CSE203

Principles of Software Design and Development

09. Iterating and Testing

The Software is
still for the Customer

Your toolbox is filling up

1. Make sure your software does what the customer wants it to do.

2. Apply basic OO principles to add flexibility.

3. Strive for a maintainable, reusable design.

The 3 steps to writing great software

Our OO principles help us write well-designed, flexible OO software.

We've got a whole slew of principles and techniques to gather requirements, analyze and design, and solve all types of software problems.

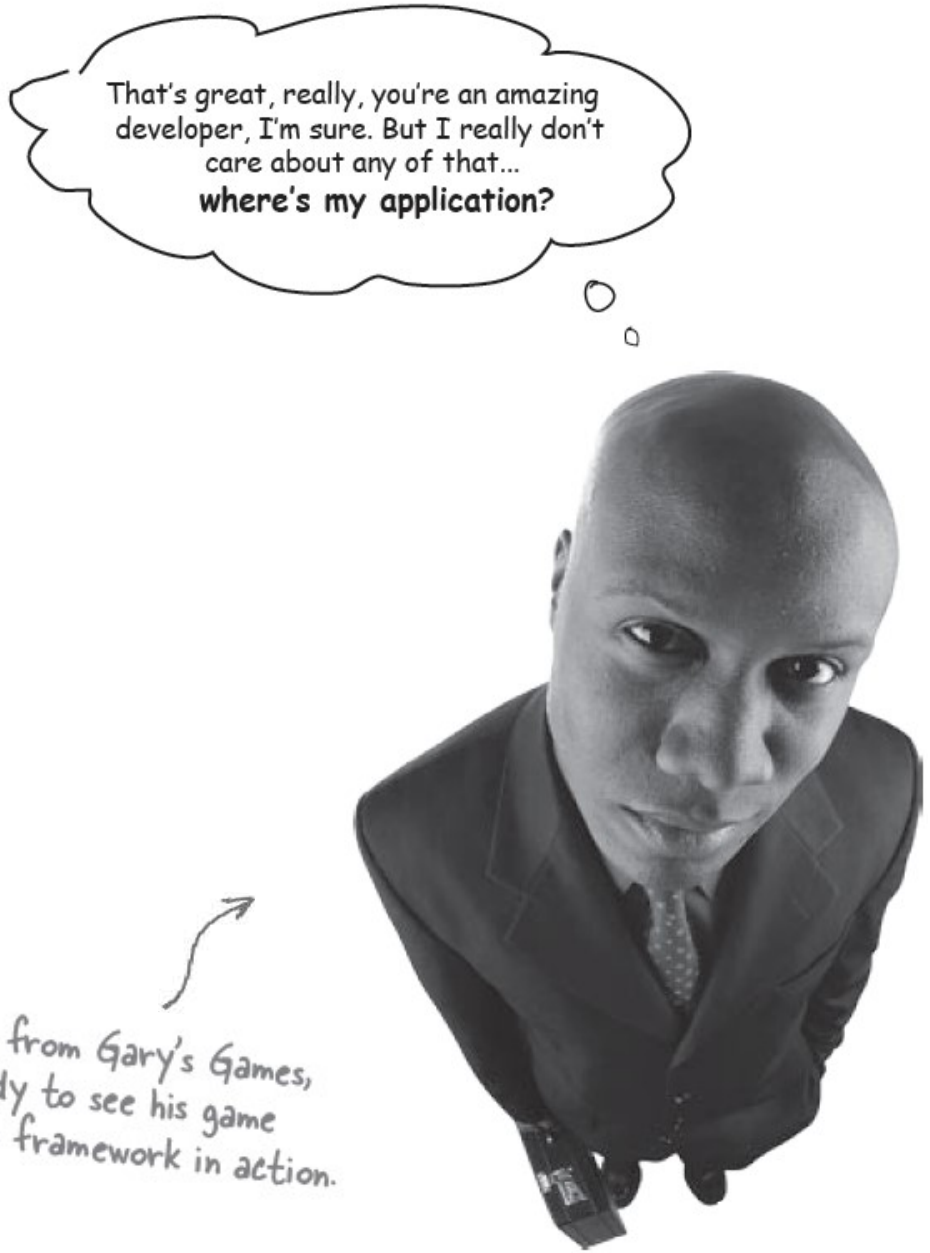
we used use case diagrams and a key feature list to turn a simple vision statement into an application architecture.

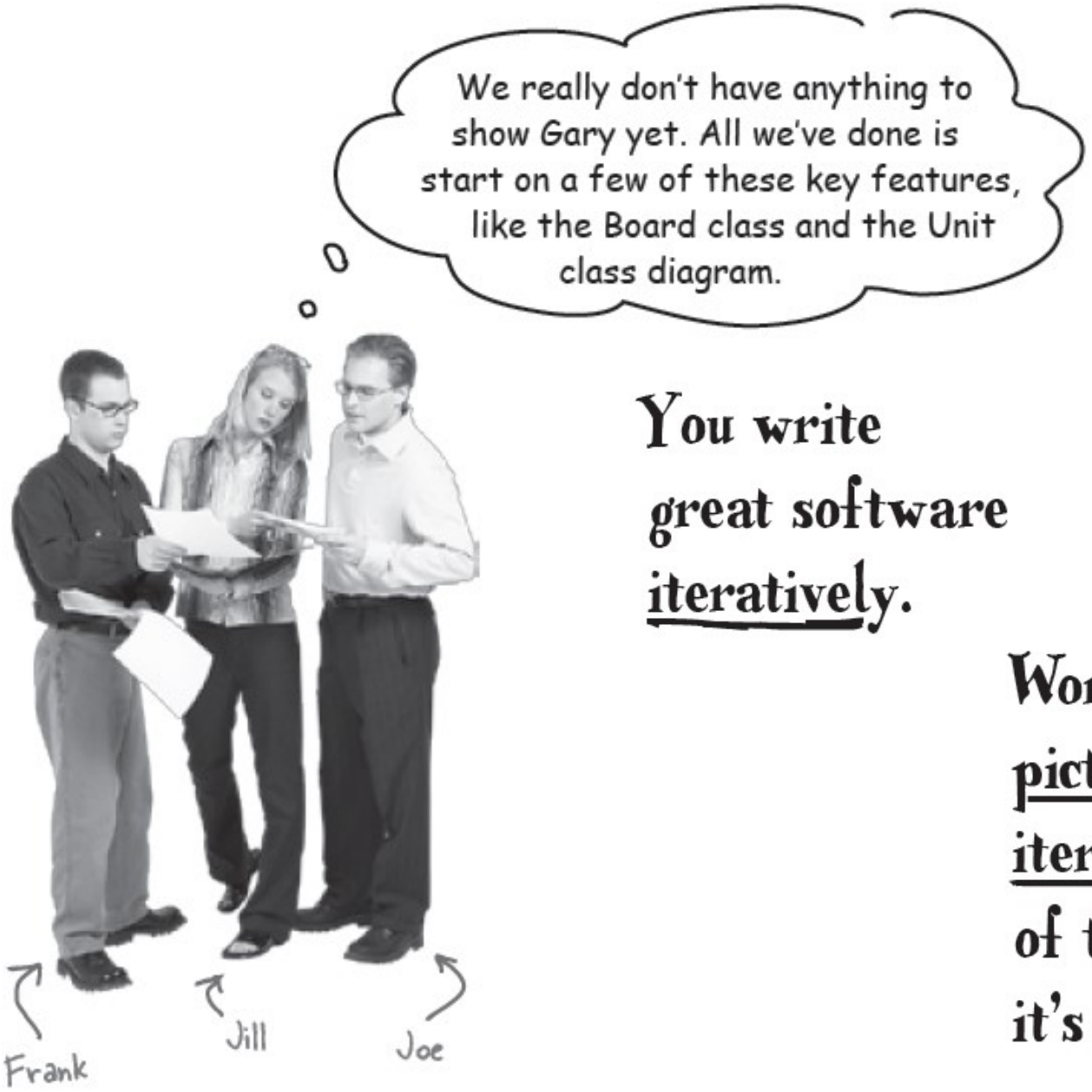
But you're still writing your software for the CUSTOMER!

All the tools and techniques you've been learning are terrific... but none of them matter if you don't use them to produce great software that makes your customer happy.

And most of the time, your customer won't care about all the OO principles and diagrams you create.

They just want the software to work the way that it's supposed to.





You write great software iteratively.

Work on the big picture, and then iterate over pieces of the app until it's complete.

Iterating deeper: two basic choices

You can choose to focus on specific features of the application. This approach is all about taking one piece of functionality that the customer wants, and working on that functionality until it's complete.

Feature driven development

...is when you pick a specific feature in your app, and plan, analyze, and develop that feature to completion.

You can also choose to focus on specific flows through the application. This approach takes a complete path through the application, with a clear start and end, and implements that path in your code.

Use case driven development

...is when you pick a scenario through a use case, and write code to support that complete scenario through the use case.

You'll often see the terms "flow" and "scenario" used interchangeably.

Both approaches to iterating are driven by good requirements.

Because requirements come from the customer, both approaches focus on delivering what the customer wants.

Feature driven development

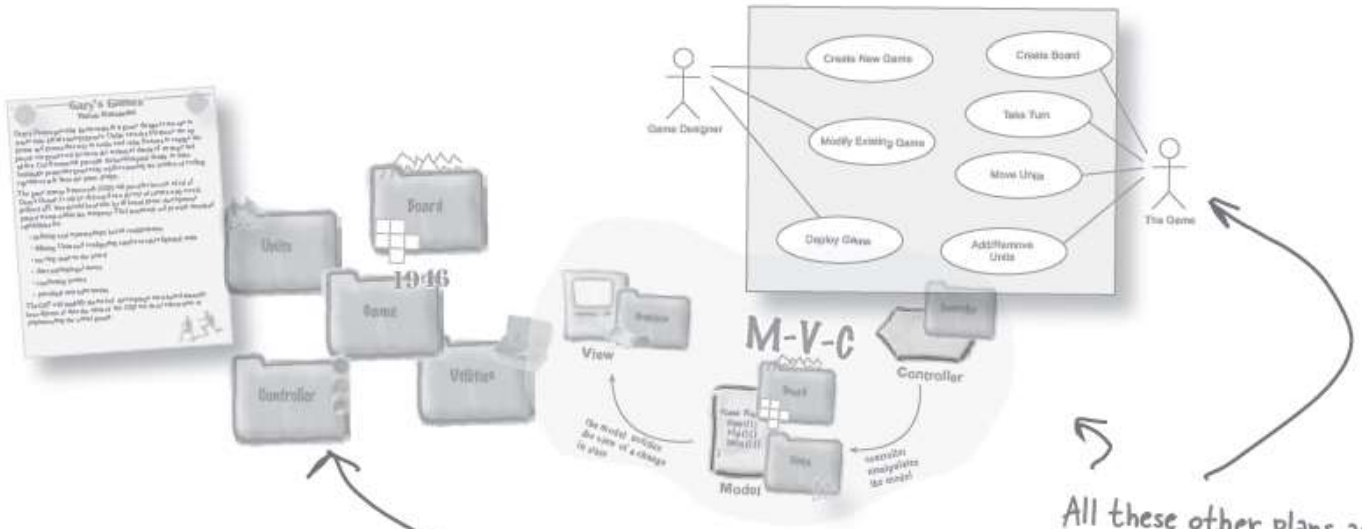
In **feature driven development**, you work on a single feature at a time, and then iterate, knocking off features one at a time until you've finished up the functionality of an application.

