CSE203 Object-Oriented Analysis and Design

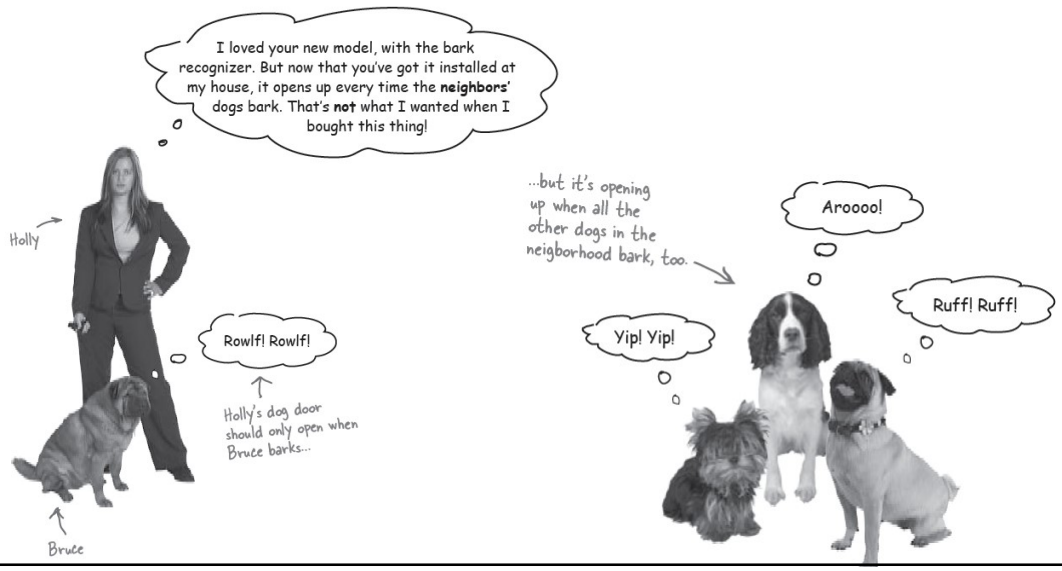
04. Analysis

Taking your Software into the Real World

1

One dog, two dog, three dog, four...

as more doors get installed, complaints have started coming in:



2

2

Your software has a context

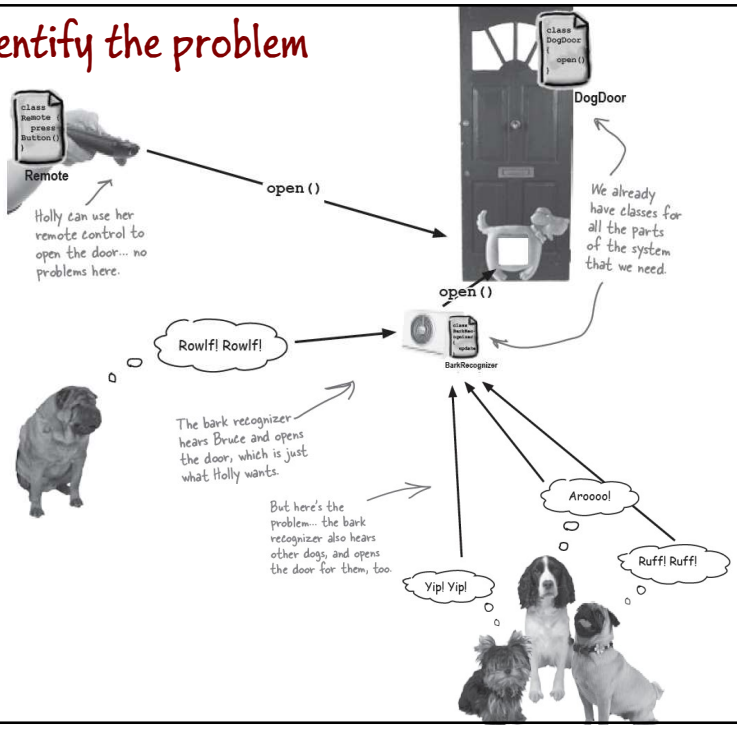
We haven't thought about the context that our software is running in. We've been thinking about our software like this:

But our software has to **work in the real world**, not just in a perfect world. That means we have to think about our software in a different context:

Analysis helps you ensure your system works in a **real-world context.**

3

Identify the problem



4

4

Plan a solution

Bruce has taken Fido's place... better update your diagram a bit.

1. Bruce barks to be let out

2. The bark recognizer "hears" a bark

The bark recognizer is part of the main path, and it's letting in all dogs, not just the owner's dog.

3. The bark recognizer sends a request to the door to open

4. The dog door opens

5. Bruce goes outside

The remote is part of the alternate path in the latest version of the dog door.

2.1 Holly hears Bruce barking

3.1 Holly presses the button on the remote control

Sharpen your pencil

What's wrong with this diagram?

Sharpen your pencil

answers

1. Bruce barks to be let out

2. The bark recognizer "hears" a bark

The bark recognizer hears all dogs, which is OK... but it's the next step that's causing a problem.

~~3. The bark recognizer sends a request to the door to open~~

3. If it's Bruce barking, send a request to the door to open

In step 3, the bark recognizer needs to evaluate the bark it hears and see if it's Bruce, or some other dog.

4. The dog door opens

5. Bruce goes outside

2.1 Holly hears Bruce barking

3.1 Holly presses the button on the remote control

Update your use case

Since we've changed our dog door diagram, we need to go back to the dog door use case, and update it with the new steps we've figured out.

We've removed all the references to specific owners and dogs, so now this use case will work for all of Doug's customers.

The Ultimate Dog Door version 3.0
What the Door Does

Main Path	Alternate Paths
1. **The owner's dog** barks to be let out.	
2. The bark recognizer "hears" a bark.	2.1. **The owner** hears her dog barking.
3. **If it's the owner's dog barking, the bark recognizer sends a request to the door to open.**	3.1. The owner presses the button on the remote control.
4. The dog door opens.	
5. The owner's dog goes outside.	
6. The owner's dog does his business.	
6.1. The door shuts automatically.	
6.2. The owner's dog barks to be let back inside.	
6.3. The bark recognizer "hears" a bark (again).	6.3.1. The owner hears her dog barking (again).
6.4. **If it's the owner's dog barking, the bark recognizer sends a request to the door to open.**	6.4.1. The owner presses the button on the remote control.
6.5. The dog door opens (again).	
7. The owner's dog goes back inside.	
8. The door shuts automatically.	

Bye bye, Fido. Let's use "the owner's dog" from now on.

Instead of Todd and Gina, or Holly, let's just use "The owner."

Here is the updated step that deals with only allowing the owner's dog in and out the door.

Don't forget to change this substep, too.

7

7

Don't we need to store the owner's dog's bark in our dog door? Otherwise, we won't have anything to compare to the bark that our bark recognizer gives us.

Do we need a new use case to store the owner's dog's bark?

Our analysis has made us realize we need to make some changes to our use case—and those changes mean that we need to make some additions to our system, too.

Sharpen your pencil

Add a new use case to store a bark.

