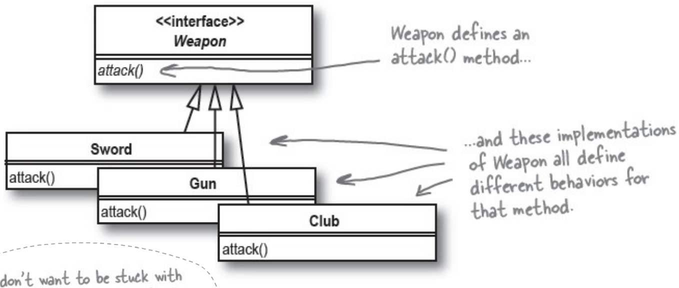
Now 3DBoard uses the z coordinate to get a Board instance in its array, and then delegates to a method on that Board using the supplied x and y coordinates.

! IF YOU NEED TO USE FUNCTIONALITY IN ANOTHER CLASS, BUT YOU DON'T WANT TO CHANGE THAT FUNCTIONALITY, CONSIDER USING DELEGATION INSTEAD OF INHERITANCE.

Use composition to assemble behaviors from other classes

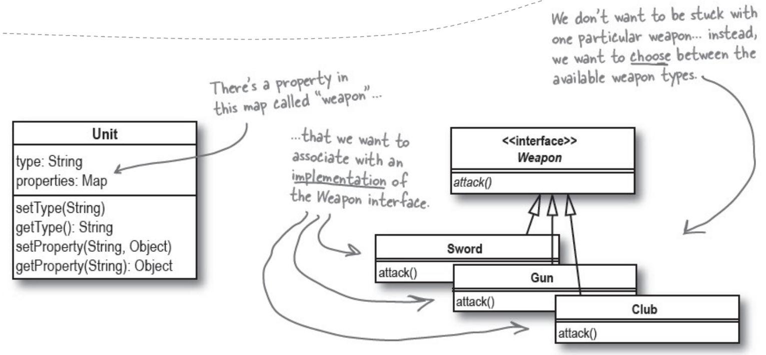
Sometimes delegation isn't quite what you need; in delegation, the behavior of the object you're delegating behavior to never changes. **3DBoard** *always* uses instances of **Board**, and the behavior of the **Board** methods *always* stay the same.

But in some cases, you need to have more than one single behavior to choose from. For example, suppose we wanted to develop a **Weapon** interface, and then create several implementations of that interface that all behave differently:



Weapon defines an attack() method...

...and these implementations of Weapon all define different behaviors for that method.



We don't want to be stuck with one particular weapon... instead, we want to choose between the available weapon types.

There's a property in this map called "weapon"...

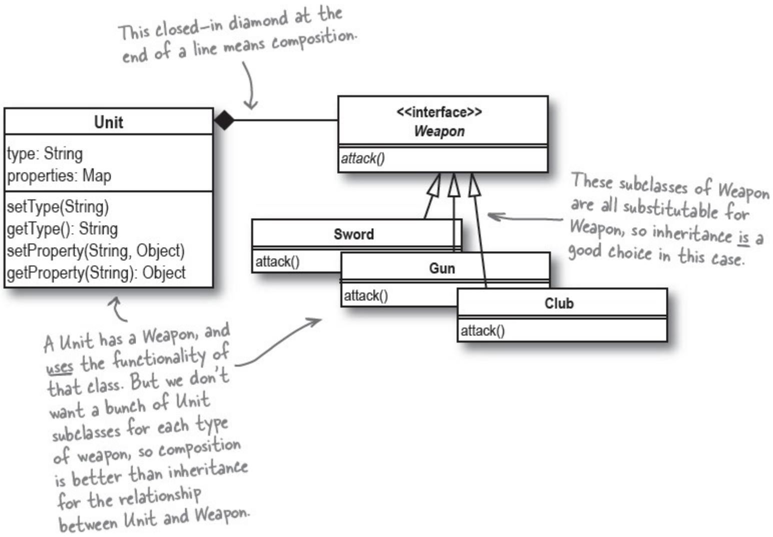
...that we want to associate with an implementation of the Weapon interface.