Gary's Game System Framework
Feature List

1. The framework supports different types of terrain.
2. The framework supports different time periods, including fictional periods like sci-fi and fantasy.
3. The framework supports multiple types of troops or units that are game-specific.
4. The framework supports add-on modules for additional campaigns or battle scenarios.
5. The framework provides a board made up of square tiles, and each tile has a terrain type.
6. The framework keeps up with whose turn it is.
7. The framework coordinates basic movement.

With feature driven development, you pick a single feature, and the focus is on the feature list of your app.

So we might take feature #1, and work on the Terrain class, as well as the Tile class, to support different types of terrain.



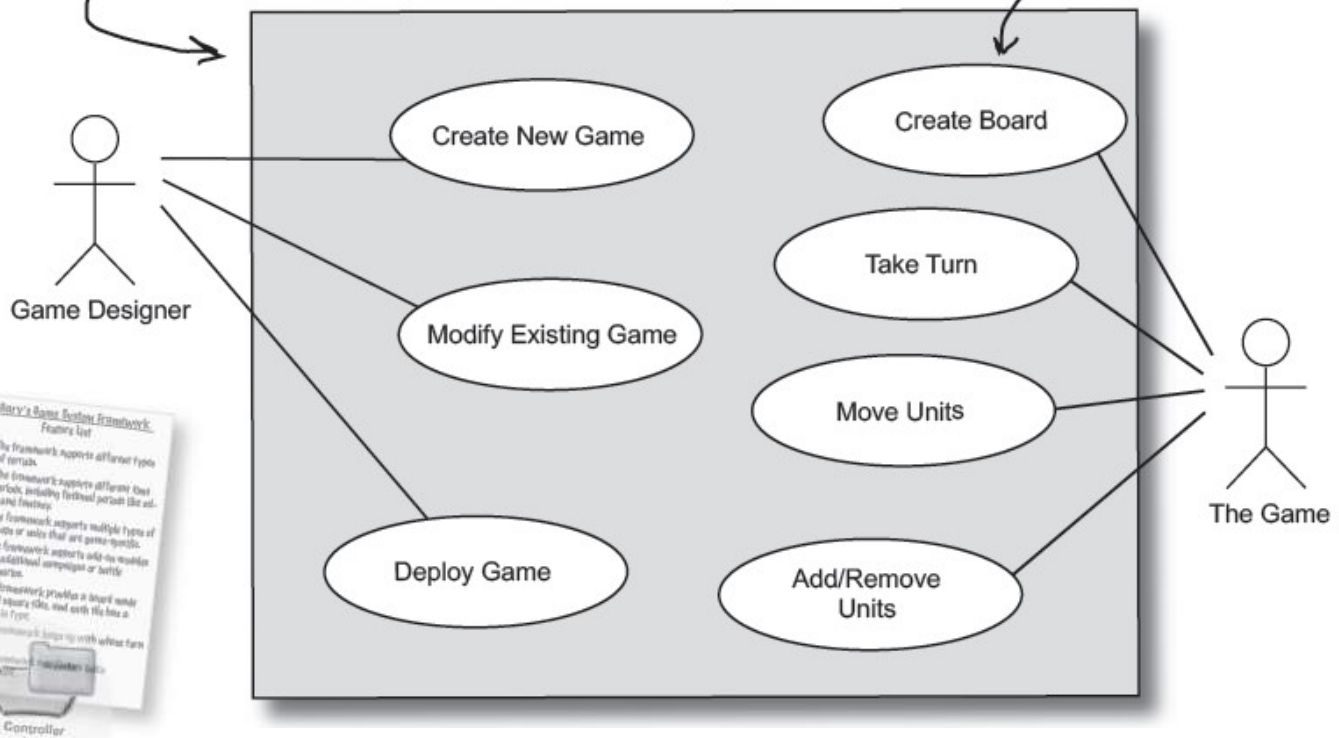
All these other plans and diagrams are used, but your feature list is the focus.

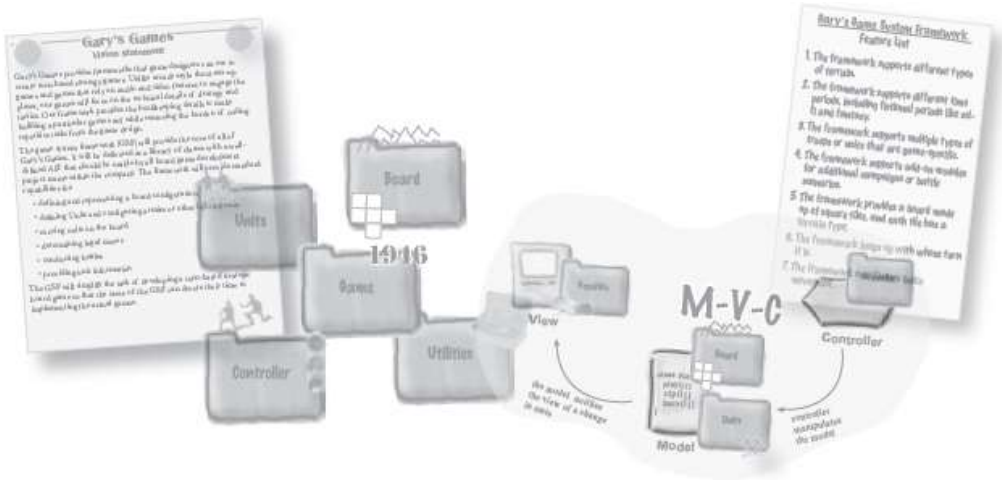
Use case driven development

With **use case driven development**, you work on completing a single scenario through a use case. Then you take another scenario and work through it, until all of the use case's scenarios are complete. Then you iterate to the next use case, until all your use cases are working.

With use case driven development, you work from the use case diagram, which lists the different use cases in your system.

Here, you could take the Create Board use case, and figure out all the scenarios for that use case, and write code to handle all of them.



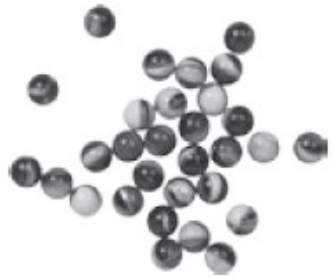


So how do you decide which to use?

WHAT'S THE DIFFERENCE

Feature driven development is more granular

A single feature is often pretty small, and every application has a lot of them.



Works well when you have a lot of different features that don't interconnect a whole lot.

Allows you to show the customer working code faster.

Is very functionality-driven. You're not going to forget about any features using feature driven development.

Works particularly well on systems with lots of disconnected pieces of functionality.

Use case driven development is more "big picture"

You'll be working on pretty major chunks of code at a time, since a single scenario often involves a lot of functionality.

Works well when your app has lots of processes and scenarios rather than individual pieces of functionality.

Allows you to show the customer bigger pieces of functionality at each stage of development.

Is very user-centric. You'll code for all the different ways a user can use your system with use case driven development.

Works particularly well on transactional systems, where the system is largely defined by lengthy, complicated processes.

NAME THAT APPROACH!

	Use Case Driven	Feature Driven
This approach deals with really small pieces of your application at a time.	☐	✓
This approach lets you focus on just a part of your application at a time.	✓	✓
This approach is all about a complete process in your application.	✓	☐
Using this approach, you can always test to see if you've completed the part of the application you're working on.	✓	✓
When you use this approach, your focus is on a diagram, not a list.	✓	☐

Let's use feature driven development

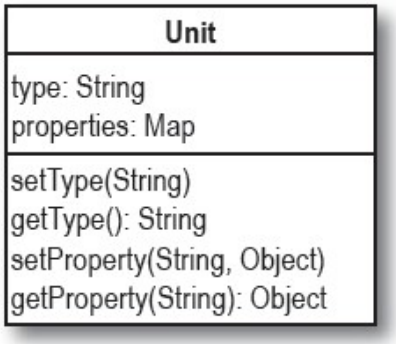
Since Gary's losing patience, let's go with feature driven development.

We go back to our feature list.

Gary's Game System Framework
Feature List

1. The framework supports different types of terrain.
2. The framework supports different time periods, including fictional periods like sci-fi and fantasy.
3. The framework supports multiple types of troops or units that are game-specific.
4. ~~The framework supports add-on modules~~ for additional campaigns or battle scenarios.
5. The framework provides a board made up of square tiles, and each tile has a terrain type.
6. The framework keeps up with whose turn it is.
7. The framework coordinates basic movement.

We already have the class diagram for Unit, so let's write the code for that class, and knock off feature #3.

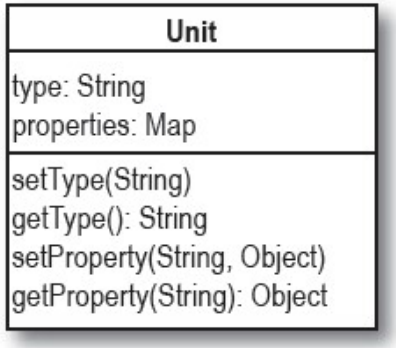


We also know that most of our other features depend on this class, so that makes it an even better candidate to start with.

Analysis of a feature

3. The framework supports multiple types of troops or units that are game-specific.

Here's what we've got so far... but this is still a pretty generic description of what we need to code.



This looks like the blueprint for a good Unit class. So is anything missing?

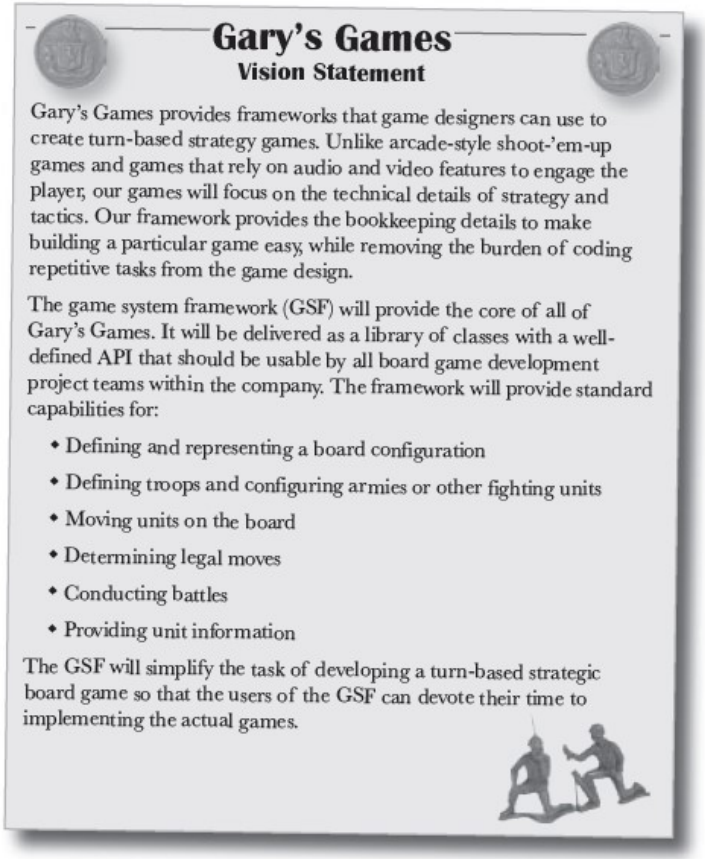
It looks like we've got everything we need to start coding, right? To help us make sure we haven't forgotten anything, let's go back to using some textual analysis.