You need a use case to store the owner's dog's bark; let's store the sound of the dog in the dog door itself. Use the use case template below to write a new use case for this task.

The Ultimate Dog Door version 3.0
Storing a dog bark

1. _____
2. _____

Since this is our second use case, let's label it according to what it describes.

You should need only two steps for this use case, and there aren't any alternate paths to worry about.

8

Sharpen your pencil

answers

We don't need to know the exact details of this, since it's a hardware issue.

The Ultimate Dog Door, version 3.0
Storing a dog bark

1. The owner's dog barks "into" the dog door.
2. The dog door stores the owner's dog's bark.

This is what we need to do... add a method to DogDoor to store the owner's dog's bark.

there are no Dumb Questions

Q: Do we really need a whole new use case for storing the owner's dog's bark?

A: Yes. Each use case should detail one particular user goal. The user goal for our original use case was to get a dog outside and back in without using the bathroom in the house, and the user goal of this new use case is to store a dog's bark. Since those aren't the same user goal, you need two different use cases.

Q: Is this really the result of good analysis, or just something we should have thought about in the last two weeks?

A: Probably a bit of both. Sure, we probably should have figured out that we needed to store the owner's dog's bark much earlier, but that's what analysis is really about: making sure that you didn't forget anything that will help your software work in a real world context.

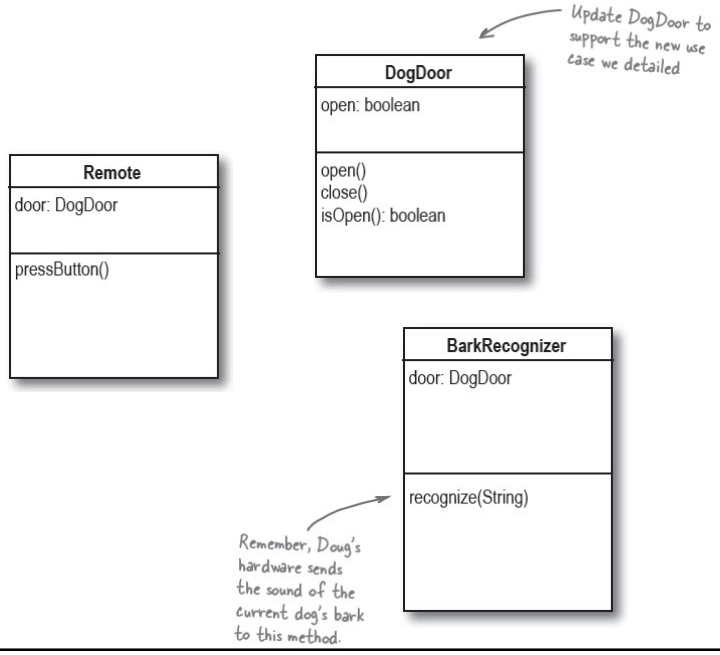
9

9

Design Puzzle

Your task

1. Add any new objects you think you might need for the new dog door.
2. Add a new method to the **DogDoor** class that will store a dog's bark, and another new method to allow other classes to access the bark.
3. If you need to make changes to any other classes or methods, write in those changes in the class diagram below.
4. Add notes to the class diagram to remind you what any tricky attributes or operations are used for, and how they should work.



10

10

A tale of two coders

Doug's offered the programmer with the best design a sparkling new Apple MacBook Pro!

17 inches of raw Apple and Intel power.

Randy:
simple is best, right?

Randy

Bark sounds are just Strings, so I'll store a String for the owner's dog's bark in DogDoor, and add a couple of simple methods. Piece of cake!

Randy adds an allowedBark variable to his DogDoor class.

```
public class DogDoor {
  private boolean open;
  private String allowedBark;

  public DogDoor() {
    open = false;
  }

  public void setAllowedBark(String bark) {
    this.allowedBark = bark;
  }

  public String getAllowedBark() {
    return allowedBark;
  }
  // etc
}
```

This handles setting the bark, which was what our new use case focused on.

Other classes can get the owner's dog's bark with this method.

DogDoor
open: boolean **allowedBark: String**
open() close() isOpen(): boolean **setAllowedBark(String)** **getAllowedBark(): String**

11

Sharpen your pencil

Your job is to write the code for Sam's Bark class based on his class diagram.

```
public class _____ {
  private _____ _____;

  public _____(_____ _____) {
    this._____ = _____;
  }

  public _____ _____() {
    _____ _____;
  }

  _____ _____ _____(_____ _____) {
    if (_____ instanceof _____) {
      Bark otherBark = (_____)_____;
      if (this._____.equalsIgnoreCase(_____._____)) {
        return _____;
      }
    }
    return _____;
  }
}
```

I've got the power of objects!

Sam:
object lover extraordinaire

Sam

Sam's new Bark class.

Bark
sound: String
getSound(): String equals(Object bark): boolean

Sam plans to store the sound of a dog's Bark as a String in his new Bark class...

...a method to return the sound of the Bark...

...and an equals() method to allow other objects to compare two Bark instances.

12

12

Sharpen your pencil answers

```
public class Bark {
  private String sound;

  public Bark(String sound) {
    this.sound = sound;
  }

  public String getSound() {
    return sound;
  }

  public boolean equals(Object bark) {
    if (bark instanceof Bark) {
      Bark otherBark = (Bark) bark;
      if (this.sound.equalsIgnoreCase(otherBark.sound)) {
        return true;
      }
    }
    return false;
  }
}
```

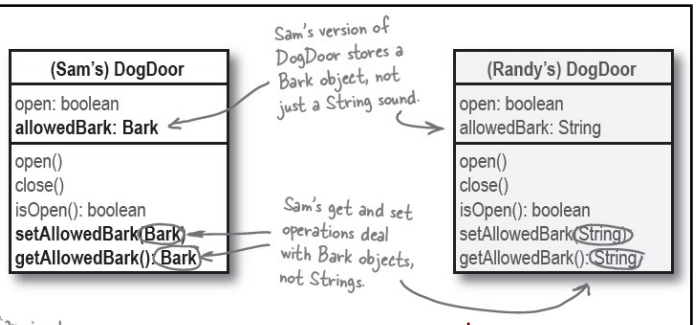
Just like Randy did, Sam is using a String to store the actual bark sound...

...but he's wrapped the sound up in a Bark object.

Sam is planning on other classes delegating Bark comparison to the Bark class's equals() method.

This method makes sure it has another Bark object to compare itself against...

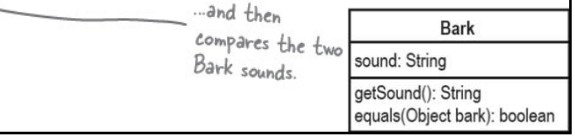
...and then compares the two Bark sounds.



Sam's version of DogDoor stores a Bark object, not just a String sound.

Sam's get and set operations deal with Bark objects, not Strings.