Now we need to use the behavior from these classes in our **Unit** class. One of the properties in our **properties Map** will be "weapon", and the value for that property needs to be an implementation of the **Weapon** class. But a **Unit** might change weapons, so we don't want to tie the weapon property to a specific implementation of **Weapon**; instead, we just want each **Unit** to be able to reference a **Weapon**, regardless of which implementation of **Weapon** we want to use.

25

When to use composition

When we reference a whole family of behaviors like in the Unit class, we're using **composition**. The **Unit**'s weapons property is *composed* of a particular **Weapon** implementation's behavior. We can show this in UML like this:



This closed-in diamond at the end of a line means composition.

These subclasses of Weapon are all substitutable for Weapon, so inheritance is a good choice in this case.

A Unit has a Weapon, and uses the functionality of that class. But we don't want a bunch of Unit subclasses for each type of weapon, so composition is better than inheritance for the relationship between Unit and Weapon.

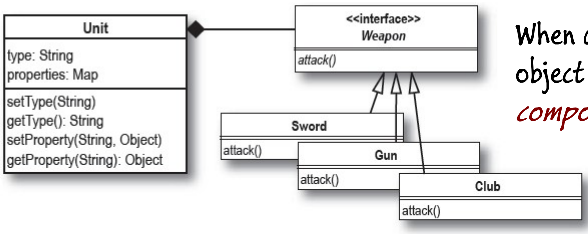
Composition is most powerful when you want to use behavior defined in an interface, and then choose from a variety of implementations of that interface, at both compile time and run time.

Composition allows you to use behavior from a family of other classes, and to change that behavior at runtime.

Pizza is actually a great example of composition: it's composed of different ingredients, but you can swap out different ingredients without affecting the overall pizza slice.

26

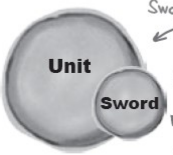
When the pizza is gone, so are the ingredients...



When an object is composed of other objects, and the owning object is destroyed, *the objects that are part of the composition go away, too.*

Suppose we create a new **Unit**, and assign its weapon property to an instance of **Sword**:

This Unit is composed with an instance of Sword.



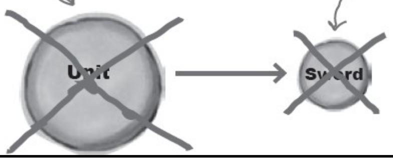
This Sword object does not exist outside of the context of this particular Unit.

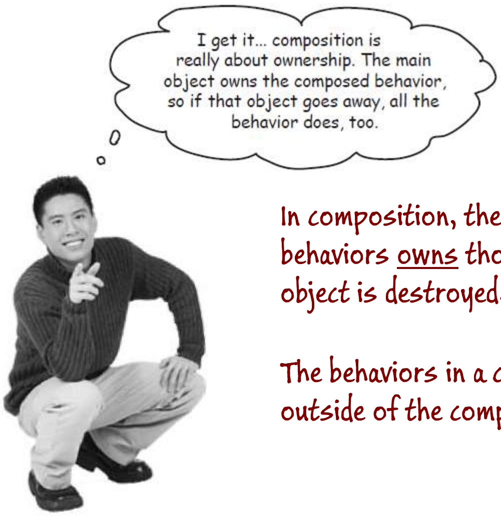
```
Unit pirate = new Unit();
pirate.setProperty("weapon", new Sword());
```

What happens if this **Unit** is destroyed? Obviously, the **pirate** variable is trashed, but the instance of **Sword** referenced by pirate is also thrown away. It doesn't exist *outside of* the **pirate** object.

If you get rid of the pirate Unit object...