We don't have a use case to analyze, but we can revisit the vision statement for Gary's games, and see if we're covering everything that Gary wanted his units to do.

Compare the class diagram for Unit with this vision statement.
Are there things missing from our class diagram?

What else might Gary expect to see when you say, "I'm done with writing code for the units in your framework?"

Here's Gary's vision statement



Gary's Games
Vision Statement

Gary's Games provides frameworks that game designers can use to create turn-based strategy games. Unlike arcade-style shoot-'em-up games and games that rely on audio and video features to engage the player, our games will focus on the technical details of strategy and tactics. Our framework provides the bookkeeping details to make building a particular game easy, while removing the burden of coding repetitive tasks from the game design.

The game system framework (GSF) will provide the core of all of Gary's Games. It will be delivered as a library of classes with a well-defined API that should be usable by all board game development project teams within the company. The framework will provide standard capabilities for:

- Defining and representing a board configuration
- Defining troops and configuring armies or other fighting units
- Moving units on the board
- Determining legal moves
- Conducting battles
- Providing unit information

The GSF will simplify the task of developing a turn-based strategic board game so that the users of the GSF can devote their time to implementing the actual games.

Fleshing out the Unit class

In our class diagram, all we've really figured out is how to represent the properties of a unit. But in Gary's vision statement, he's expecting his game system framework to support a lot more than just those game-specific properties.

This makes sense, because the key feature we were focusing on in Chapter 7 was not the entire Unit class, but just game-specific properties of a Unit.

Unit
type: String properties: Map
setType(String) getType(): String setProperty(String, Object) getProperty(String): Object

1. Each unit should have properties, and game designers can add new properties to unit types in their own games.

Our class diagram is focused on this particular aspect of the Unit class right now.

2. Units have to be able to move from one tile on a board to another.

You should have some ideas about how to handle this from our work on a related key feature

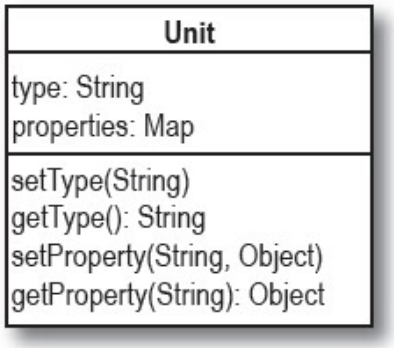
3. Units can be grouped together into armies.

These new features are all pulled straight from Gary's vision statement.

Wow, great. **Another** list of things you're going to do. Look, I trust you and all, but I need to see something more than scraps of paper to believe your code is working.

Gary isn't satisfied with your use cases and lists... what do you think would make Gary believe that what you're working on will satisfy his requirements?

Showing off the Unit class



This may be what you need to start coding the Unit class, but it doesn't do anything to prove to Gary that you've got working units in the game system framework.

How about a test? Can't you come up with a way to show me the unit has properties, and can move around, and that you've got support for armies? I want to see your code actually running.

YOUR CUSTOMERS WANT TO SEE SOMETHING THAT MAKES SENSE TO THEM

Writing test scenarios

Be careful... this "scenario" isn't the same as the "scenario" we've been talking about in a use case scenario.

Test cases don't have to be very complex; they just provide a way to show your customer that the functionality in your classes is working correctly.

For the properties of a unit, we can start out with a simple test scenario that creates a new **Unit**, and adds a property to the unit. We could just show our customer a running program that displays output like this:

This test, although simple, lets your customer "see" that the code you're writing really works.

We start out by creating the Unit...

...then we set some properties...

...and finally get the properties and make sure the values match up with what we set.

File Edit Window Help ItWorks

```
%java UnitTester
Testing the Unit class...
...Created a new unit
...Set "type" to "infantry"
...Set "hitPoints" to 25
...Getting unit type: "infantry"
...Getting unit hitPoints: 25

Test complete.
```

1. Each unit should have properties, and game designers can add new properties to unit types in their own games.

BE the Customer

We've already got one test scenario. It's your job to play Gary, and think of two more test scenarios that we can use to prove that the **Unit** class is working like it should. Write the output of each scenario in the code windows on the right.

Write the output you want Gary to see in these blanks.

```
File  Edit  Window  Help  Scenario2
%java UnitTester
Testing the Unit class...
_____
_____
_____
_____
_____

Test complete.
```

```
File  Edit  Window  Help  Scenario3
%java UnitTester
Testing the Unit class...
_____
_____
_____
_____
_____

Test complete.
```

Test driven development focuses on getting the behavior of your classes right.

BE the Customer Solutions

Scenario #2: Changing property values

This is pretty similar-looking to the first scenario, above, but it tests changing a property value, rather than just setting and retrieving a property.

We always begin by creating a new Unit, so we can test things out.

Next we set the value of hitPoints, and then reset it to a new value.

Finally, we make sure hitPoints has the most current value, and not the original value of 25.

```
File Edit Window Help Scenario2
%java UnitTester
Testing the Unit class...
...Created a new unit
...Set "hitPoints" to 25
...Set "hitPoints" to 15
...Getting unit hitPoints: 15

Test complete.
```

You should test your software for every possible usage you can think of. Be creative!

BE the Customer Solutions

This test shows the customer that you're not just dealing with happy paths... you're thinking about how to deal with uses of the software that are outside of the norm.

Scenario #3: Getting non-existent property values

Start out by creating a new Unit again.

Next, we set a hitPoints property, which is the normal Unit usage.

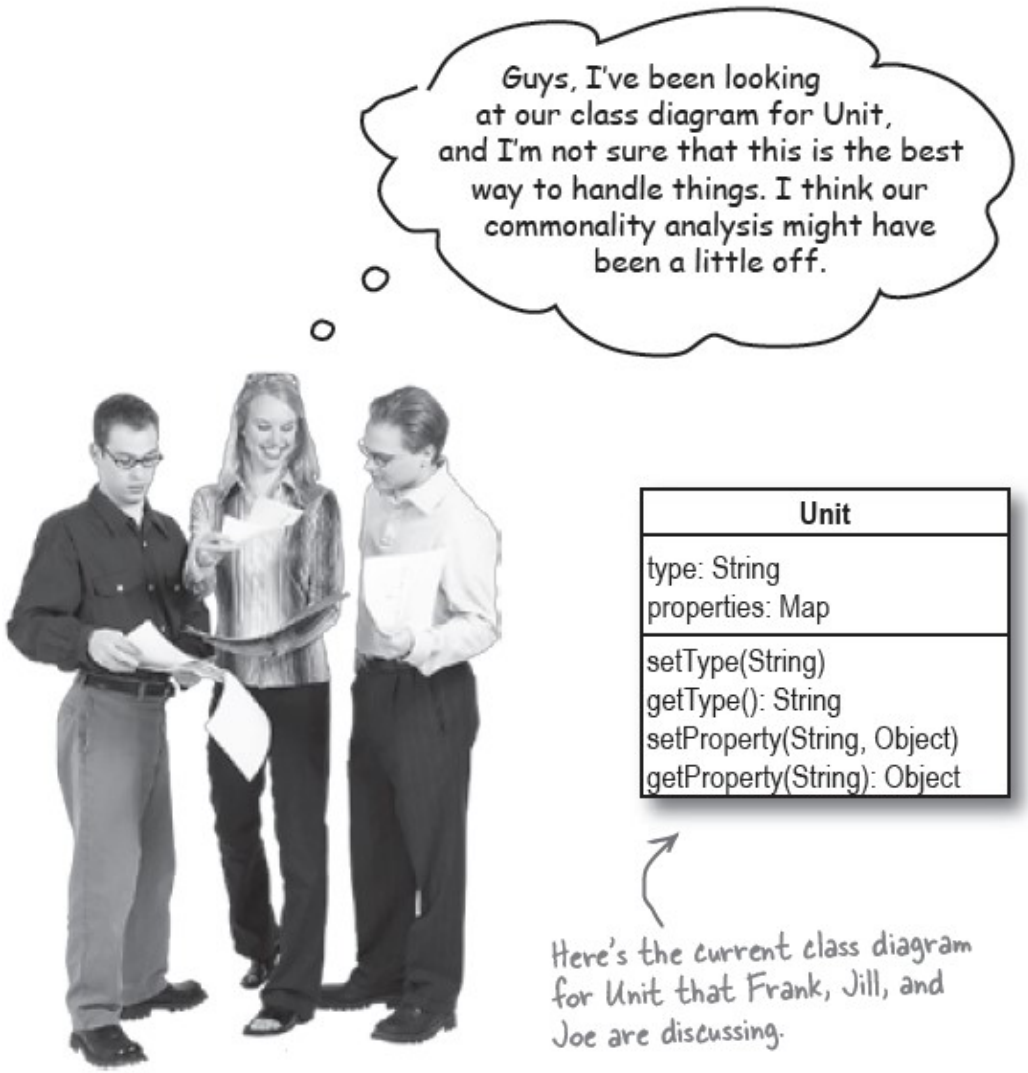
Now let's try and access a property that has no value.

Finally, make sure the Unit still behaves when you ask for a property that does have a value.

```
File Edit Window Help Scenario3
%java UnitTester
Testing the Unit class...
...Created a new unit
...Set "hitPoints" to 25
...Getting unit strength: [no value]
...Getting unit hitPoints: 25

Test complete.
```

Don't forget to test for incorrect usage of the software, too. You'll catch errors early, and make your customers very happy.



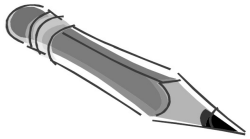
3. Units can be grouped together into armies.

Jill: ...but we also now know that units can be assembled into groups, like armies or fleets or whatever. So what happens if we have two units of the same type in the same group... how can we tell the difference between them?

Frank: OK, but that still doesn't mean we need to change our design. We can just add the `ID` property into our `property Map`. So we've got a nice, uniform way to access all those properties, using the `getProperty()` method.

Jill: But what about commonality? If `ID` is really common to all types of `Unit`, shouldn't it be moved out of the `Map`, sort of like we did with the `type` property?

Joe: Whoa... encapsulation or commonality. That's tough... it seems like we can't do one without screwing up the other.



Sharpen your pencil

Refine your commonality analysis.

Potential Unit Properties

name	weapon	allegiance	gear
type	hitPoints	wingspan	lastName
strength	weight	landSpeed	id
speed	intelligence	experience	groundSpeed
stamina	firstName	weapons	hunger

Anything that you think is generic enough to apply to all units gets written down here.

Commonality

Unit-specific stuff goes on this sheet of paper.

Variability

You can cross out or modify the existing properties of Unit, in addition to adding new stuff.