Sam: updating the DogDoor class



Comparing barks

Randy: I'll just compare two strings

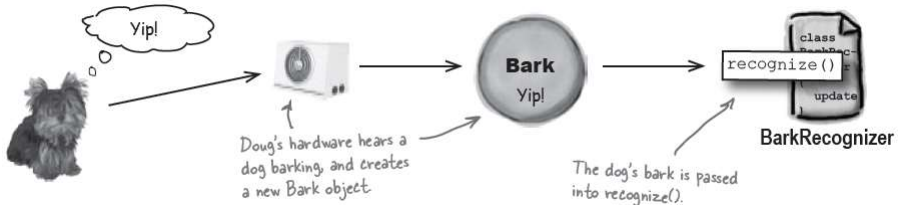
```
public class BarkRecognizer {
  public void recognize(String bark) {
    System.out.println(" BarkRecognizer: "
      "Heard a '" + bark + "'");
    if (door.getAllowedBark().equals(bark)) {
      door.open();
    } else {
      System.out.println("This dog is " +
        "not allowed.");
    }
  }
  // etc
}
```

Sam: I'll delegate bark comparison

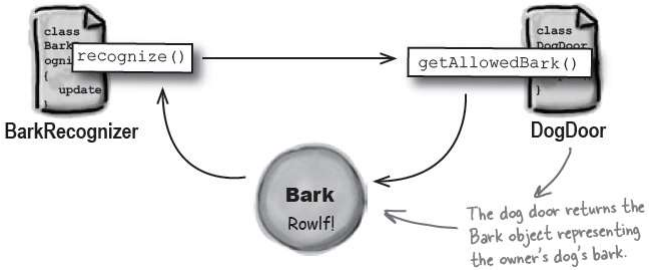
```
public class BarkRecognizer {
  public void recognize(Bark bark) {
    System.out.println(" BarkRecognizer: "
      "Heard a '" + bark.getSound() + "'");
    if (door.getAllowedBark().equals(bark)) {
      door.open();
    } else {
      System.out.println("This dog is " +
        "not allowed.");
    }
  }
  // etc
}
```

Delegation in Sam's dog door: an in-depth look

1. The BarkRecognizer gets a Bark to evaluate.



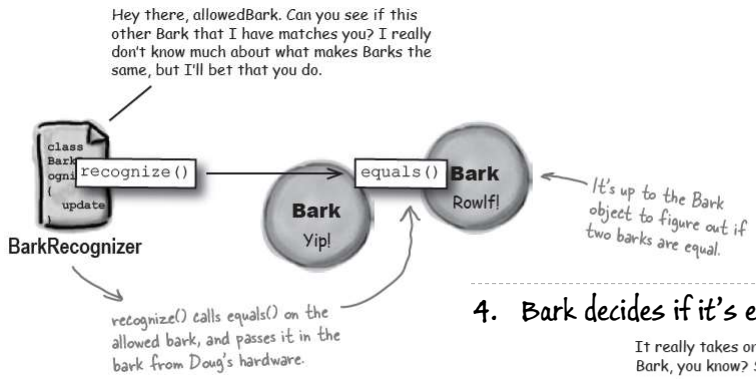
2. BarkRecognizer gets the owner's dog's bark from DogDoor



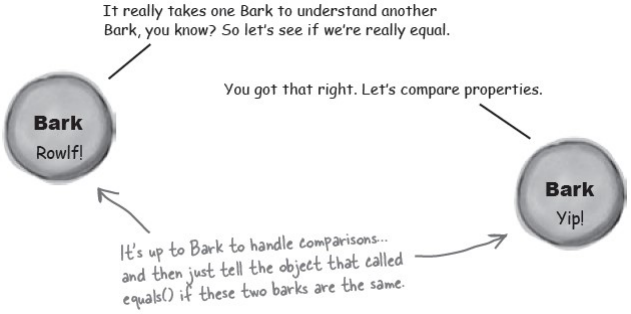
15

15

3. BarkRecognizer delegates bark comparison to Bark



4. Bark decides if it's equal to the bark from Doug's hardware



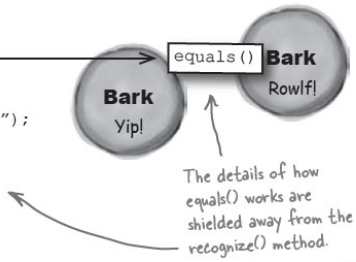
16

16

The power of loosely coupled applications

- Loosely coupled means objects are independent of each other
- changes to one object don't require you to make a bunch of changes to other objects.

```
public void recognize(Bark bark) {
  System.out.println("   BarkRecognizer: "
    "Heard a '" + bark.getSound() + "'");
  if (door.getAllowedBark().equals(bark)) {
    door.open();
  } else {
    System.out.println("This dog is not allowed.");
  }
}
```

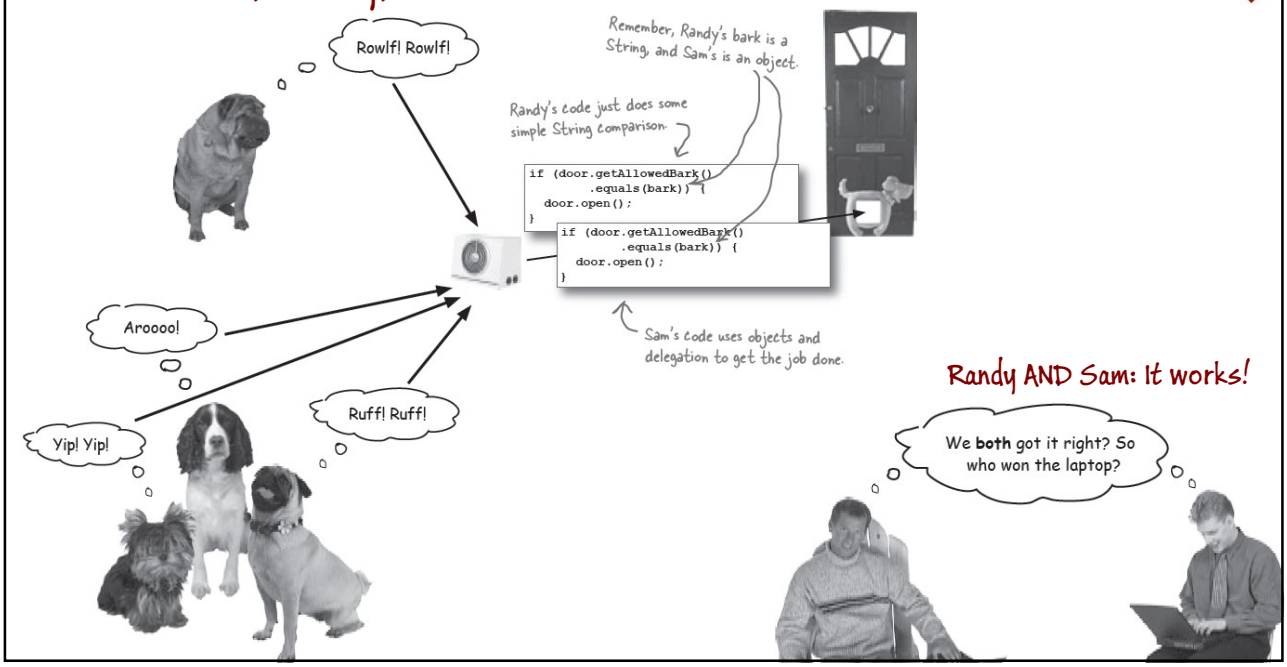


- Now suppose that we started storing the sound of a dog barking as a WAV file in **Bark**.
- We'd need to change the **equals()** method in the **Bark** class to do a more advanced comparison.
- But, since the **recognize()** method delegates bark comparison, no code in **BarkRecognizer** would have to change.