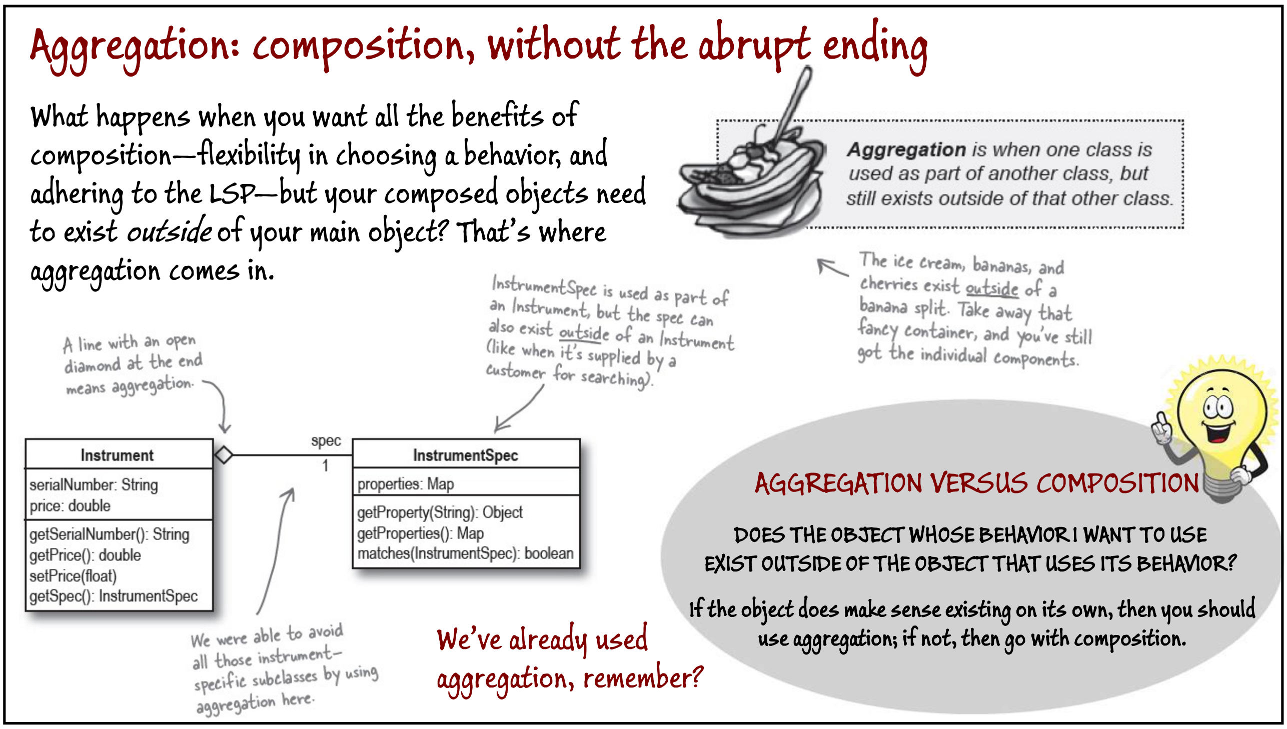
...then you're automatically getting rid of the Sword object associated with pirate, too.




I get it... composition is really about ownership. The main object owns the composed behavior, so if that object goes away, all the behavior does, too.

In composition, the object composed of other behaviors owns those behaviors. When the object is destroyed, so are all of its behaviors.

The behaviors in a composition do not exist outside of the composition itself.



Aggregation: composition, without the abrupt ending

What happens when you want all the benefits of composition—flexibility in choosing a behavior, and adhering to the LSP—but your composed objects need to exist *outside* of your main object? That's where aggregation comes in.

Aggregation is when one class is used as part of another class, but still exists outside of that other class.

The ice cream, bananas, and cherries exist outside of a banana split. Take away that fancy container, and you've still got the individual components.

A line with an open diamond at the end means aggregation.

InstrumentSpec is used as part of an Instrument, but the spec can also exist outside of an Instrument (like when it's supplied by a customer for searching).

We were able to avoid all those instrument-specific subclasses by using aggregation here.