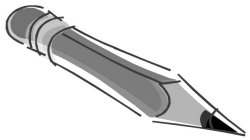
Change and add to the methods in Unit to match what you figured out in the commonality and variability analysis on the last page.

Unit

type: String
properties: Map

Add any new properties you think Unit needs here.

setType(String)
getType(): String
setProperty(String, Object)
getProperty(String): Object



Sharpen your pencil
answers

You should definitely have written down "type", since we figured out that was common to all units back in Chapter 7.

"name" and "id" are pretty basic, and we thought that all units would probably have both properties.

We didn't find a lot of properties that could apply to any type of unit, so our commonality page is pretty thin.

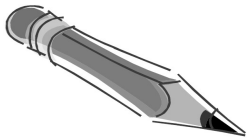
Commonality

- type
- name
- id
- weapons

"weapons" might be a bit of a controversial decision. We figured that since these are war games, all units would have at least one weapon, but that some might have more than one. So a generic "weapons" property seemed a good fit for all unit types.

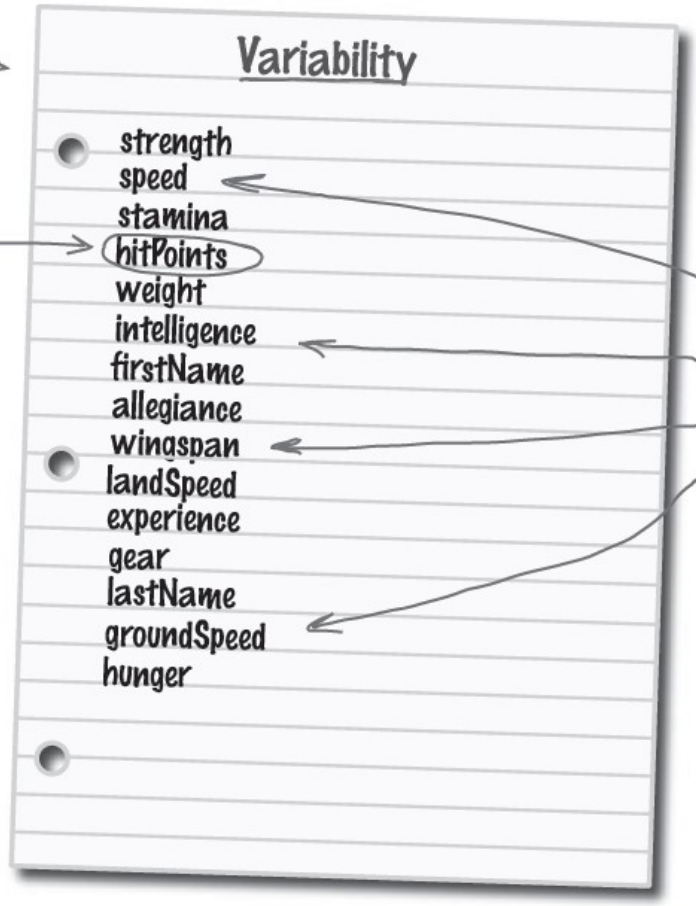
We also decided that we would never need just a single "weapon" property... units with one weapon would have just a single weapon in the "weapons" property. So we ditched this property.

~~weapon~~



Sharpen your pencil
answers

We moved most of the properties onto the Variability list, since they only apply to certain types of units.

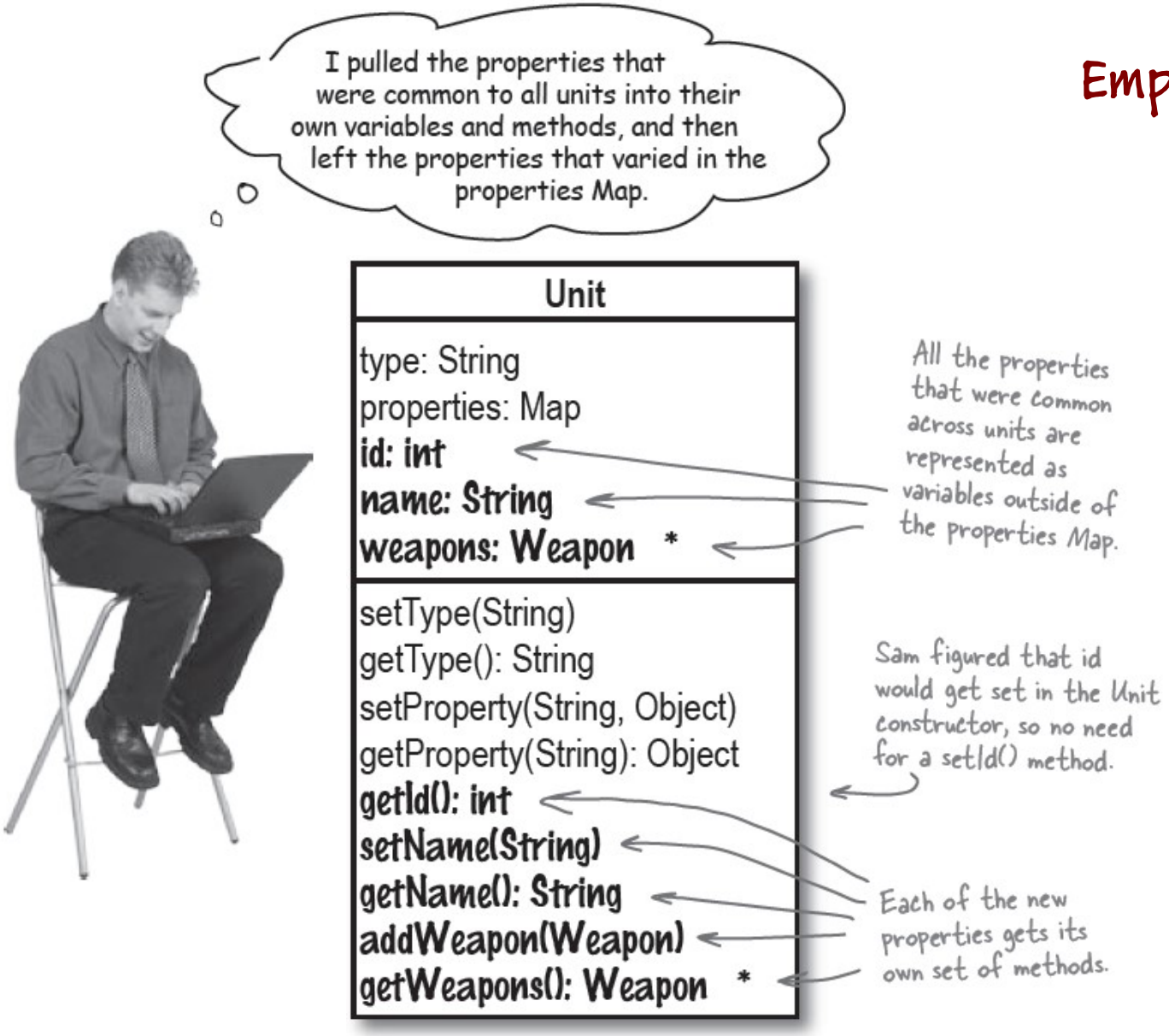


hitPoints was one that could have potentially been common to all units, and gone on the Commonality list. We kept it on Variability, since some object units, like tank or airplane, didn't map as cleanly to hitPoints, which is usually used for units that are human, or at least "alive."

Most of these properties applied to either human-like units, or to vehicular units, but not to both.

You may have a different idea about what makes a good game.

Solution #1: Emphasizing Commonality



In this solution, all game designers can directly access the `id`, `name`, and `weapons` properties, instead of having to use `getProperty()` and work through the more generic properties `Map`.

The emphasis is on keeping the **common** properties of a `Unit` **outside** of the `properties Map`, and leaving properties that **vary inside** the `properties Map`.

You defintely found some commonality between different unit types, but what about good encapsulation? That Unit class doesn't seem very resistant to change, if you ask me.

DESIGN DECISIONS ARE ALWAYS A TRADEOFF

We're repeating ourselves

Now there are two different ways to access properties: through the `getId()`, `getName()`, and property-specific methods, and the `getProperty()` method. Two ways to access properties is almost certainly going to mean duplicate code somewhere.

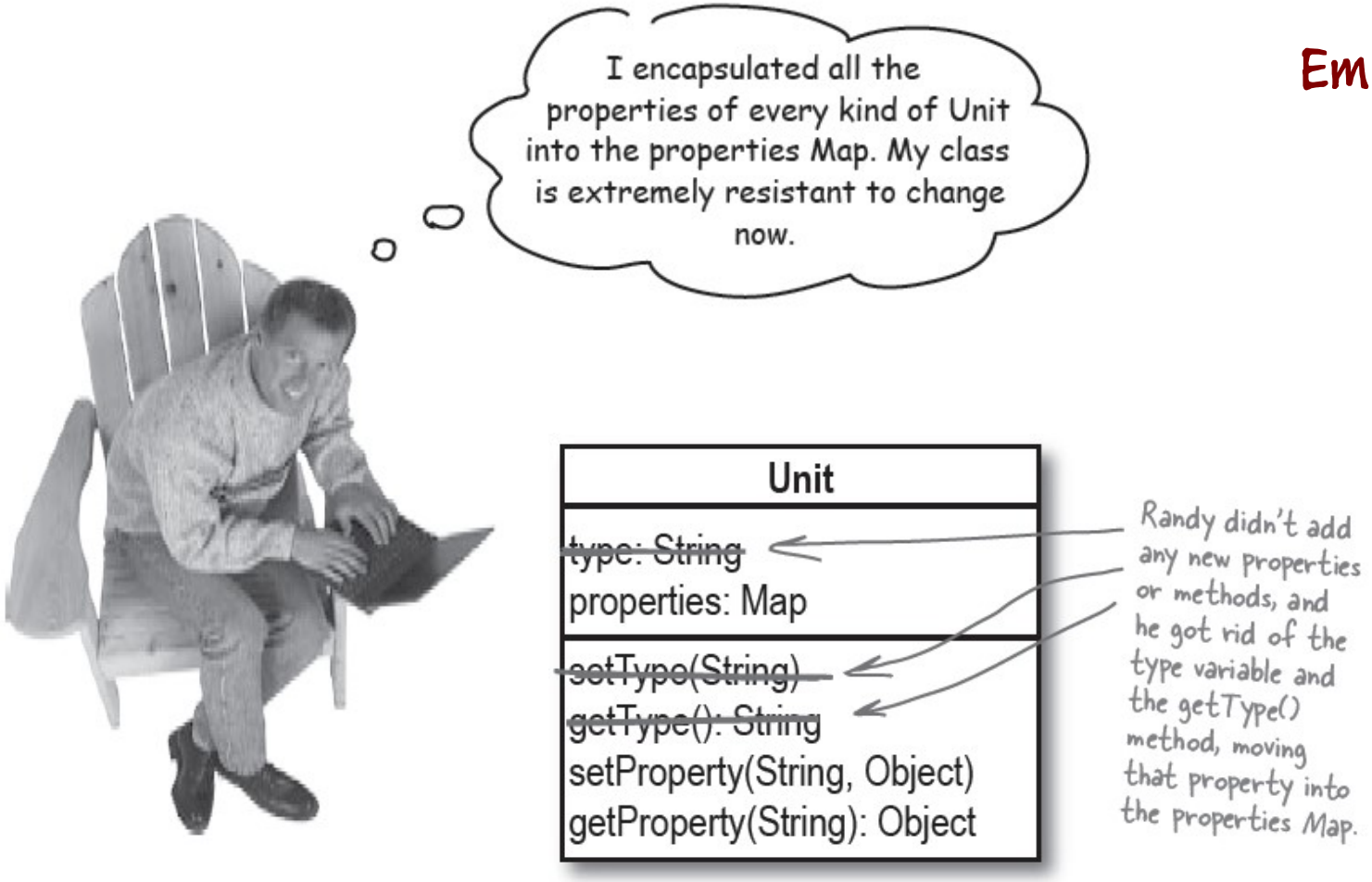
When you see the potential for duplicate code, you'll almost always find maintenance and flexibility issues, as well.