Delegation shields your objects from implementation changes to other objects in your software.

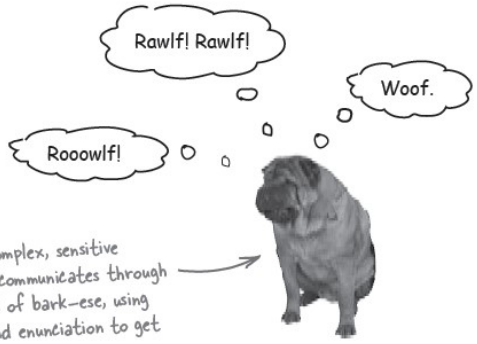
17

Back to Sam, Randy, and the contest...



18

Maria won the MacBook Pro!

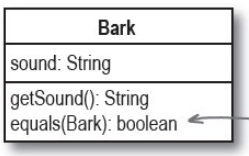


Maria: Umm, guys, I don't mean to interrupt, but I'm not sure either one of your dog doors *really* worked.

...what if Bruce were to make a different sound? Like "Woof" or "Ruff"?

19

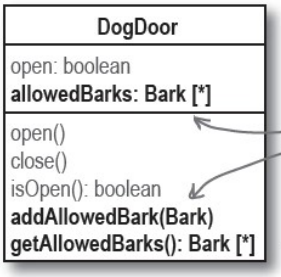
So what did Maria do differently?



Maria started out a lot like Sam did. She created a **Bark** object to represent the bark of a dog.

Maria knew she'd need delegation via the equals() method, just as Sam did.

...the dog door should store *multiple* **Bark** objects.



Here's where Maria really went down a different path. She decided that the dog door should store more than just one bark, since the owner's dog can bark in different ways.

UML Up Close

We've added something new to our class diagrams:

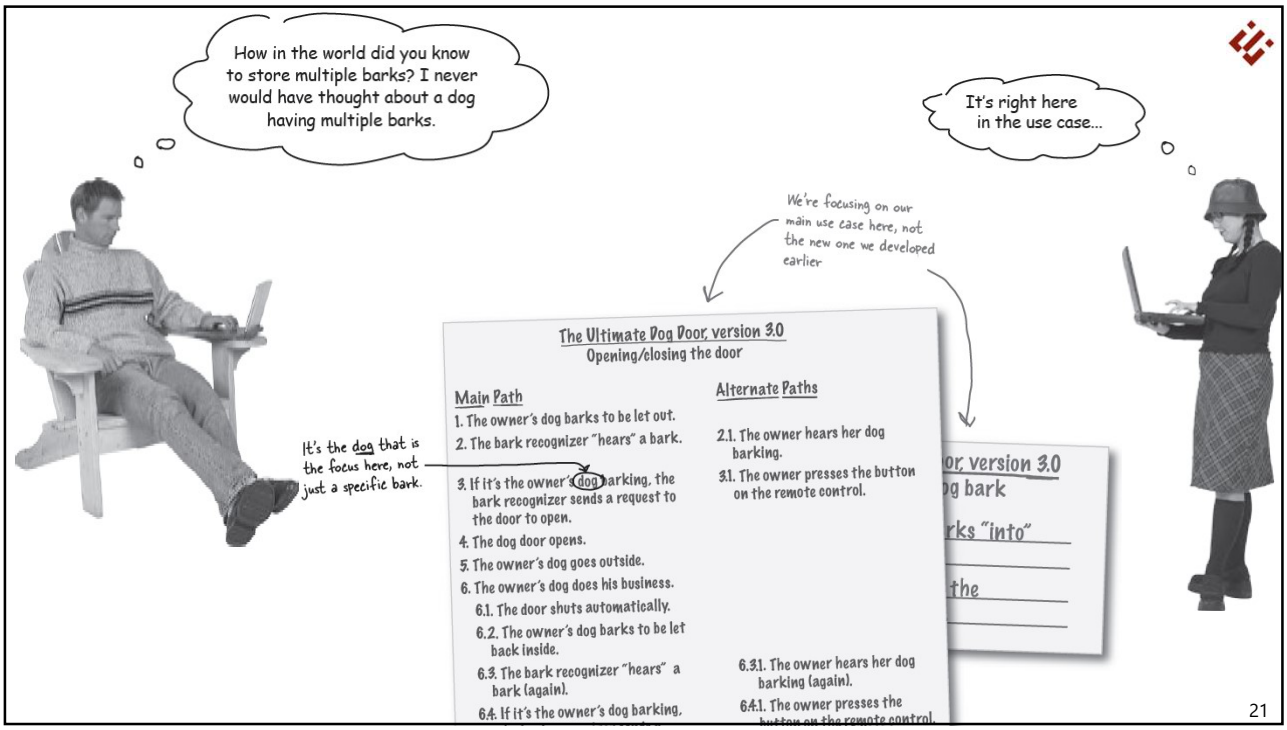
allowedBarks: Bark [*]

Anytime you see brackets, it indicates the **multiplicity** of an attribute: how many of a certain type that the attribute can hold.

The type of the allowedBarks attribute is Bark.

And this asterisk means that allowedBarks can hold an unlimited number of Bark objects.

20



21

Pay attention to the nouns in your use case

Maria's figured out something really important: the nouns in a use case are usually the classes you need to write and focus on in your system.

Sharpen your pencil

Your job is to circle each noun (that's a person, place, or thing) in the use case below. Then, in the blanks below, list all the nouns that you found.

_____
_____
_____
_____
_____
_____
_____

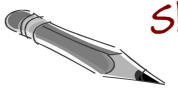
"dog" is a noun (or you could circle "owner's dog").

The Ultimate Dog Door, version 3.0
Opening/closing the door

Main Path
1. The owner's dog barks to be let out.
2. The bark recognizer "hears" a bark.
3. If it's the owner's dog barking, the bark recognizer sends a request to the door to open.
4. The dog door opens.
5. The owner's dog goes outside.
6. The owner's dog does his business.
6.1. The door shuts automatically.
6.2. The owner's dog barks to be let back inside.
6.3. The bark recognizer "hears" a bark (again).
6.4. If it's the owner's dog barking, the bark recognizer sends a request to the door to open.
6.5. The dog door opens (again).
7. The owner's dog goes back inside.
8. The door shuts automatically.

Alternate Paths
2.1. The owner hears her dog barking.
3.1. The owner presses the button on the remote control.
6.3.1. The owner hears her dog barking (again).
6.4.1. The owner presses the button on the remote control.