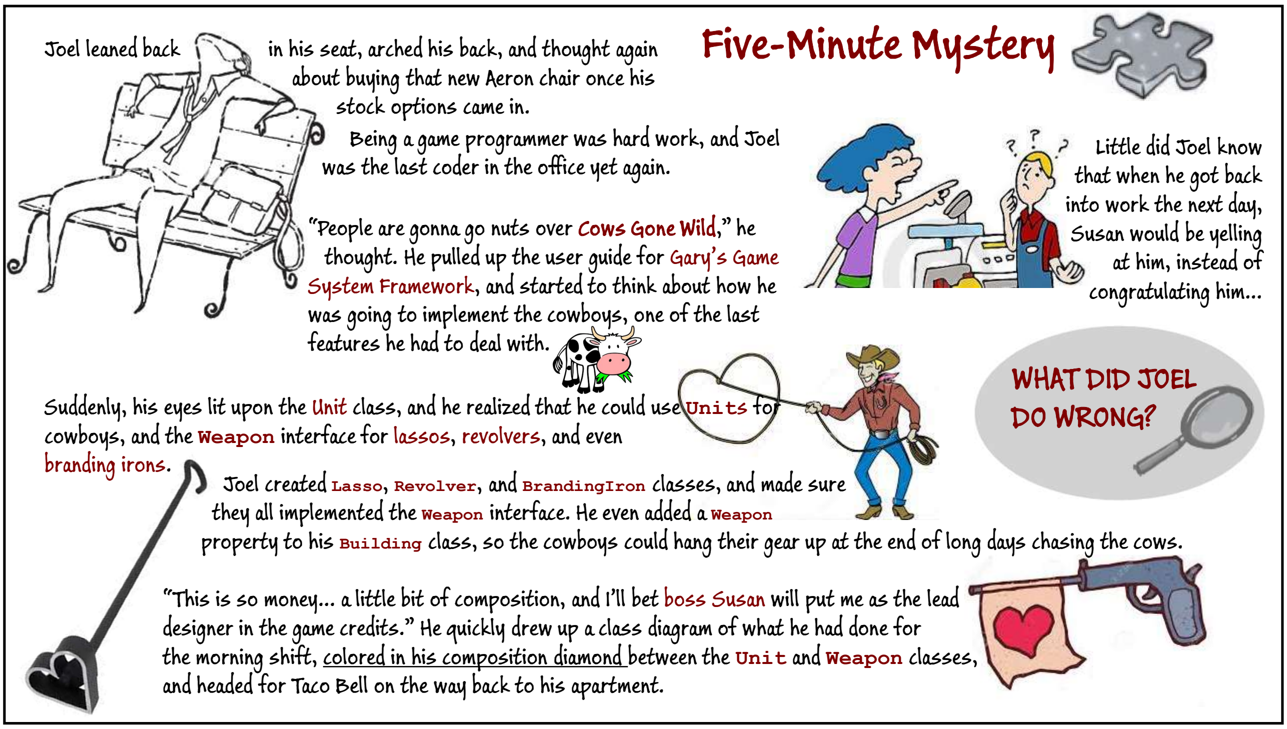
We've already used aggregation, remember?

AGGREGATION VERSUS COMPOSITION

DOES THE OBJECT WHOSE BEHAVIOR I WANT TO USE EXIST OUTSIDE OF THE OBJECT THAT USES ITS BEHAVIOR?

If the object does make sense existing on its own, then you should use aggregation; if not, then go with composition.

29



Five-Minute Mystery

Joel leaned back in his seat, arched his back, and thought again about buying that new Aeron chair once his stock options came in.

Being a game programmer was hard work, and Joel was the last coder in the office yet again.

"People are gonna go nuts over Cows Gone Wild," he thought. He pulled up the user guide for Gary's Game System Framework, and started to think about how he was going to implement the cowboys, one of the last features he had to deal with.

Little did Joel know that when he got back into work the next day, Susan would be yelling at him, instead of congratulating him...

