

Part I
03. Requirement Change: The Constant in Software Developement

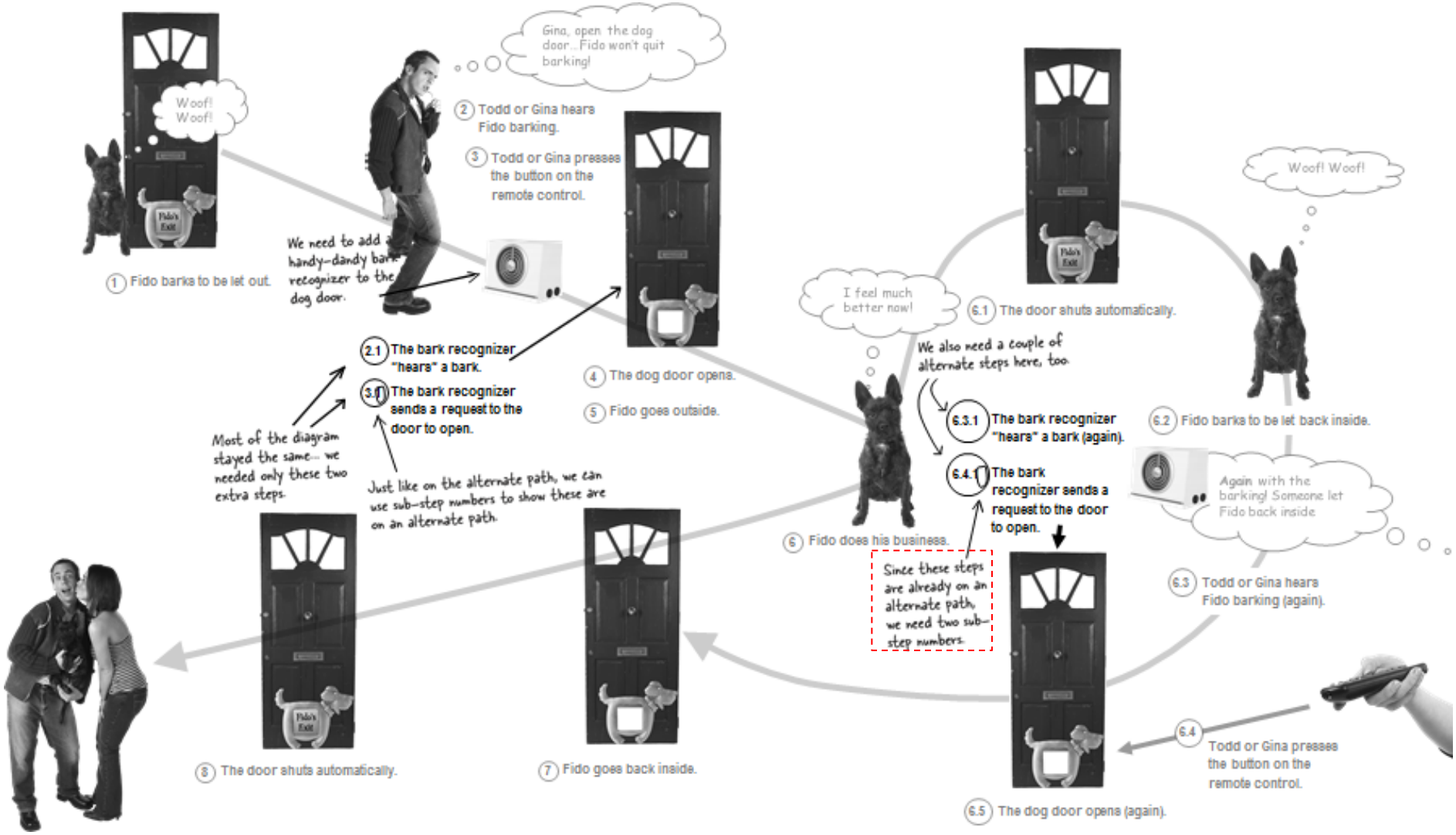
Requirement Change

- What might Change
  - Environment changes
  - Market changes
  - Program evolvements
- Requirements always change.
  - If we've got good **use cases**, we can usually change our software quickly to adjust to those new requirements.

Todd and Gina's Dog Door

Can you add some hardware to recognize **Fido's bark** when he wants go to out and come back in and automatically open the door? That way we don't need to hear him or find that darn remote that keeps getting lost.