22



Sharpen your pencil answers

- the (owner's) dog
- the owner
- bark recognizer
- request
- dog door
- remote control
- the button
- inside/outside
- bark

The Ultimate Dog Door, version 3.0
Opening/closing the door

Main Path

1. The owner's dog barks to be let out.
2. The bark recognizer "hears" a bark.
3. If it's the owner's dog barking, the bark recognizer sends a request to the door to open.
4. The dog door opens.
5. The owner's dog goes outside.
6. The owner's dog does his business.
   - 6.1. The door shuts automatically.
   - 6.2. The owner's dog barks to be let back inside.
   - 6.3. The bark recognizer "hears" a bark (again).
   - 6.4. If it's the owner's dog barking, the bark recognizer sends a request to the door to open
   - 6.5. The dog door opens (again)
7. The owner's dog goes back inside.
8. The door shuts automatically.

Alternate Paths

- 2.1. The owner hears her dog barking.
- 3.1. The owner presses the button on the remote control.
- 6.3.1. The owner hears her dog barking (again).
- 6.4.1. The owner presses the button on the remote control.

Looking at the nouns (and verbs) in your use case to figure out classes and methods is called textual analysis.

It's all about the use case

Take a close look at Step 3 in the use case, and see exactly which classes are being used:

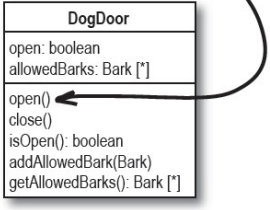
"owner's dog" is a noun, but we don't need a class for this since the dog is an actor, and outside the system.

3. If it's the owner's dog barking, the bark recognizer sends a request to the door to open.

This request—another noun without a class—is actually represented by the bark recognizer calling the open() method on the dog door.

There is no Bark class here!

The classes in use here in Step 3 are BarkRecognizer and DogDoor... not Bark!



3. If the owner's dog's bark matches the bark heard by the bark recognizer, the dog door should open.

Here's Step 3 from the use case that Randy wrote for his dog door. In his Step 3, "bark" is a noun.

Wait a second... I don't buy that. What if I happened to use just a slightly different wording?

Step 3 in Randy's use case looks a lot like Step 3 in our use case... but in his step, the focus is on the noun "bark", and not "the owner's dog." So is Randy right? Does this whole textual analysis thing fall apart if you use a few different words in your use case?

What do YOU think?

25

25

One of these things is not like the other...

Here's our Step 3, from the original use case we wrote back in Chapter 3.

3. If it's the **owner's dog barking**, the bark recognizer sends a request to the door to open.

Focus: owner's dog

Rawlf! Rawlf!

Woof.

Rooowlf!

With the right Step 3, the dog door will open for all of Bruce's barks.

And here's Step 3 from the use case that Randy came up with for the same dog door.

3. If the **owner's dog's bark** matches the bark heard by the bark recognizer, the dog door should open.

Rawlf! Rawlf!

Woof.

Rooowlf!

Focus: owner's dog's bark

With a poorly written Step 3, only one of Bruce's barks will get him in and out of the dog door.

26

26

OK, I see what Randy's mistake was: he got hung up on a bark, not the owner's dog. But even in the correct use case, we don't **have** a Dog object. So what's the point of all this, if our analysis doesn't tell us what classes to create and use?

Sharpen your pencil

Why is there no Dog class?

When you picked the nouns out of the use case, one that kept showing up was "the owner's dog." But Maria decided not to create a Dog object. Why not?

A good use case clearly and accurately explains what a system does, in language that's easily understood.

With a good use case complete, textual analysis is a quick and easy way to figure out the classes in your system.

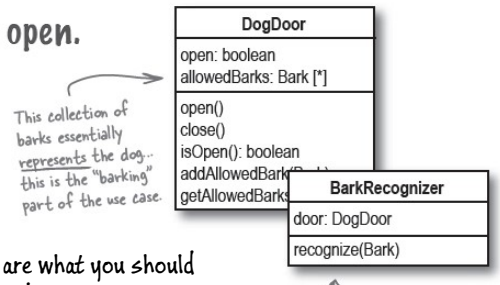
27

Remember: pay attention to those nouns!

In this use case, "owner's dog" is a noun, but it's not a class...

...and even though "barking" isn't a noun in this step, we have a Bark class.

3. If it's the **owner's dog barking**, the bark recognizer sends a request to the door to open.



This collection of barks essentially represents the dog... this is the "barking" part of the use case.

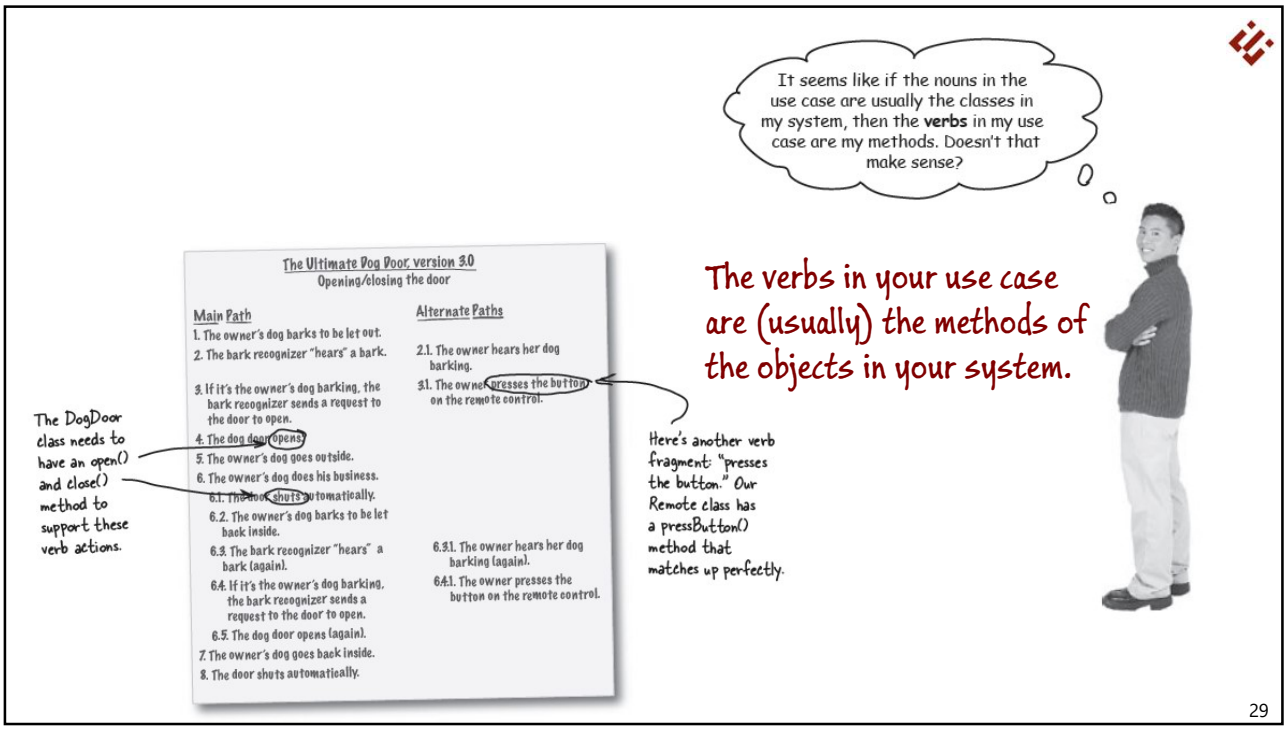
Even though this method gets a single bark, its purpose is to find out which dog barked. It runs through all the allowed barks in the dog door to see if this bark comes from the owner's dog.