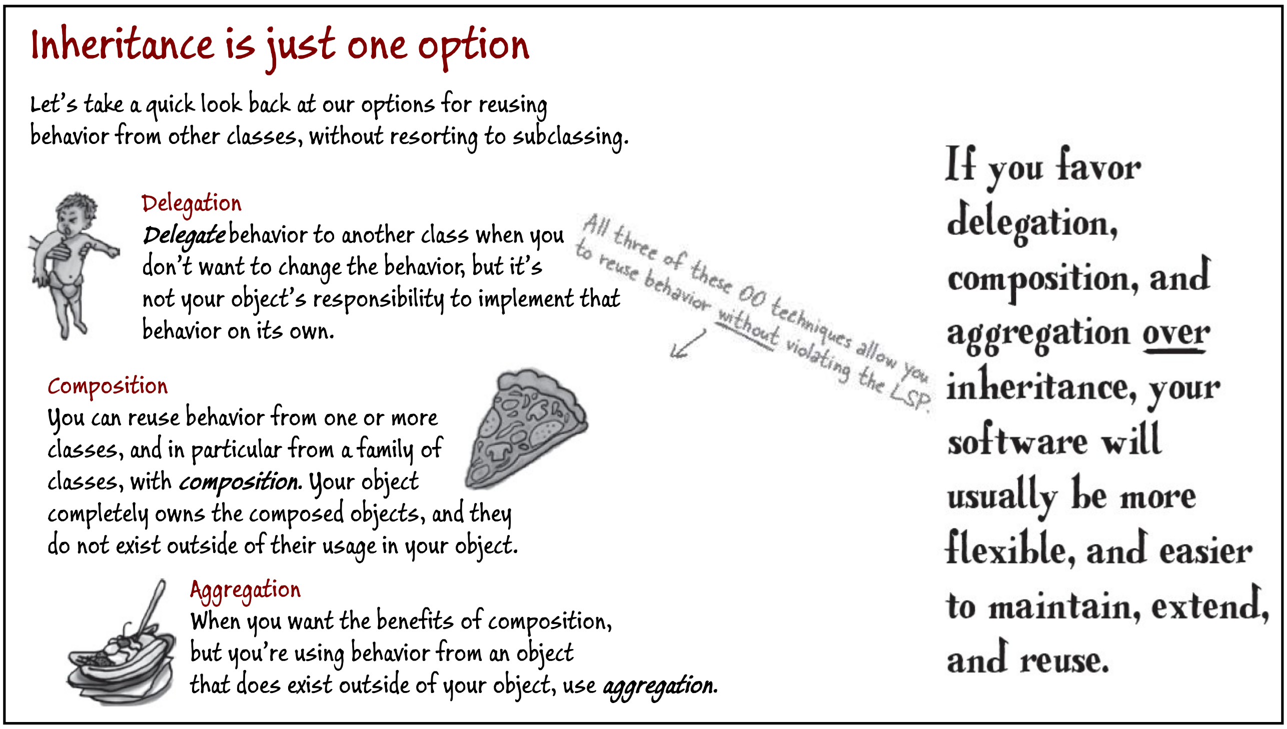
Suddenly, his eyes lit upon the Unit class, and he realized that he could use `Units` for cowboys, and the `Weapon` interface for lassos, revolvers, and even branding irons.

Joel created `Lasso`, `Revolver`, and `BrandingIron` classes, and made sure they all implemented the `Weapon` interface. He even added a `Weapon` property to his `Building` class, so the cowboys could hang their gear up at the end of long days chasing the cows.

"This is so money... a little bit of composition, and I'll bet boss Susan will put me as the lead designer in the game credits." He quickly drew up a class diagram of what he had done for the morning shift, colored in his composition diamond between the `Unit` and `Weapon` classes, and headed for Taco Bell on the way back to his apartment.

WHAT DID JOEL DO WRONG?

30



Inheritance is just one option

Let's take a quick look back at our options for reusing behavior from other classes, without resorting to subclassing.

Delegation

Delegate behavior to another class when you don't want to change the behavior, but it's not your object's responsibility to implement that behavior on its own.

All three of these OO techniques allow you to reuse behavior without violating the LSP.