New Scenario



New Use Case

Todd and Gina's Dog Door, version 2.1
What the Door Does

1. Fido barks to be let out.
2. Todd or Gina hears Fido barking.
   - 2.1. The bark recognizer "hears" a bark.
3. Todd or Gina presses the button on the remote control.
   - 3.1. The bark recognizer sends a request to the door to open.
4. The dog door opens.
5. Fido goes outside.
6. Fido does his business.
   - 6.1. The door shuts automatically.
   - 6.2. Fido barks to be let back inside.
   - 6.3. Todd or Gina hears Fido barking (again).
     - 6.3.1. The bark recognizer "hears" a bark (again).
   - 6.4. Todd or Gina presses the button on the remote control.
     - 6.4.1. The bark recognizer sends a request to the door to open.
   - 6.5. The dog door opens (again).
7. Fido goes back inside.
8. The door shuts automatically.

There are now alternate steps for both #2 and #3.

These are listed as sub-steps, but they really are providing a completely different path through the use case.

These sub-steps provide an additional set of steps that can be followed...

...but these sub-steps are really a different way to work through the use case.

Even the alternate steps now have alternate steps.

Does this make sense to you? How would you improve it?

Improved New Use Case

We've moved the steps that can occur instead of the steps on the main path over here to the right.

Now we've added a label to tell us that these steps on the left are part of the main path.

Todd and Gina's Dog Door, version 2.2
What the Door Does

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

This is a little clearer: we can use Step 2, OR Step 2.1, and then Step 3, OR Step 3.1.

When there's only a single step, we'll always use that step when we go through the use case.

These sub-steps are optional... you may use them, but you don't have to. But they're still on the left, because they don't replace steps on the main path.

These steps on the right can replace Steps 6.3 and 6.4. You can only take one step to work through the use case: either the step on the left, OR the step on the right.

No matter how you work through this use case, you'll always end up at Step 8 on the main path.



Improved New Use Case (Cont.)

The main path should be what we want to happen most of the time.

Todd and Gina's Dog Door, version 2.3
What the Door Does

Main Path

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

Now the steps that involve the bark recognizer are on the main path, instead of an alternate path.

Alternate Paths

- 2.1. Todd or Gina hears Fido barking.
- 3.1. Todd or Gina presses the button on the remote control.

Todd and Gina won't use the remote most of the time, so the steps related to the remote are better as an alternate path.

- 6.3.1. Todd or Gina hears Fido barking (again).
- 6.4.1. Todd or Gina presses the button on the remote control.

Revisiting Alternative Path

- Alternate paths can
  - 1. Be additional steps added to the main path, or
  - 2. Provide steps that allow to get to the goal in a totally different paths through parts of a use case.

Scenario

Todd and Gina's Dog Door, version 2.3
What the Door Does

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

Each path through this use case starts with Step 1.

Let's take this alternate path, and let Todd and Gina handle opening the door with the remote.

We'll take the optional sub-path here, where Fido gets stuck outside.

We're letting Todd and Gina handle opening the door again, on the alternate path.

You'll always end up at Step 8, with Fido back inside.

Following the arrows gives us a particular path through the use case.
A path like this is called a ***scenario***.
There are usually several possible scenarios in a single use case.

Scenario (Cont.)

- A complete path through a use case, from first step to the last, is called a scenario.
- Most use cases have several different scenarios, but they always share the same user goal.
- How many scenarios in Todd and Gina's use case? (six)

This is just the use case's main path.

These two don't take the optional alternate path where Fido gets stuck outside.