The point is that the nouns are what you should focus on. If you focus on the dog in this step, you'll figure out that you need to make sure the dog gets in and out of the dog door—whether he has one bark, or multiple barks.

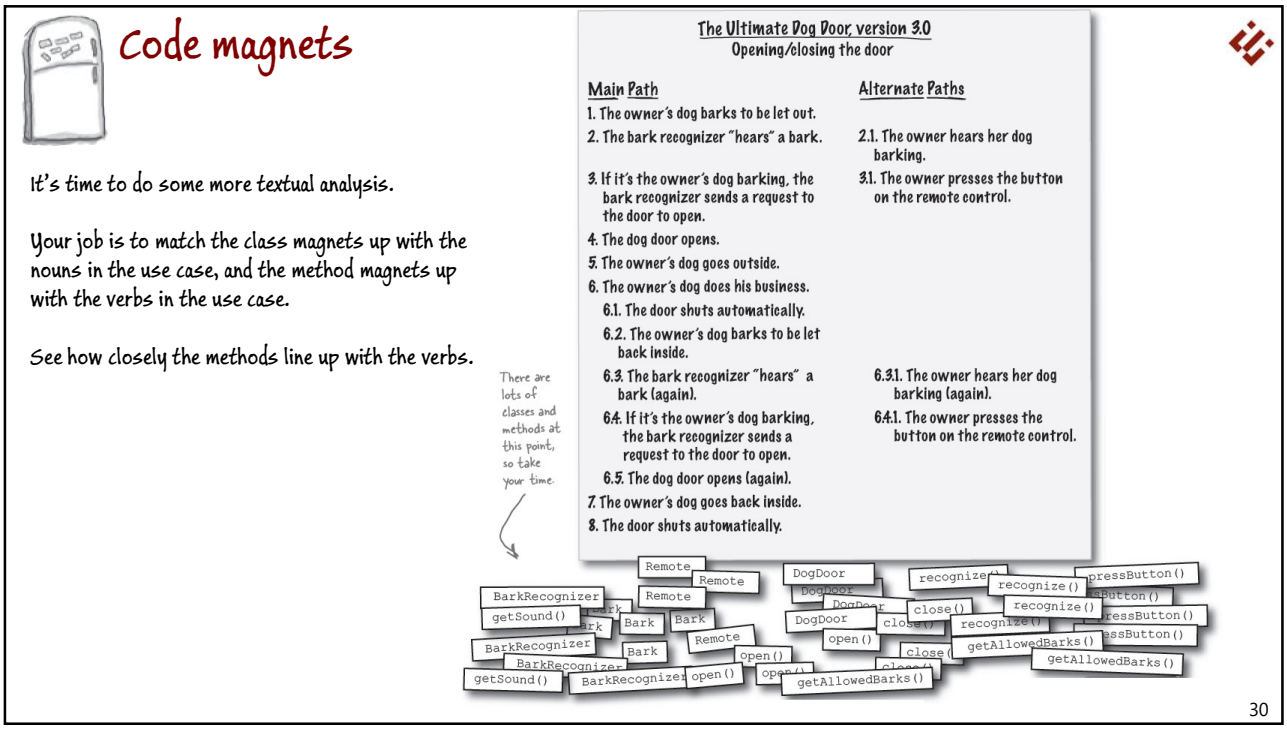
Pay attention to the nouns in your use case, even when they aren't classes in your system.

Think about how the classes you do have can support the behavior your use case describes.

28


It seems like if the nouns in the use case are usually the classes in my system, then the verbs in my use case are my methods. Doesn't that make sense?
The verbs in your use case are (usually) the methods of the objects in your system.
The Ultimate Dog Door, version 3.0
Opening/closing the door
Main Path
1. The owner's dog barks to be let out.
2. The bark recognizer "hears" a bark.
3. If it's the owner's dog barking, the bark recognizer sends a request to the door to open.
4. The dog door opens.
5. The owner's dog goes outside.
6. The owner's dog does his business.
6.1. The door shuts automatically.
6.2. The owner's dog barks to be let back inside.
6.3. The bark recognizer "hears" a bark (again).
6.4. If it's the owner's dog barking, the bark recognizer sends a request to the door to open.
6.5. The dog door opens (again).
7. The owner's dog goes back inside.
8. The door shuts automatically.
Alternate Paths
2.1. The owner hears her dog barking.
3.1. The owner presses the button on the remote control.
6.3.1. The owner hears her dog barking (again).
6.4.1. The owner presses the button on the remote control.
The DogDoor class needs to have an open() and close() method to support these verb actions.
Here's another verb fragment: "presses the button." Our Remote class has a pressButton() method that matches up perfectly.
29

29


Code magnets
It's time to do some more textual analysis.
Your job is to match the class magnets up with the nouns in the use case, and the method magnets up with the verbs in the use case.
See how closely the methods line up with the verbs.
The Ultimate Dog Door, version 3.0
Opening/closing the door
Main Path
1. The owner's dog barks to be let out.
2. The bark recognizer "hears" a bark.
3. If it's the owner's dog barking, the bark recognizer sends a request to the door to open.
4. The dog door opens.
5. The owner's dog goes outside.
6. The owner's dog does his business.
6.1. The door shuts automatically.
6.2. The owner's dog barks to be let back inside.
6.3. The bark recognizer "hears" a bark (again).
6.4. If it's the owner's dog barking, the bark recognizer sends a request to the door to open.
6.5. The dog door opens (again).
7. The owner's dog goes back inside.
8. The door shuts automatically.
Alternate Paths
2.1. The owner hears her dog barking.
3.1. The owner presses the button on the remote control.
6.3.1. The owner hears her dog barking (again).
6.4.1. The owner presses the button on the remote control.
There are lots of classes and methods at this point, so take your time.
Remote
DogDoor
BarkRecognizer
Bark
getSound()
open()
close()
recognize()
pressButton()
getAllowedBarks()
30

30

Code magnets solution

The Ultimate Dog Door, version 3.0
Opening/closing the door

Main Path

1. The owner's dog barks to be let out.
2. BarkRecogni… recognize() Bark
3. If it's getAllowedBarks() the BarkRecognizer …ends a request to … DogDoor open()
4. Th… DogDoor open()
5. The owner's dog goes outside.
6. The owner's dog does his business.
   - 6.1. The DogDoor close()
   - 6.2. The owner's dog barks to be let back inside.
   - 6.3. Th… BarkRecogni… recognize() Bark …again).
   - 6.4. If it's the o… getAllowedBarks() BarkRecognizer …ends a reque… DogDoor … open()
   - 6.5. T… DogDoor … open()
7. The owner's dog goes back inside.
8. The DogDoor … close()

Alternate Paths

- 2.1. The owner hears her dog barking.
- 3.1. The owner pressButton() on the rem… Remote …
- 6.3.1. The owner hears her dog barking (again).
- 6.4.1. The owne… pressButton() button on the Remote …trol.