Composition

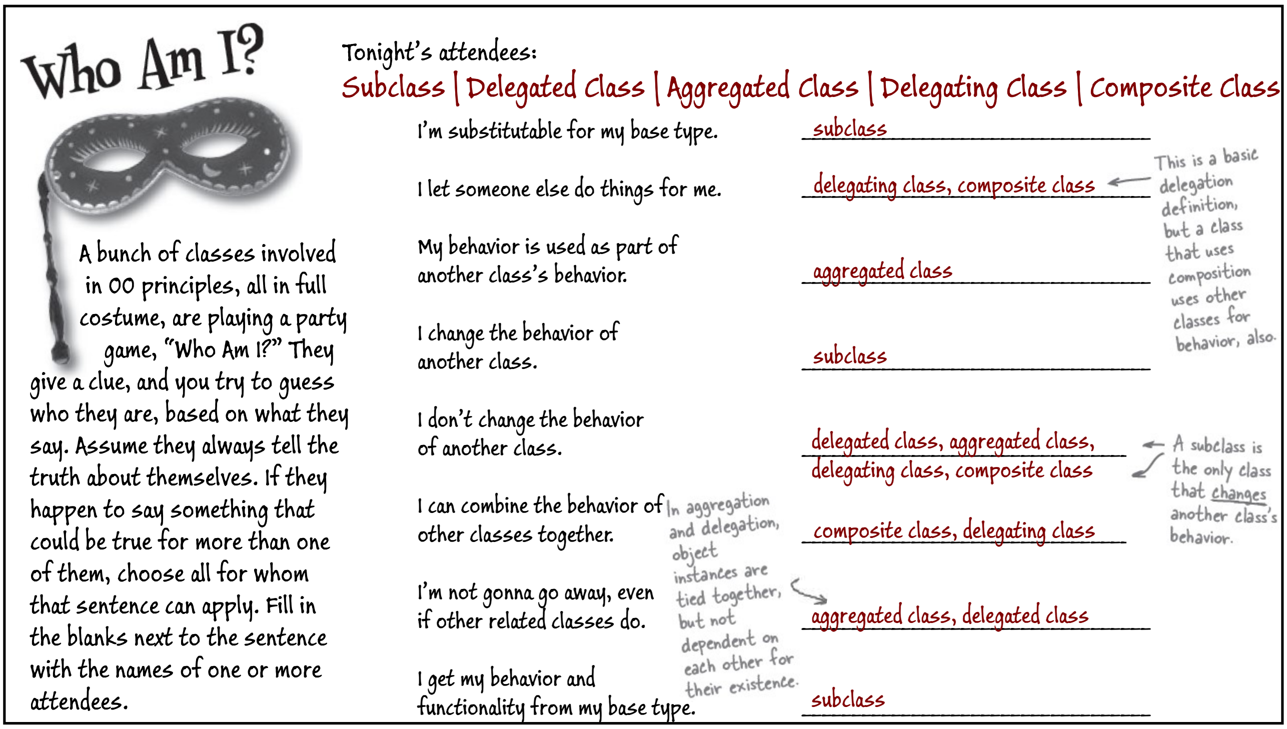
You can reuse behavior from one or more classes, and in particular from a family of classes, with ***composition***. Your object completely owns the composed objects, and they do not exist outside of their usage in your object.

Aggregation

When you want the benefits of composition, but you're using behavior from an object that does exist outside of your object, use ***aggregation***.

If you favor delegation, composition, and aggregation over inheritance, your software will usually be more flexible, and easier to maintain, extend, and reuse.

31



Who Am I?

A bunch of classes involved in OO principles, all in full costume, are playing a party game, "Who Am I?" They give a clue, and you try to guess who they are, based on what they say. Assume they always tell the truth about themselves. If they happen to say something that could be true for more than one of them, choose all for whom that sentence can apply. Fill in the blanks next to the sentence with the names of one or more attendees.

Tonight's attendees:

Subclass | Delegated Class | Aggregated Class | Delegating Class | Composite Class

I'm substitutable for my base type. subclass

I let someone else do things for me. delegating class, composite class

This is a basic delegation definition, but a class that uses composition uses other classes for behavior, also.

My behavior is used as part of another class's behavior. aggregated class

I change the behavior of another class. subclass

I don't change the behavior of another class. delegated class, aggregated class, delegating class, composite class

A subclass is the only class that changes another class's behavior.