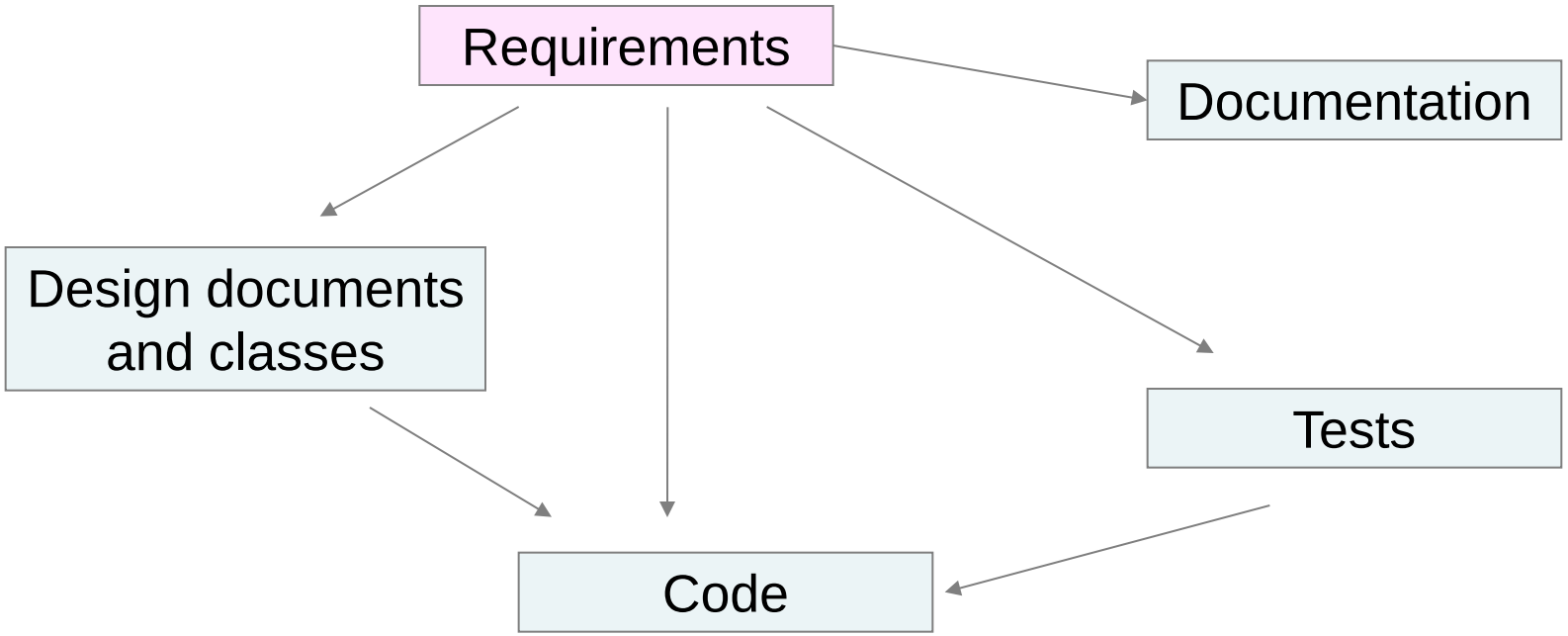
1. 1, 2.1, 3.1, 4, 5, 6, 6.1, 6.2, 6.3.1, 6.4.1, 6.5, 7, 8
2. 1, 2, 3, 4, 5, 6, 7, 8
3. 1, 2.1, 3.1, 4, 5, 6, 7, 8
4. 1, 2.1, 3.1, 4, 5, 6, 6.1, 6.2, 6.3, 6.4, 6.5, 7, 8

If you take Step 2.1, you'll always also take Step 3.1.

When you take 6.3.1, you'll also take Step 6.4.1.

5. 1, 2, 3, 4, 5, 6, 6.1, 6.2, 6.3.1, 6.4.1, 6.5, 7, 8
6. 1, 2, 3, 4, 5, 6, 6.1, 6.2, 6.3, 6.4, 6.5, 7, 8
7. <nothing else>
8. <nothing else>