Notice that most of these steps without any magnets are things that occur outside of the system and that the system then reacts to.

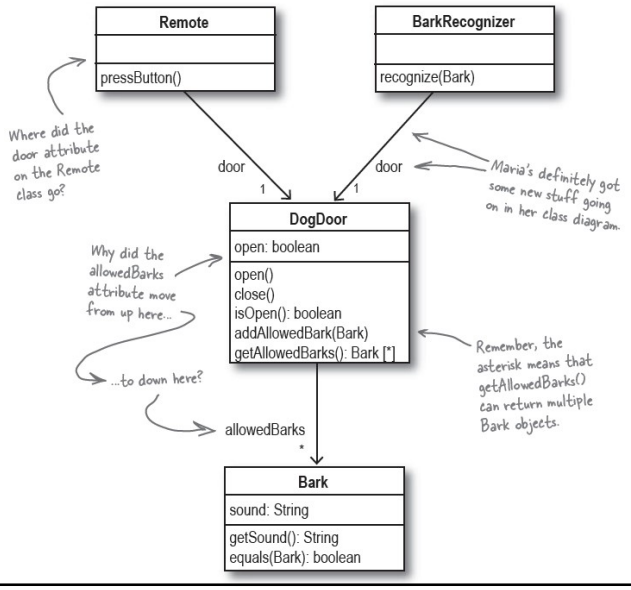
The use case still makes a lot of sense with the magnets in place! That's a good sign that our classes and methods are doing exactly what they're supposed to so that the system will be a success.

31

31

From good analysis to good classes...

Maria's Dog Door Class Diagram



32

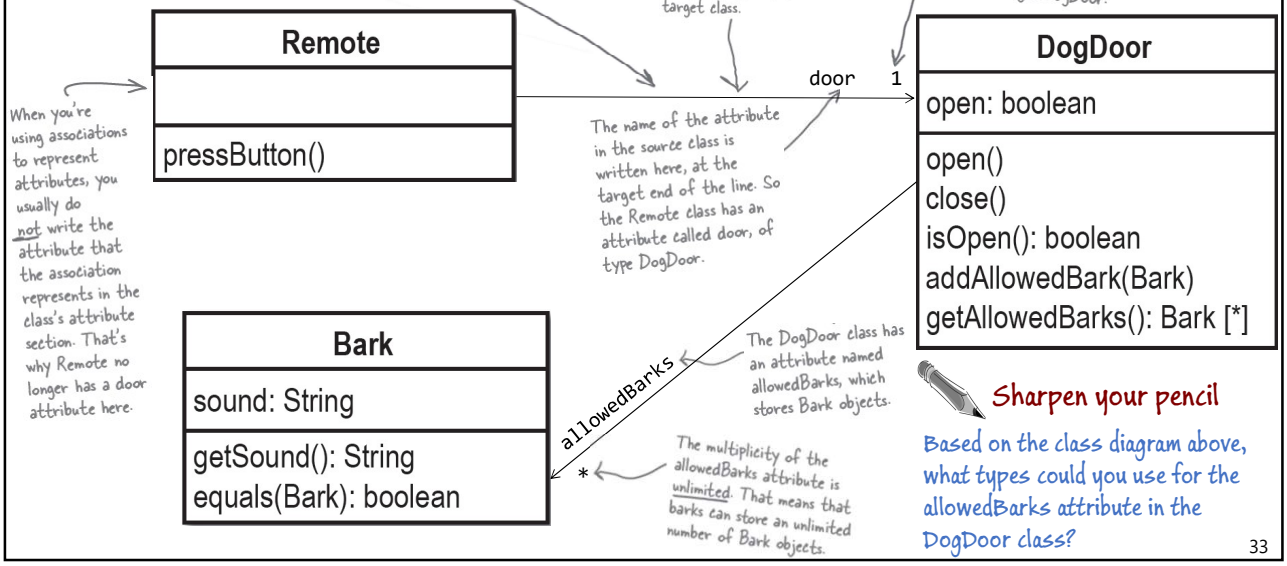
32

Class diagrams dissected

A solid line from one class to another is called an **association**. It means that one class is associated with another class, by reference, extension, inheritance, etc.

This line goes from the source class (Remote) to the target class (DogDoor). This means that the source class, Remote, has an attribute of type DogDoor, the target class.

This number is the **multiplicity** of this association. It's how many of the target type is stored in the attribute of the source class. In this case, the door attribute stores a single DogDoor.



When you're using associations to represent attributes, you usually do not write the attribute that the association represents in the class's attribute section. That's why Remote no longer has a door attribute here.

The name of the attribute in the source class is written here, at the target end of the line. So the Remote class has an attribute called door, of type DogDoor.

The DogDoor class has an attribute named allowedBarks, which stores Bark objects.

The multiplicity of the allowedBarks attribute is unlimited. That means that barks can store an unlimited number of Bark objects.

Sharpen your pencil

Based on the class diagram above, what types could you use for the allowedBarks attribute in the DogDoor class?

33

33

why use class diagrams?

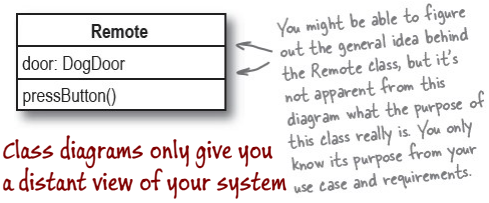
- a picture is worth a thousand words
- it helps you see the big picture and correct your mistakes
- it helps you to explain your ideas to your colleagues and boss

34

34

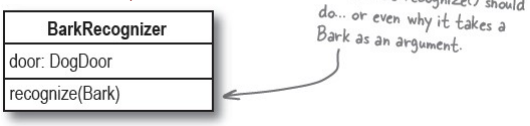
Class diagrams aren't everything

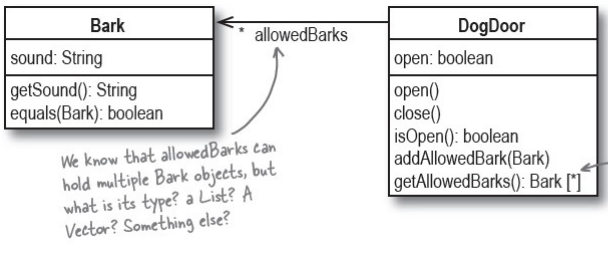
Class diagrams are a great way to get an overview of your system, and show the parts of your system to co-workers and other programmers. But there's still plenty of things that they *don't* show.