Maintenance is a problem

Hard-coded `id` and `name`

Randy's learned a lot about OO design since we saw him last in Chapter 4.

Solution #2: Emphasizing Encapsulation



This solution focuses on encapsulating **all** the properties for a `Unit` into the `properties Map`, and providing a standard interface—the `getProperty()` method—for accessing all properties. Even properties that apply to all units, like `type` and `id`, are accessed through the `properties Map` in this solution.

The emphasis is on **encapsulation**, and a **flexible design**. Even if the names of common properties change, the Unit class can stay the same, since no property names are hardcoded into the class itself.

? Which developer's solution do you think is best

But you're totally ignoring what's common across Units. And how are game designers going to know that we intended name, type, id, and weapons to be standard properties?

TRADEOFFS WITH THIS DECISION, TOO...

We're ignoring commonality

Randy encapsulated all of the properties into the `properties Map`, but now there's nothing to indicate that `type`, `name`, `id`, and `weapons` are intended to be properties common to all `Unit` types.

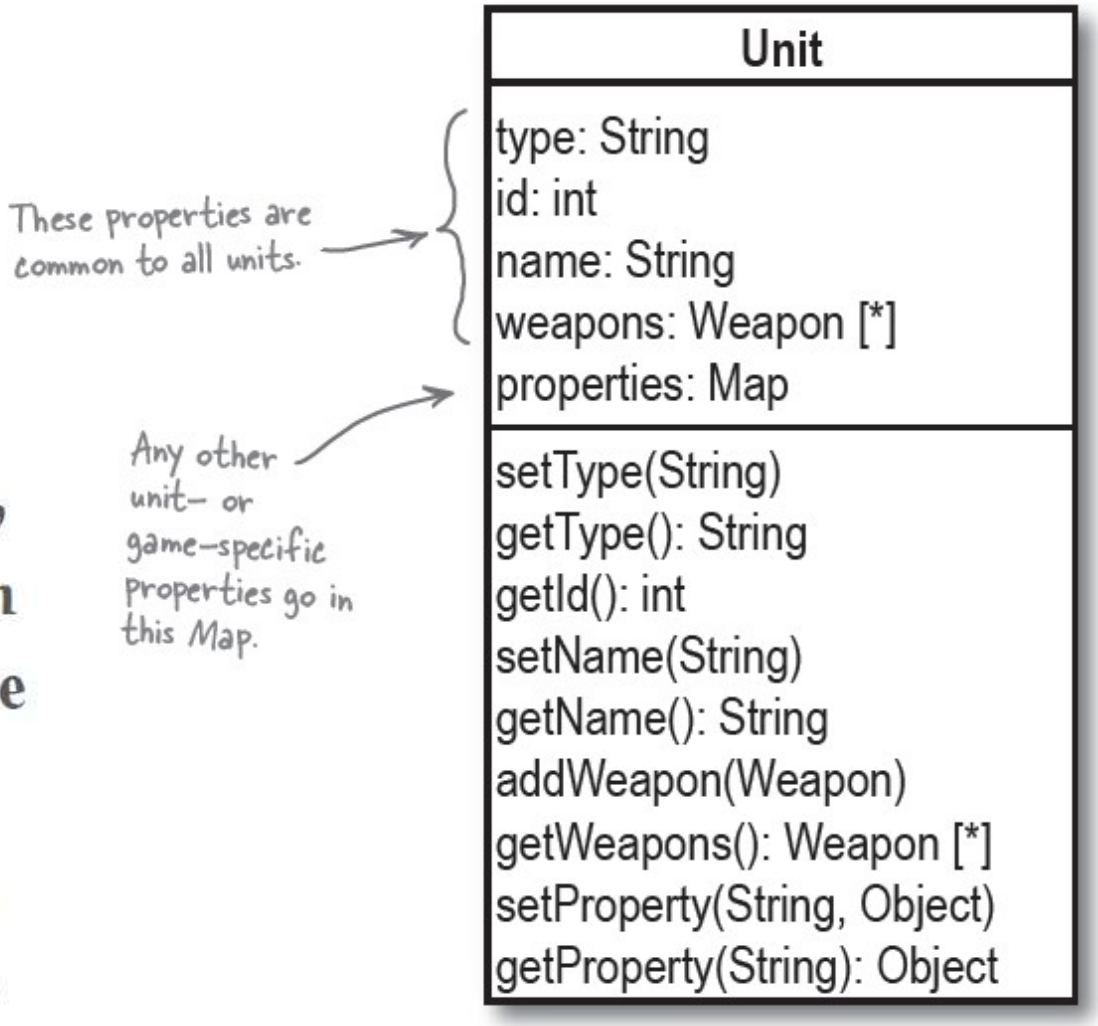
Lots of work at runtime

`getProperty()` returns an `Object`, and you're going to have to cast that into the right value type for each different property, all at runtime. That's a lot of casting, and a lot of extra work that your code has to do at runtime, even for the properties that are common to all `Unit` types.

Good software is built **iteratively**. Analyze, design, and then **iterate** **again**, working on smaller and smaller parts of your app.

Each time you iterate, **reevaluate** your design decisions, and don't be afraid to **CHANGE** something if it makes sense for your design.

Let's go with the commonality-focused solution



Match your tests to your design

Unit properties | Unit movement | Unit groups

Unit
type: String id: int name: String weapons: Weapon [*] properties: Map
setType(String) getType(): String getId(): int setName(String) getName(): String addWeapon(Weapon) getWeapons(): Weapon [*] setProperty(String, Object) getProperty(String): Object

We should have methods in this class to allow us to do everything in all of our tests.

Creating a unit is just calling "new Unit()", so we're all set there.

We can use setType() and getType() to handle this, since it's a common property for all Unit types.

Any properties like hitPoints that aren't common to all units can be set and retrieved using setProperty() and getProperty().

For this test, we just need to call getProperty("strength"), without ever setting the "strength" property, and see what happens.

This tests re-setting a value, which just means another call to setProperty(), so we're covered on this scenario.

```
File Edit Window Help ItWorks
%java UnitTester
Testing the Unit class...
...Created a new unit
...Set type to "infantry"
...Set hitPoints to 25
...Getting unit type: "infantry"
...Getting unit hitPoints: 25

Test
```

```
File Edit Window Help Scenario2
%java UnitTester
Testing the Unit class...
...Created a new unit
...Set hitPoints to 25
...Set hitPoints to 15
...Getting unit hitPoints: 15

Test compl
```

```
File Edit Window Help Scenario3
%java UnitTester
Testing the Unit class...
...Created a new unit
...Set hitPoints to 25
...Getting unit strength: [no value]
...Getting unit hitPoints: 25

Test complete.
```

Let's write the Unit class

Unit.java

```
package headfirst.gsf.unit;
public class Unit {
  private String type;
  private int id;
  private String name;
  private List weapons;
  private Map properties;
  public Unit(int id) {
    this.id = id;
  }
  public int getId() {
    return id;
  }
  // getName() and setName() methods
  // getType() and setType() methods
  public void addWeapon(Weapon weapon) {
    if (weapons == null) {
      weapons = new LinkedList();
    }
    weapons.add(weapon);
  }
  public List getWeapons() {
    return weapons;
  }
  public void setProperty(String property, Object value) {
    if (properties == null) {
      properties = new HashMap();
    }
    properties.put(property, value);
  }
  public Object getProperty(String property) {
    if (properties == null) {
      return null;
    }
    return properties.get(property);
  }
}
```

We take the ID of the Unit in through the constructor...

...so we only need a getId(), and not a setId() as well.

You'll need a simple Weapon class to make this code compile. We created an empty Weapon.java, with no methods, for testing.

We wait until there's a need for a weapons list to instantiate a new List. That saves a little bit of memory, especially when there may be thousands of units.

Just like the weapons List, we don't create a HashMap for properties until it's needed.

Since properties might not be initialized, there's an extra check here before looking up a property's value.

Test cases dissected...

1. Each test case should have an ID and a name.

Try not to refer to tests as test1, test2, etc. Use descriptive names whenever possible.

2. Each test case should have one specific thing that it tests.

One piece of functionality may involve one method, two methods, or even multiple classes... but to start with, focus on very simple pieces of functionality, one at a time.

3. Each test case should have an input you supply.

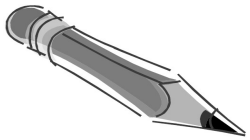
If you're setting hitPoints to 15, then "15" becomes the input you supply to your test case.

4. Each test case should have an output that you expect.

This is what you want the program to output. So if you set type to "infantry", and then call getType(), your expected output is "infantry".

5. Most test cases have a starting state.

There's not much starting state for the Unit class. We do need to create a new Unit, but that's about it.



Sharpen your pencil

Design your test cases.

This test sets the "type" property to a value of "infantry."

ID	What we're testing	Input	Expected Output	Starting State
1	Setting/Getting the type property	"type", "infantry"	"type", "infantry"	Existing Unit object
		"hitPoints", 25		
				Existing Unit object with hitPoints set to 25
4				

Remember, there's a difference between a common property like type, and unit-specific properties.

There are actually 4 individual things being tested in these three scenarios.

The first scenario tests getting and setting the "type" property, as shown here.

File Edit Window Help ItWorks

```
%java UnitTester
Testing the Unit class...
...Created a new unit
...Set type to "infantry"
...Set hitPoints to 25
...Getting unit type: "infantry"
...Getting unit hitPoints: 25

Test complete.
```

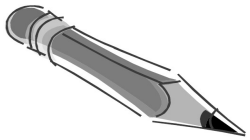
File Edit Window Help Scenario2

```
%java UnitTester
Testing the Unit class...
...Created a new unit
...Set hitPoints to 25
...Set hitPoints to 15
...Getting unit hitPoints: 15
```

File Edit Window Help Scenario3

```
%java UnitTester
Testing the Unit class...
...Created a new unit
...Set hitPoints to 25
...Getting unit strength: [no value]
...Getting unit hitPoints: 25


Test complete.
```



Sharpen your pencil
answers

Design your test cases.

In most of our tests, we want as output exactly what we supplied as input.