A Cain of Development Artifacts



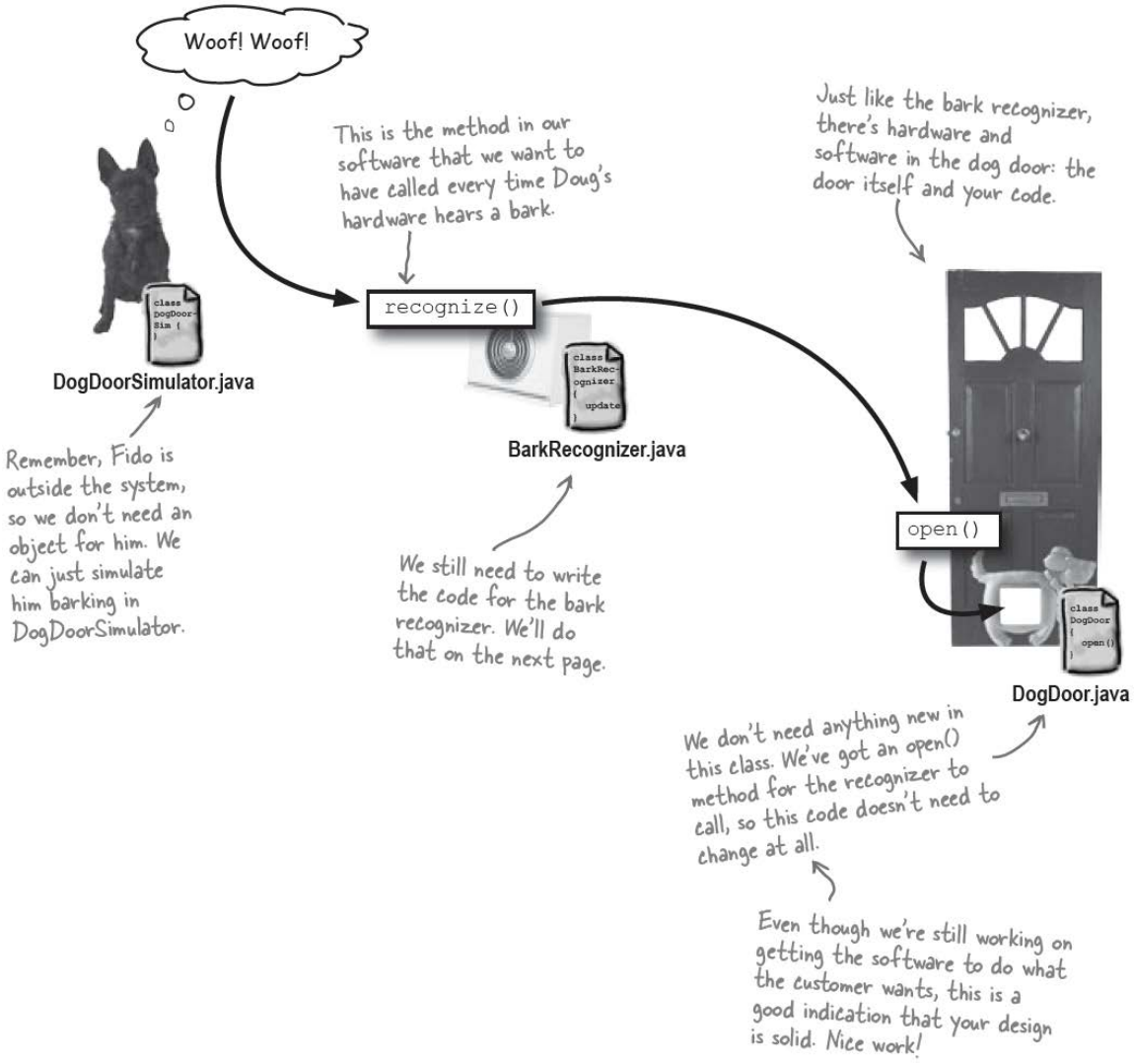
Requirement Change

Todd and Gina's Dog Door, version 2.3

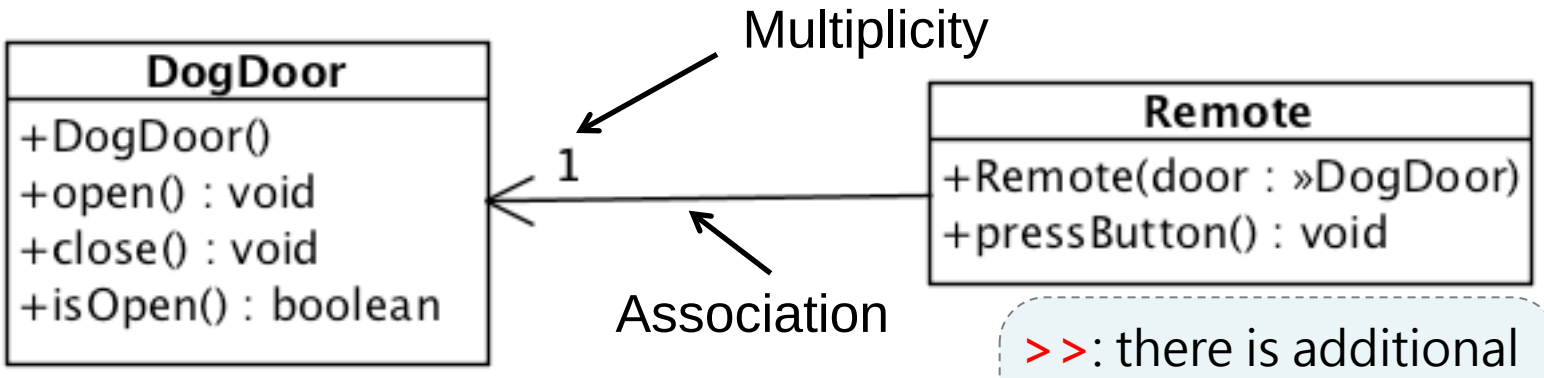
Requirements List

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.
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.