Class diagrams only give you a distant view of your system

Class diagrams don't tell you how to code your methods





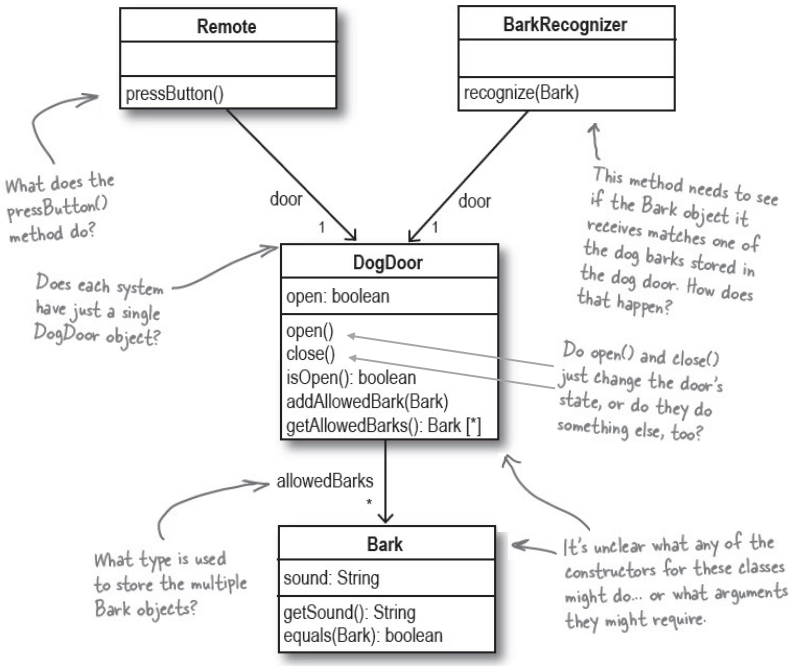
Class diagrams provide limited type information

35

35

? What's missing

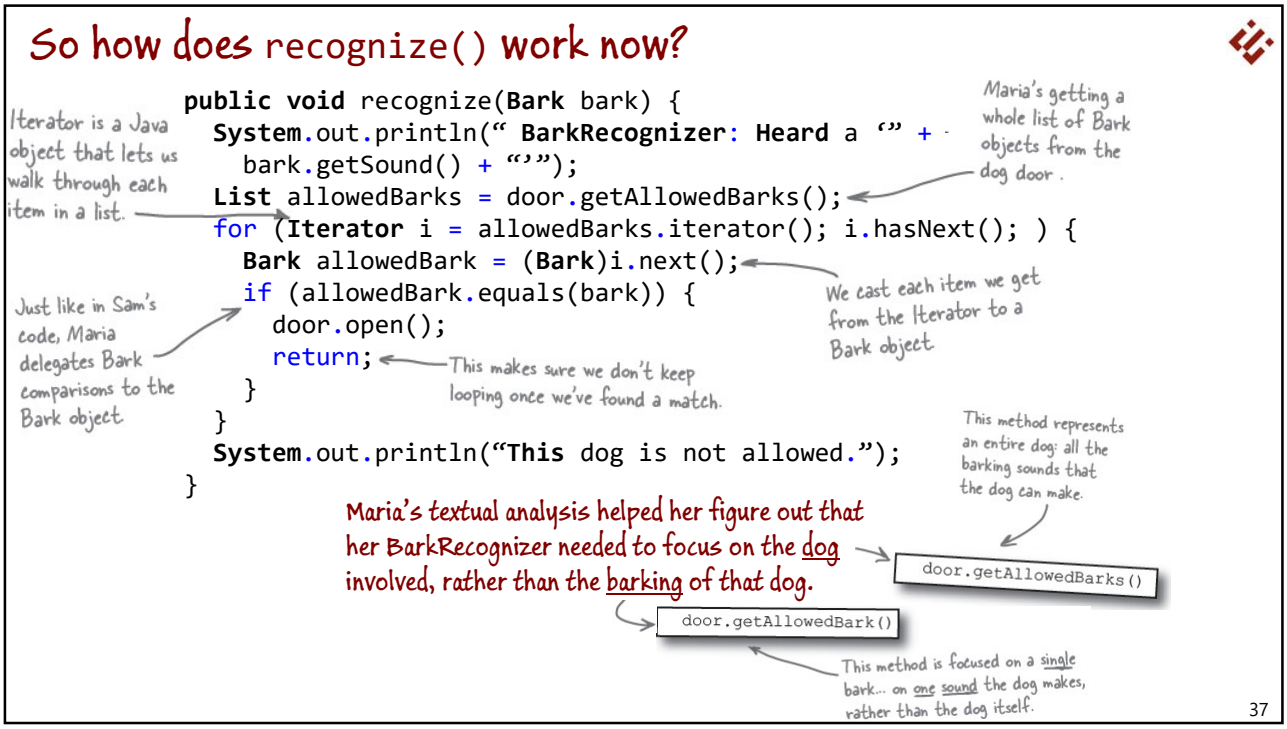
Class diagrams are great for modeling the classes you need to create, but they don't provide all the answers you'll need in programming your system.



36

36

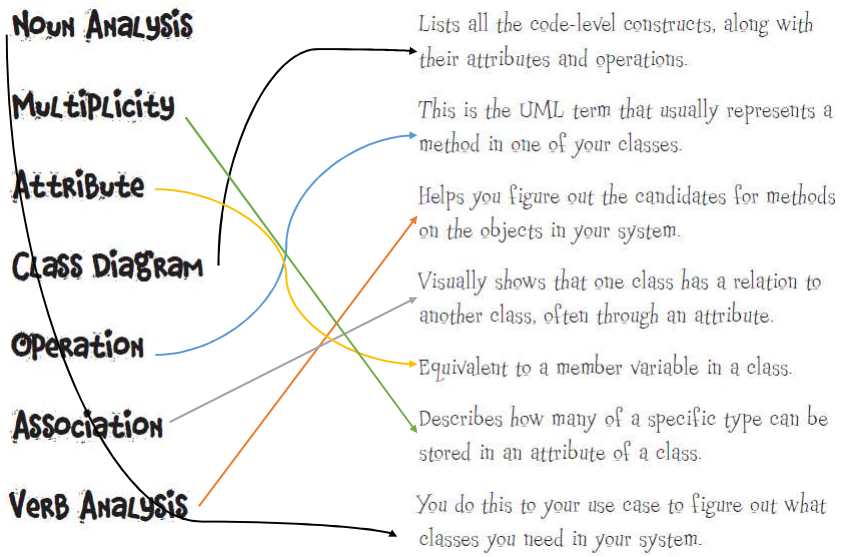
So how does recognize() work now?



37

37

? WHAT'S MY DEFINITION ?



38

38

Tools for your toolbox

Bullet Points

- Analysis helps you ensure that your software Works in the real world context, and not just in a perfect environment.
- Use cases are meant to be understood by you, your managers, your customers, and other programmers.
- You should write your use cases in whatever format makes them most usable to you and the other people who are looking at them.
- A good use case precisely lays out what a system does, but does not indicate how the system accomplishes that task.
- Each use case should focus on only one customer goal. If you have multiple goals, you will need to write mutiple use cases.
- Class diagrams give you an easy way to show your system and its code constructs at a 10,000-foot view.
- The attributes in a class diagram usually map to the member variables of your classes.
- The operations in a class diagram usually represent the methods of your classes.
- Class diagrams leave lots of detail out, such as class constructors, some type information, and the purpose of operations on your classes.
- Textual analysis helps you translate a use case into code-level classes, attributes, and operations.
- The nouns of a use case are candidates for classes in your system, and the verbs are candidates for methods on your system's classes.

39

39