ID	What we're testing	Input	Expected Output	Starting State
1	Setting/Getting a common property	"type", "infantry"	"type", "infantry"	Existing Unit object
2	Setting/Getting a unit-specific property	"hitPoints", 25	"hitPoints", 25	Existing Unit object
3	Changing an existing property's value	"hitPoints", 15	"hitPoints", 15	Existing Unit object with hitPoints set to 25
4	Getting a non-existent property's value	N/A	"strength", no value	Existing Unit object without strength value

You should have one test case for working with common properties, and one for working with unit-specific ones.

The entire point of this test is to not supply a value for a property, and then try and retrieve that property's value.

Did you figure out that you needed to make sure there was no property with a previous value for this test case?

Test Puzzle

The problem:

Gary wants to know that you're making progress on supporting units in his game system framework, and you want to be sure that the code you've written for `Unit.java` works properly.

Your task:

1. Create a new class called `UnitTester.java`, and import the `Unit` class and any related classes into it.
2. Add a new method for each test case you figured out from the table on the last slide. Be sure to use descriptive names for the test methods.
3. Each test method should take in an instance of `Unit` with any starting state already set, and any other parameters you think you'll need to run the test and compare an input value with an expected output value.
4. The test method should set the supplied property name and property value on the provided `Unit`, and then retrieve the expected output property value using the expected output property name.
5. If the provided input value and expected output value match, the method should print out "Test passed"; if they don't match, the method should print "Test failed", along with the mismatched values.
6. Write a `main()` method that sets up the starting state for each test, and then runs each test.

Bonus Credit:

Create a new class called `UnitTester.java`, and import the `Unit` class and any related classes into it.

1. There are several things in `Unit.java` that are not being tested by the scenarios on the last slide. Identify what each of these are, and create a test method for each.
2. Run these tests from your `main()` method as well.

Test Puzzle
solutions 1 of 5

Here's the class we wrote to test the Unit class.

```
public class UnitTester {

  public void testType(Unit unit, String type, String expectedOutputType) {
    System.out.println("\nTesting setting/getting the type property.");
    unit.setType(type); String outputType = unit.getType();
    if (expectedOutputType.equals(outputType)) {
      System.out.println("Test passed");
    } else {
      System.out.println("Test failed: " + outputType + " didn't match " +
        expectedOutputType);
    }
  }
```

Each test method has different parameters, since each method is testing different things in the Unit class.

UnitTester.java

Properties stored in the Map take Objects as input and output values.

```
public void testUnitSpecificProperty(Unit unit, String propertyName,
                      Object inputValue, Object expectedOutputValue) {
    System.out.println("\nTesting setting/getting a unit-specific property.");
    unit.setProperty(propertyName, inputValue);
    Object outputValue = unit.getProperty(propertyName);
    if (expectedOutputValue.equals(outputValue)) {
      System.out.println("Test passed");
    } else {
      System.out.println("Test failed: " + outputValue + " didn't match " +
        expectedOutputValue);
    }
  }
```

UnitTester.java

This test is almost identical to test2(), because the starting state takes care of pre-setting the property to another value.

```
public void testChangeProperty(Unit unit, String propertyName,
                Object inputValue, Object expectedOutputValue) {
   System.out.println("\nTesting changing an existing property's value.");
   unit.setProperty(propertyName, inputValue);
   Object outputValue = unit.getProperty(propertyName);
   if (expectedOutputValue.equals(outputValue)) {
     System.out.println("Test passed");
   } else {
     System.out.println("Test failed: " + outputValue + " didn't match " +
       expectedOutputValue);
   }
 }
```

UnitTester.java

This last test doesn't need an input value, because that's what is being tested for: a property without a preset value.

```
public void testNonExistentProperty(Unit unit, String propertyName) {
    System.out.println("\nTesting getting a non-existent property's value.");
    Object outputValue = unit.getProperty(propertyName);
    if (outputValue == null) {
      System.out.println("Test passed");
    } else {
      System.out.println("Test failed with value of " + outputValue);
    }
  }

  public static void main(String args[]) {
    UnitTester tester = new UnitTester();
    Unit unit = new Unit(1000);
    tester.testType(unit, "infantry", "infantry");
    tester.testUnitSpecificProperty(unit, "hitPoints",
                                    new Integer(25), new Integer(25));
    tester.testChangeProperty(unit, "hitPoints",
                                  new Integer(15), new Integer(15));
    tester.testNonExistentProperty(unit, "strength");
  }
}
```

All our main() method needs to do is create a new Unit, and then run through the tests.

UnitTester.java

And now for some bonus credit...

This test case doesn't test all common properties; it just tests the type property.

ID	What we're testing	Input	Expected Output	Starting State
1	Setting/Getting the type property	"type", "infantry"	"type", "infantry"	Existing Unit object
2	Setting/Getting a unit-specific property	"hitPoints", 25	"hitPoints", 25	Existing Unit object
3	Changing an existing property's value	"hitPoints", 15	"hitPoints", 15	Existing Unit object with hitPoints set to 25
4	Getting a non-existent property's value	N/A	"strength", no value	Existing Unit object without strength value
5	Getting the id property	N/A	1000	Existing Unit object with an id of 1000
6	Setting/getting the name property	"name", "Damon"	"name", "Damon"	Existing Unit object
7	Adding/getting weapons	Axe object	Axe object	Existing Unit object

We need to test all three of the other common properties, since each uses its own specific methods.

Prove yourself to the customer

Test classes aren't supposed to be exciting or sexy... they just need to prove that your software does what it's supposed to do.

```
File Edit Window Help ProveToMe
%java UnitTester

Testing setting/getting the type property.
Test passed

Testing setting/getting a unit-specific
property.
Test passed

Testing changing an existing property's
value.
Test passed

Testing getting a non-existent property's
value.
Test passed
```

This is great! You really do know what you're doing. I'll put a check in the mail, and you just keep on with the Unit class. Do you have units moving around the board yet?

Customers that see running code tend to get happy, and keep paying you. Customers that only see diagrams get impatient and frustrated, so don't expect much support or cash.

When you program by contract, you and your software's users are agreeing that your software will behave in a certain way.

Let's change the programming contract for the game system

Meet Sue. She manages a team of top-notch game developers, and they're interested in using Gary's game framework.

We've been programming by contract so far

You probably didn't notice, but we've been doing something called *programming by contract* in the `Unit` class so far. In the `Unit` class, if someone asks for a property that doesn't exist, we've just returned null. We've been doing the same thing in `getWeapons();` if the `weapons` list isn't initialized, we just return null there, too

This list might not be initialized, so this could return null if there aren't any weapons for this unit.

If there aren't any properties, we return null...

...and if there isn't a value for the requested property, this will return null.

```
public List getWeapons() {
  return weapons;
}
// other methods
public Object getProperty(String property) {
  if (properties == null) {
    return null;
  }
  return properties.get(property);
}
```

Unit.java

Even though you didn't know it, this code is defining a contract for what happens when a property doesn't exist.

When you are **programming by contract**, you're working with client code to **agree** on how you'll handle problem situations.

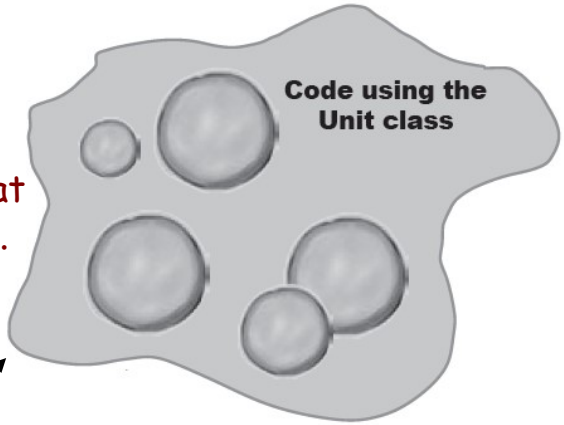
This is the contract for Unit

The `Unit` class assumes that people using it are competent programmers, and that they can handle null return values. So our contract states something like this:

Hey, you look pretty smart. I'm gonna return null if you ask for properties or weapons that don't exist. You can handle the null values, OK?

This is our contract... it states what we'll do in a certain situation.

Look, we know what we're doing. Our code will only ask you for properties that exist. So just return null... trust us to do the right thing, OK?



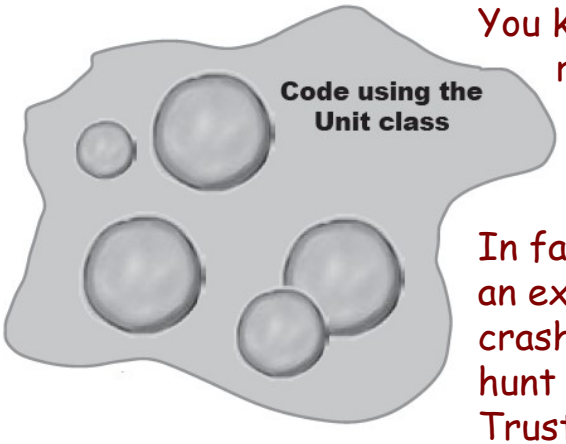
Hey, you look pretty smart. I'm gonna return null if you ask for properties or weapons that don't exist. You can handle the null values, OK?

Programming by contract is really all about trust

When you return null, you're **trusting** programmers to be able to deal with null return values.

And we can always change the contract if we need to...

We were asked to stop returning null, and throw an exception instead. This really isn't a big change to the contract; it just means that now game designers are going to have big problems if they ask for non-existent properties or weapons..



You know what? We're really confident we're not going to ask you for non-existent properties.

In fact, if we do, just throw an exception, and it will crash the program, and we'll hunt down the bug. Trust us... throwing an exception is no problem.

Sure. As long as you know I'm going to start throwing an Exception, we're good to go. I'll just change my code, and we'll start using this new contract.

But if you don't trust your users...

Suppose you were really worried that game designers using the Unit class, and asking for non-existent properties, were getting null values and not handling them properly. You might rewrite the `getProperty()` method like this:

I'm sure you're great code and all, but I just don't trust you. I could send you null, and you could totally blow up. So let's just be safe, and I'll send you a checked exception that you'll have to catch, to make sure you don't get a null value back and do something stupid with it.

Defensive programming assumes the worst, and tries to protect itself (and you) against misuse or bad data.

```
public Object getProperty(String property)
  throws IllegalAccessException {
  if (properties == null) {
    return null;
    throw new IllegalAccessException(
      "What are you doing? No properties!");
  }
  return properties.get(property);
  Object value = properties.get(property);
  if (value == null) {
    throw new IllegalAccessException(
      "You're screwing up! No property value.");
  } else {
    return value;
  }
}
```

This version of getProperty() can throw a CHECKED exception, so code using Unit will have to catch this exception.

We don't return null anymore... we make a BIG deal about asking for a non-existent property.

Unit.java

This is a defensive version of Unit.java.

-or if they don't trust you...