Bark Recognizer



Review the Old Design

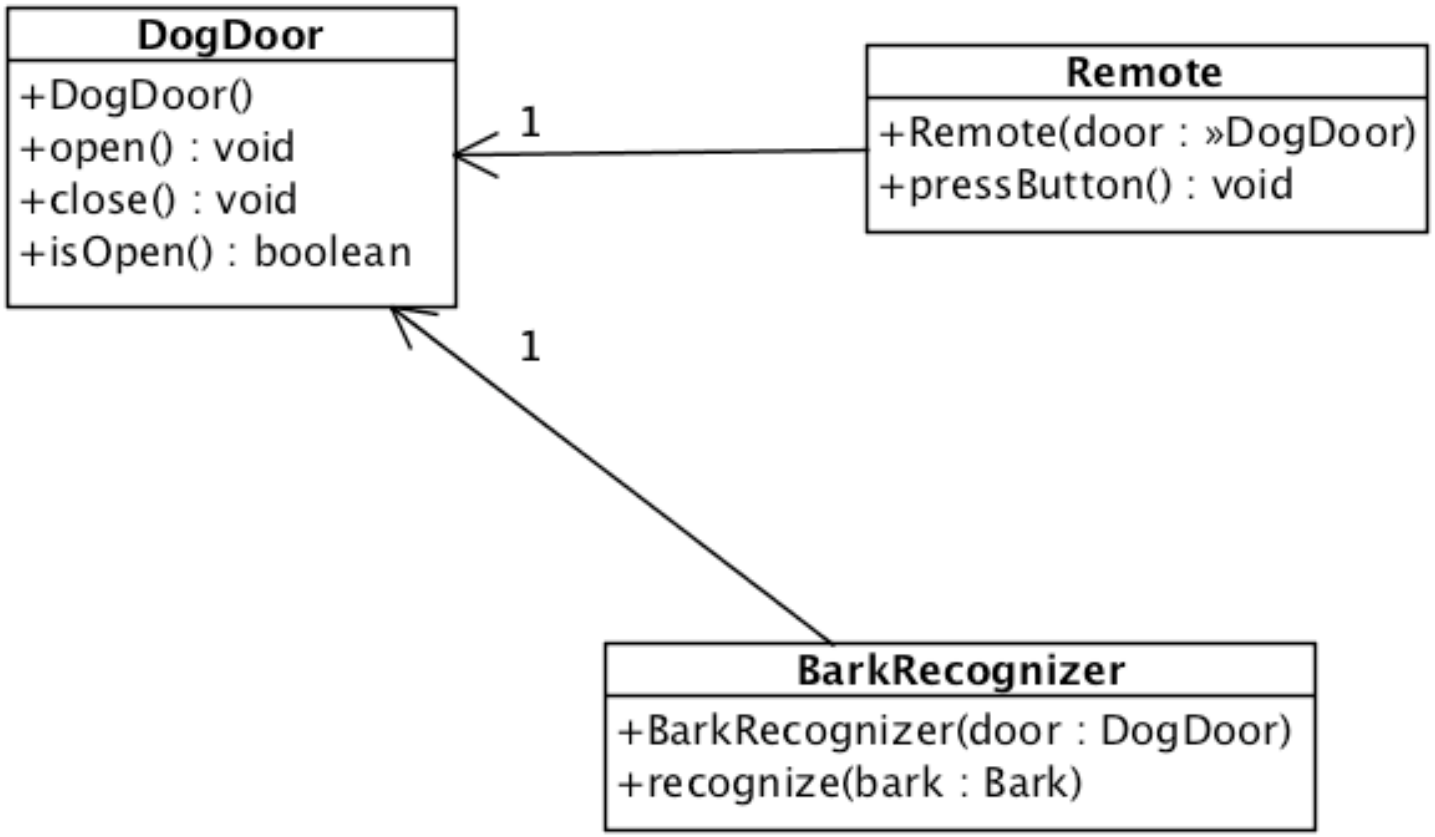


>>: there is additional type information preceeding the DogDoor

- The line between the classes is called an ***association***. An association represents some relationship between the two classes. In this case, it shows that a Remote is related (or associated) to a DogDoor.
- The ***multiplicity*** indicates that a Remote is associated with exactly one DogDoor.
- The ***arrow*** on the association shows that the code is written in such a way that from a Remote, you can get to the DogDoor to which it is associated.