I can combine the behavior of other classes together. composite class, delegating class

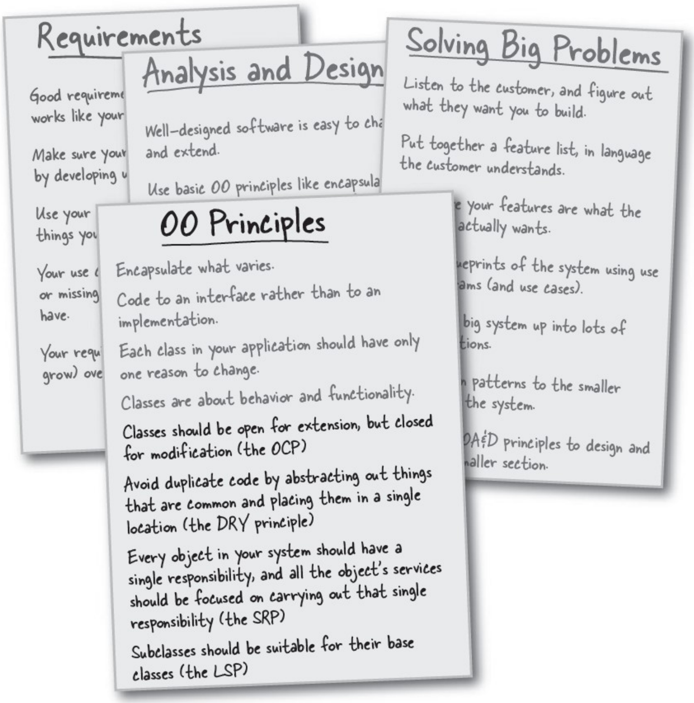
In aggregation and delegation, object instances are tied together, but not dependent on each other for their existence.

I'm not gonna go away, even if other related classes do. aggregated class, delegated class

I get my behavior and functionality from my base type. subclass

32

Tools for your toolbox



Requirements

Good requireme
works like your

Make sure your
by developing u

Use your
things you

Your use c
or missing
have.

Your requ
grow) ove

Analysis and Design

Well-designed software is easy to cha
and extend.

Use basic OO principles like encapsula

Solving Big Problems

Listen to the customer, and figure out what they want you to build.

Put together a feature list, in language the customer understands.

e your features are what the
actually wants.

eprints of the system using use
ams (and use cases).

big system up into lots of
tions.

n patterns to the smaller
the system.

OA&D principles to design and
aller section.

OO Principles

Encapsulate what varies.

Code to an interface rather than to an implementation.

Each class in your application should have only one reason to change.

Classes are about behavior and functionality.

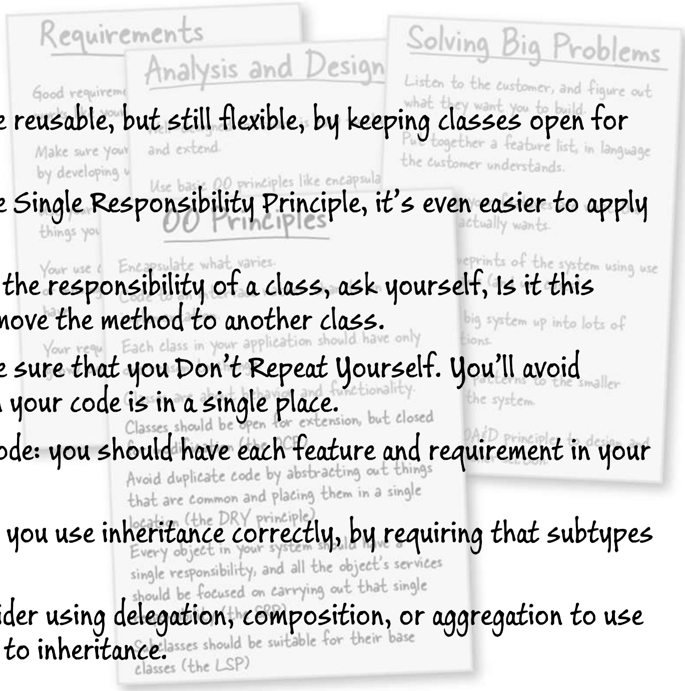
Classes should be open for extension, but closed for modification (the OCP)