Of course, when programmers use your code, they might not trust you either... they can program defensively as well. What if they don't believe that you'll only return non-null values from `getProperty()`? Then they're going to protect their code, and use defensive programming, as well:

Here's a sample of code that uses the Unit class.

```
// Some method goes out and gets a unit
Unit unit = getUnit();
// Now let's use the unit...
String name = unit.getName();
if ((name != null) && (name.length() > 0)) {
  System.out.println("Unit name: " + name);
}
Object value = unit.getProperty("hitPoints");
if (value != null) {
  try {
    Integer hitPoints = (Integer)value;
  } catch (ClassCastException e) {
    // Handle the potential error
  }
}
// etc...
```

This code does a LOT of error checking... it doesn't ever trust Unit to return valid data.

This code is written extremely defensively.

When you're **programming defensively**, you're making sure the client gets a "safe" response, **no matter what** the client wants to have happen.

Who Am I?

+ Feature Driven Development
+ Use Case Driven Development
+ Programming by Contract
+ Defensive Programming

I'm very well-ordered. I prefer to take things one step at a time, until I've made it from start to finish.

Use Case Driven Development

Well, sure, she said she would call, but how can you really believe anyone anymore?

Defensive Programming

Oh, absolutely, requirements really get me motivated.

Feature Driven Development, Use Case Driven Development

You might have added Programming by Contract here, since a contract is really a form of requirements.

I'm very well-behaved. In fact, I've been focusing on all of my own behavior before moving on to anything else.

Feature Driven Development

Really, it's all about my customer. I just want to satisfy them, after all.

All of them!

All of these techniques and tools are really about getting the customer the software that they want.

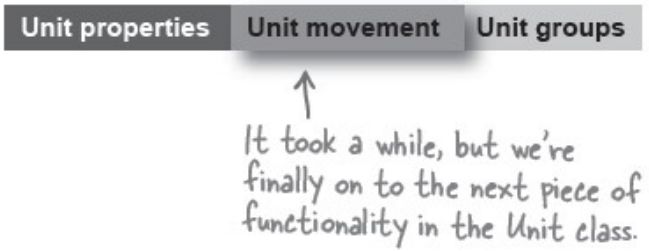
Hey, you're a big boy. You can deal with that on your own... it's really not my problem anymore, is it?

Programming by Contract

As long as you're good with it, so am I. Who am I to tell you what to do, so long as you know what you can expect from me.

Programming by Contract

Moving units



1. Each unit should have properties, and game designers can add new properties to unit types in their own games. ✓

2. Units have to be able to move from one tile on a board to another.

We're on to dealing with movement, now.

3. Units can be grouped together into armies.

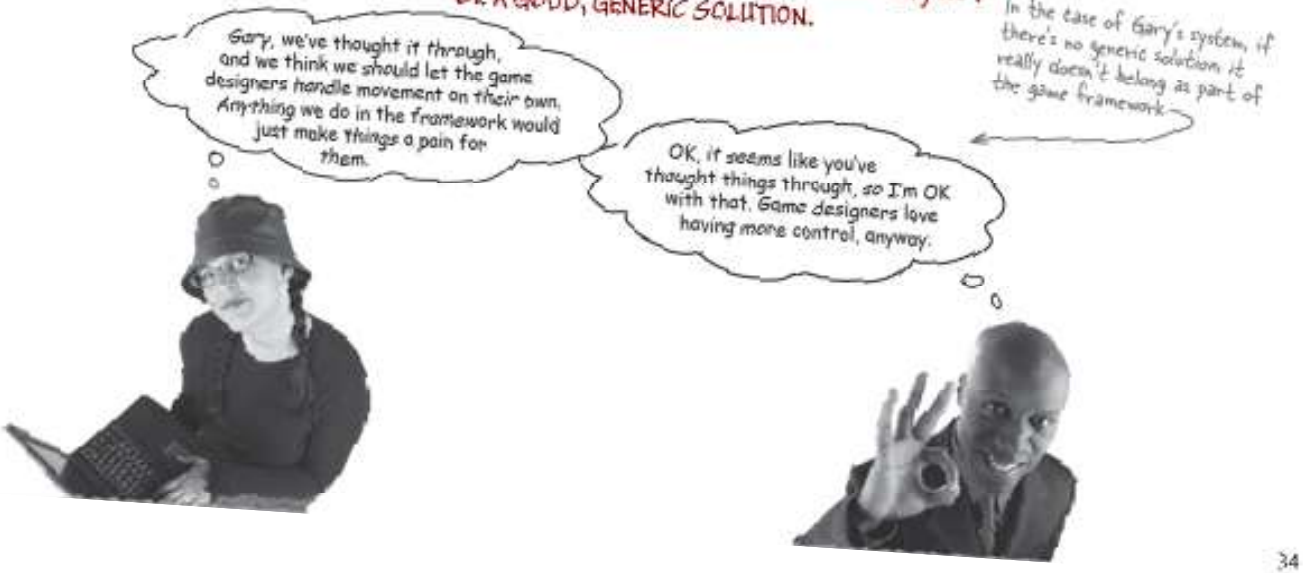
Haven't we been here before?

Back in Chapter 7, we decided that handling movement was different for every game.

It's "different for every game"

Did you see what kept showing up in our chart? Every time we found some commonality, the variability column had the same words: "different for every game."

WHEN YOU FIND MORE THINGS THAT ARE DIFFERENT ABOUT A FEATURE THAN THINGS THAT ARE THE SAME, THERE MAY NOT BE A GOOD, GENERIC SOLUTION.



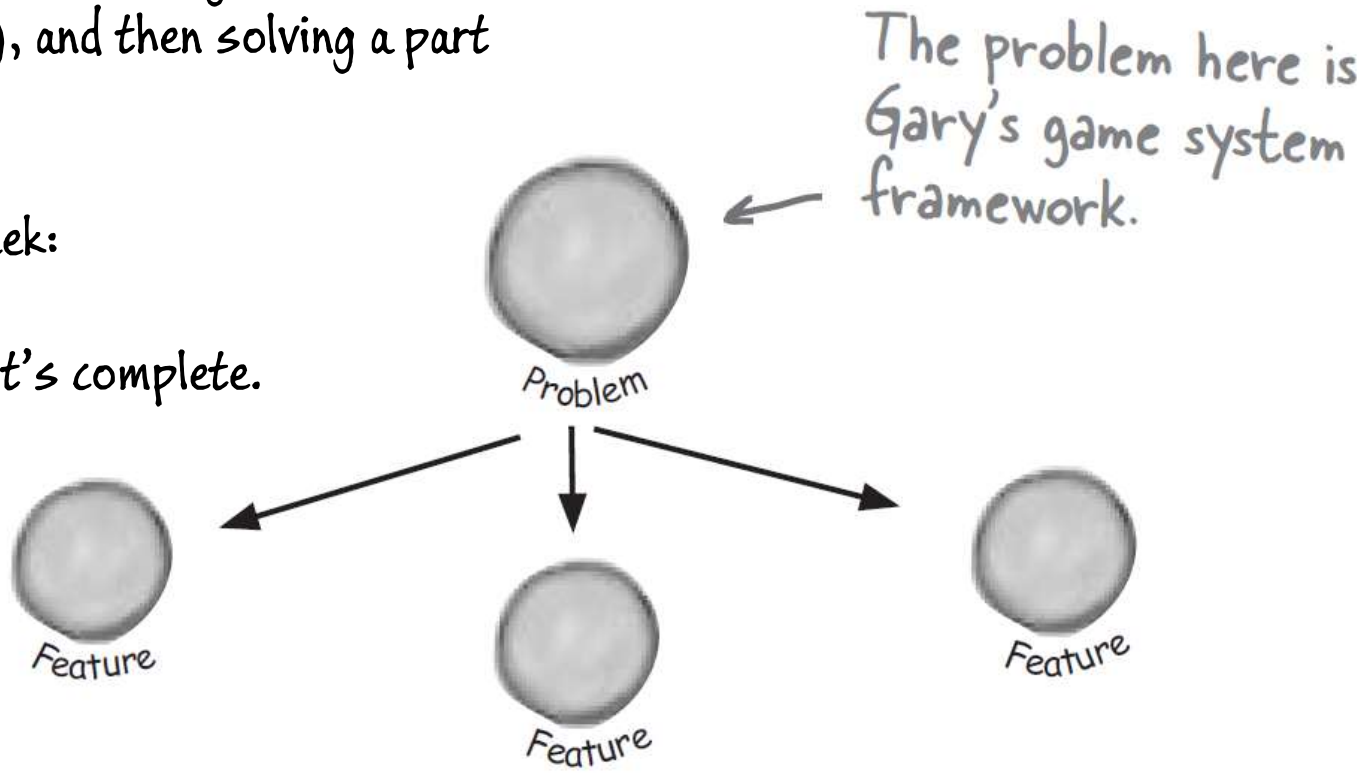
Break your apps up into smaller chunks of functionality

We've been talking a lot about iterating deeper into your application, and at each stage, doing more analysis and design.

So you're taking each problem, and then breaking it up (either into use cases or features), and then solving a part of the problem, over and over.

This is what we've been doing this week:

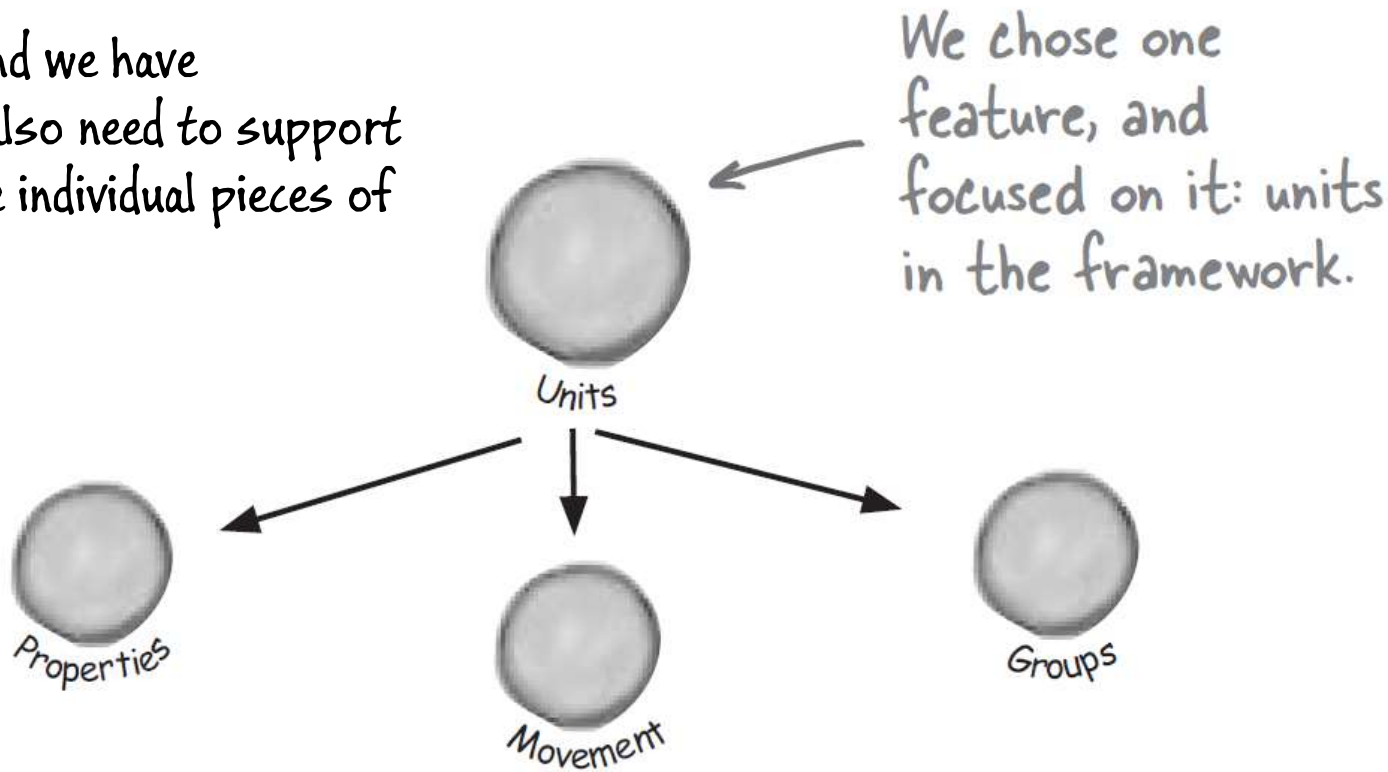
- taking a single feature,
- working on that feature until it's complete.