Multiplicity	Option	Cardinality
0..0	0	Collection must be empty
0..1		No instances or one instance
1..1	1	Exactly one instance
0..*	*	Zero or more instances
1..*		At least one instance
5..5	5	Exactly 5 instances
m..n		At least m but no more than n instances

New Design


DogDoor
+DogDoor()
+open() : void
+close() : void
+isOpen() : boolean
Remote
+Remote(door : »DogDoor)
+pressButton() : void
1
1
BarkRecognizer
+BarkRecognizer(door : DogDoor)
+recognize(bark : Bark)

New Code – BarkRecognizer.java

```
public class BarkRecognizer {

  private DogDoor door;

  public BarkRecognizer(DogDoor door) {
    this.door = door;
  }

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

We'll store the dog door that this bark recognizer is attached to in this member variable.

The BarkRecognizer needs to know which door it will open.

Every time the hardware hears a bark, it will call this method with the sound of the bark it heard.

All we need to do is output a message letting the system know we heard a bark...

...and then open up the dog door.

New Test Drive (DogDoorSimulator.java)

```
public class DogDoorSimulator {

  public static void main(String[] args) {
    DogDoor door = new DogDoor();
    BarkRecognizer recognizer = new BarkRecognizer(door);
    Remote remote = new Remote(door);

    // Simulate the hardware hearing a bark
    System.out.println("Fido starts barking.");
    recognizer.recognize("Woof");

    System.out.println("\nFido has gone outside...");

    System.out.println("\nFido's all done...");

    try {
      Thread.currentThread().sleep(10000);
    } catch (InterruptedException e) { }

    System.out.println("...but he's stuck outside!");

    // Simulate the hardware hearing a bark again
    System.out.println("Fido starts barking.");
    recognizer.recognize("Woof");

    System.out.println("\nFido's back inside...");
  }
}
```

Create the BarkRecognizer, connect it to the door, and let it listen for some barking.

We don't have real hardware, so we'll just simulate the hardware hearing a bark.*

Here's where our new BarkRecognizer software gets to go into action.

We simulate some time passing here.

We test the process when Fido's outside, just to make sure everything works like it should.

Notice that Todd and Gina never press a button on the remote this time around.

```
%java DogDoorSimulator
Fido starts barking.
  BarkRecognizer: Heard a 'Woof'
The dog door opens.

Fido has gone outside...

Fido's all done...
...but he's stuck outside!
Fido starts barking.
  BarkRecognizer: Heard a 'Woof'
The dog door opens.

Fido's back inside...
```

There's a big problem with our code, and it shows up in the simulator.
Can you figure out what the problem is?

Problem in the New Tester

- In the new version, the door doesn't close automatically.

```
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);
  }
}
```

When Todd and Gina press the button on the remote, this code also sets up a timer to close the door automatically.

Remember, this timer waits 5 seconds, and the sends a request to the dog door to close itself.

class Remote { pressButton() }

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

We open the door, but never close it.

class BarkRecognizer

Update DogDoor and Simply Remote

```
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
    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!

```
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);
  }
}
```

A Final Test Drive

```
File Edit Window Help PestControl
%java DogDoorSimulator
Fido starts barking.
  BarkRecognizer: Heard a 'Woof'
The dog door opens.

Fido has gone outside...

Fido's all done...
The dog door closes.
...but he's stuck outside!

Fido starts barking.
  BarkRecognizer: Heard a 'Woof'
The dog door opens.

Fido's back inside...
The dog door closes.
```

Yes! The door is closing by itself now.

Remarks

- Sometimes a change in requirements reveals problems with our system that we didn't know were there.
- Change is constant, and our system should always improve every time we work on it.
- When our system needs to work in a new or different way, begin by updating the use case.
- We should almost always avoid duplicate code.
- A single use case can have multiple scenarios.