Avoid duplicate code by abstracting out things that are common and placing them in a single location (the DRY 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 (the SRP)

Subclasses should be suitable for their base classes (the LSP)

Tools for your toolbox



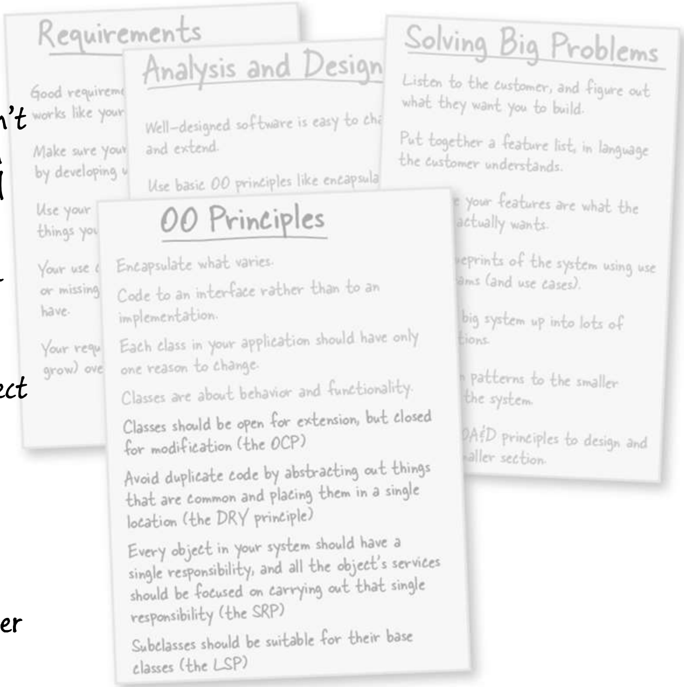
Bullet Points

- The Open-Closed Principle keeps your software reusable, but still flexible, by keeping classes open for extension, but closed for modification.
- With classes doing one single thing through the Single Responsibility Principle, it's even easier to apply the OCP to your code.
- When you're trying to determine if a method is the responsibility of a class, ask yourself, Is it this class's job to do this particular thing? If not, move the method to another class.
- Once you have your OO code nearly complete, be sure that you Don't Repeat Yourself. You'll avoid duplicate code, and ensure that each behavior in your code is in a single place.
- DRY applies to requirements as well as your code: you should have each feature and requirement in your software implemented in a single place.
- The Liskov Substitution Principle ensures that you use inheritance correctly, by requiring that subtypes be substitutable for their base types.
- When you find code that violates the LSP, consider using delegation, composition, or aggregation to use behavior from other classes without resorting to inheritance.

Tools for your toolbox

Bullet Points

- If you need behavior from another class but don't need to change or modify that behavior, you can simply delegate to that class to use the desired behavior.
- Composition lets you choose a behavior from a family of behaviors, often via several implementations of an interface.
- When you use composition, the composing object owns the behaviors it uses, and they stop existing as soon as the composing object does.
- Aggregation allows you to use behaviors from another class without limiting the lifetime to those behaviors.
- Aggregated behaviors continue to exist even after the aggregating object is destroyed.



Requirements

Good requireme
works like your

Make sure your
by developing u

Use your
things you

Your use c
or missing
have.

Your requ
grow) ove

Analysis and Design

Well-designed software is easy to cha
and extend.

Use basic OO principles like encapsula

Solving Big Problems

Listen to the customer, and figure out what they want you to build.

Put together a feature list, in language the customer understands.

e your features are what the
actually wants.

eprints of the system using use
ams (and use cases).

big system up into lots of
tions.

patterns to the smaller
the system.

OA&D principles to design and
aller section.

OO Principles

Encapsulate what varies.

Code to an interface rather than to an implementation.

Each class in your application should have only one reason to change.

Classes are about behavior and functionality.

Classes should be open for extension, but closed for modification (the OCP)

Avoid duplicate code by abstracting out things that are common and placing them in a single location (the DRY 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 (the SRP)

Subclasses should be suitable for their base classes (the LSP)