But you can still break things up further...

But once you choose a single feature or use case, you can usually break that feature up into even smaller pieces of behavior.

For example, a unit has properties and we have to deal with unit movement. And we also need to support groupings of units. So each of these individual pieces of behavior has to be dealt with.



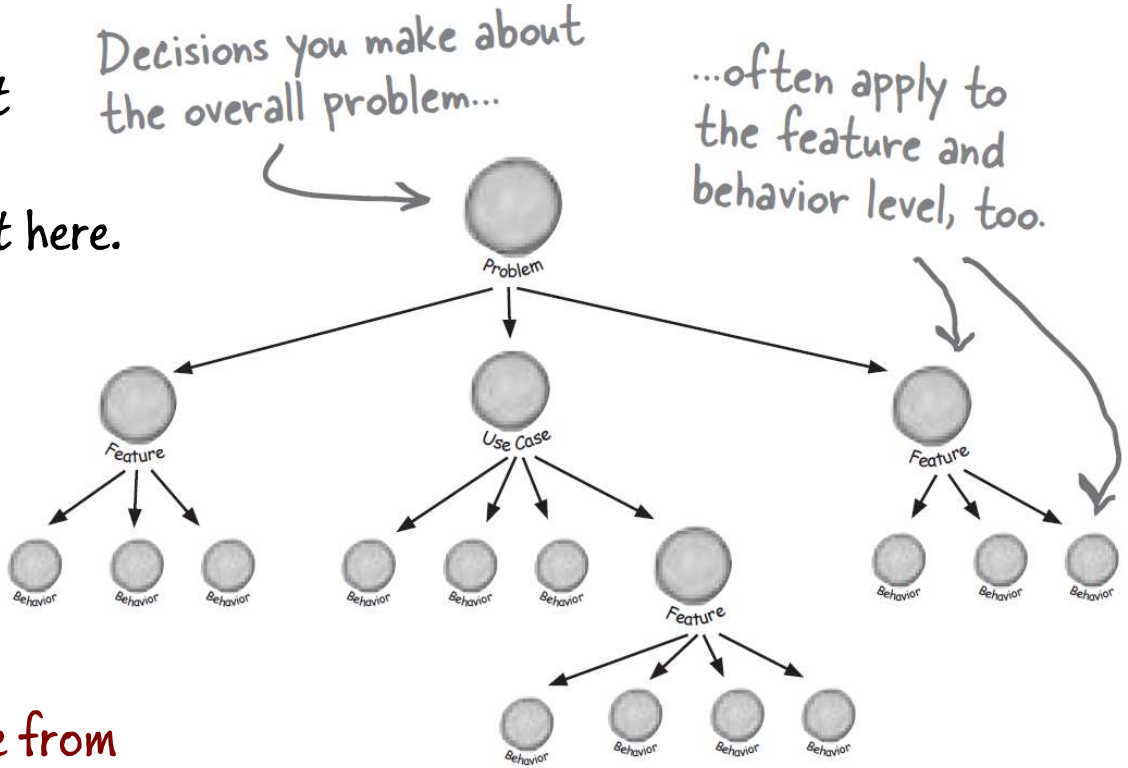
Your decisions can iterate down, too

Lots of times you'll find that decisions you made earlier save you work down the line.

In Gary's system, we decided that game designers would deal with movement on their own. So now that we're talking about how to handle unit movement, we can take that decision we made earlier, and apply it here.

Since it still seems sensible—there's no reason to change that decision—we can have game designers worry about handling unit movement, and move on to the next piece of behavior.

Just add a note to your docs for the game designers that movement is up to them.



2. Units have to be able to move from one tile on a board to another. ✓

Feature Puzzle

Unit properties | Unit movement | Unit groups

Solve this puzzle, and you've completed the behavior for the Unit feature.

By now, you should have feature driven development, iteration, analysis, and design down pretty solid. We're going to leave it up to you to handle the last bit of behavior and finish off the Unit feature of Gary's game system framework.

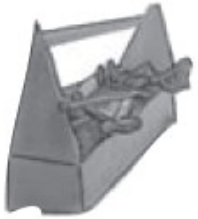
The problem:

Gary's framework needs to support groups of units.

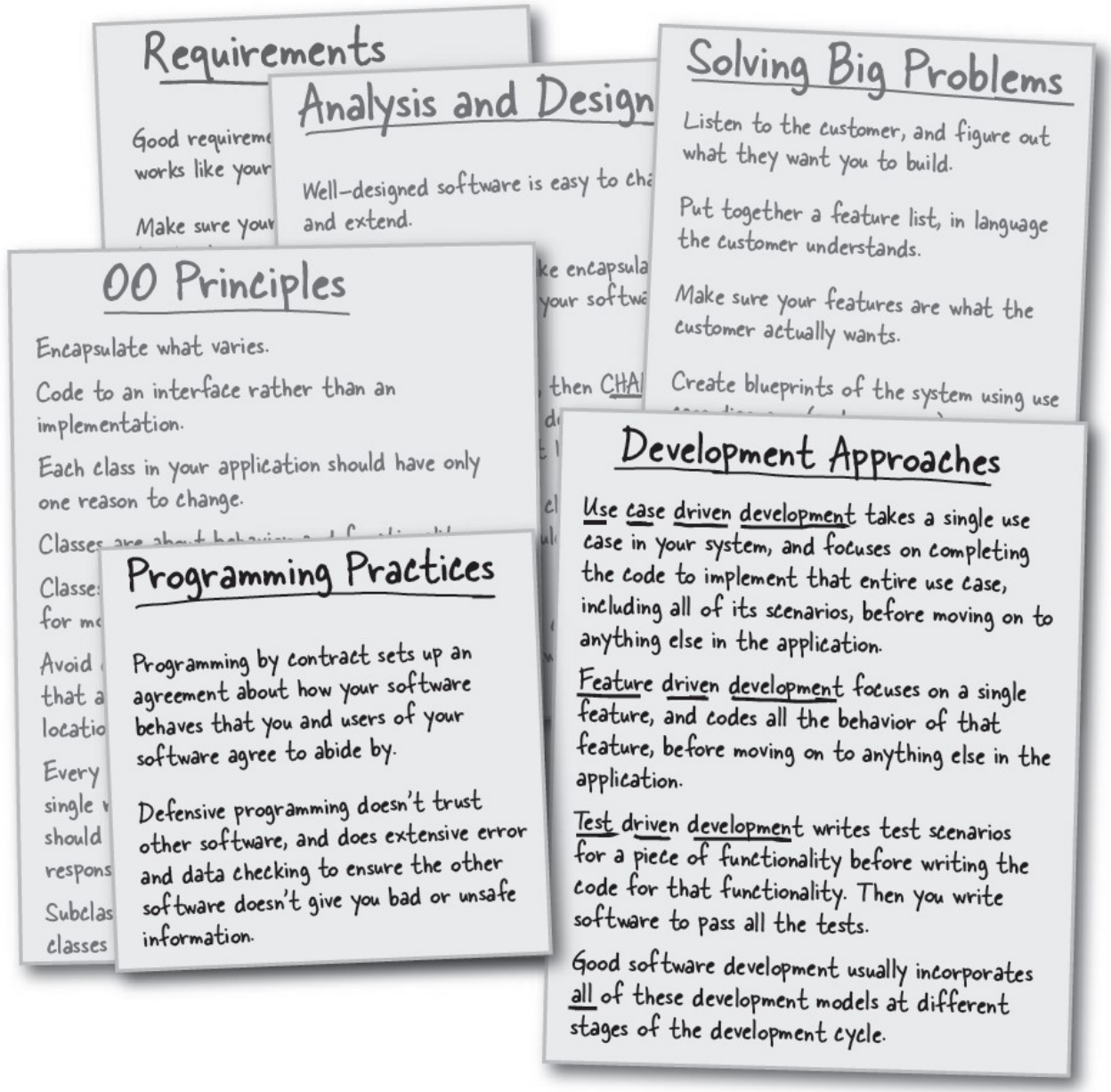
Your task:

1. Create a new class that can group units together, and both add and remove units to the group.
2. Fill out the table below with test case scenarios that will test your software, and prove to Gary that the grouping of units works.
3. Add methods to `UnitTester` to implement the test scenarios in the table, and make sure all your tests pass.

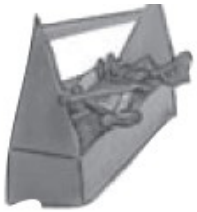
ID	What we're testing	Input	Expected Output	Starting State



Tools for your toolbox



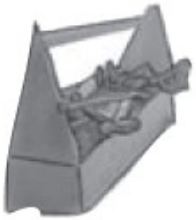
Tools for your toolbox



Bullet Points

- The first step in writing good software is to make sure your application works like the customer expects and wants it to.
- Customers don't usually care about diagrams and lists; they want to see your software actually do something.
- Use case driven development focuses on one scenario in a use case in your application at a time.
- In use case driven development, you focus on a single scenario at a time, but you also usually code all the scenarios in a single use case before moving on to any other scenarios, in other use cases.
- Feature driven development allows you to code a complete feature before moving on to anything else.
- You can choose to work on either big or small features in feature-driven development, as long as you take each feature one at a time.
- Software development is always iterative. You look at the big picture, and then iterate down to smaller pieces of functionality.
- You have to do analysis and design at each step of your development cycle, including when you start working on a new feature or use case.

Tools for your toolbox



Bullet Points

- Tests allow you to make sure your sottware doesn't have any bugs, and let you prove to your customer that your software works.
- A good test case only tests one specific piece of functionality.
- Test cases may involve only one, or several, methods in a single class, or may involve multiple classes.
- Test driven development is based on the idea that you write your tests first, and then develop software that passes those tests. The result is fully functional, working software.
- Programming by contract assumes both sides in a transaction understand what actions generate what behavior, and will abide by that contract.
- Methods usually return null or unchecked exceptions when errors occur in programming by contract environments.
- Defensive programming looks for things to go wrong, and tests extensively to avoid problem situations.
- Methods usually return "empty" objects or throw checked exceptions in defensive programming environments.