CSE203 Principles of Software Design and Development

I was just thinking... do you remember if we ever tightened the bolts down on those basement girders? Oh well...

06. Solving Really big Problems

"My Name is Art Vandelay...
I am an Architect"

1

Look, all this stuff about writing great software sounds terrific, but *real* applications have a lot more than five or ten classes. How am I supposed to turn **big applications** into great software?

Remember these steps to writing great software? They all apply to working with huge, 1000+ class applications just as much as when you're working with just a couple of classes.

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.

You solve BIG PROBLEMS the same way you solve small problems

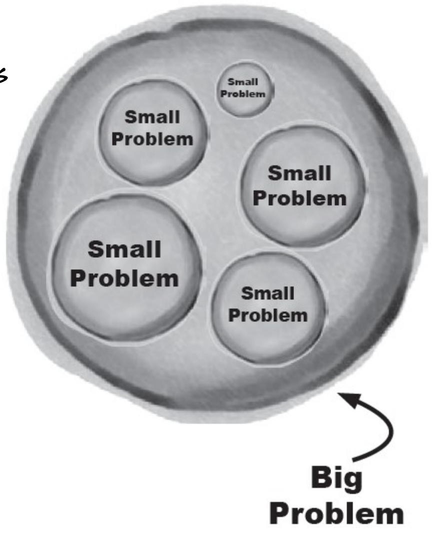
2

It's all in how you look at the BIG PROBLEM

The best way to look at a big problem is to see it as lots of individual pieces of functionality.

You can treat each of those pieces as an individual problem to solve, and apply the things you already know.

This BIG PROBLEM is really just a collection of functionalities, where each piece of functionality is really a smaller problem on its own.



You can solve a big problem by breaking it into lots of functional pieces, and then working on each of those pieces individually.

3

3

The things you already know...

By encapsulating what varies, you make your application more flexible, and easier to change.

Using encapsulation helps with big problems, too. The more you encapsulate things, the easier it will be for you to break a large app up into different pieces of functionality.

Coding to an interface, rather than to an implementation, makes your software easier to extend.

This is even more important in big apps. By coding to an interface, you reduce dependencies between different parts of your application... and "loosely coupled" is always a good thing, remember?

The best way to get good requirements is to understand what a system is supposed to do.

If you know what each small piece of your app's functionality should do, then it's easy to combine those parts into a big app that does what it's supposed to do.

4

4

This sure doesn't change with bigger problems. In fact, the higher the cohesion of your app, the more independent each piece of functionality is, and the easier it is to work on those pieces one at a time.

SO LET'S SOLVE A BIG PROBLEM!

Great software is easy to change and extend, and does what the customer wants it to do.

Got a big problem? Take a few of these little principles, and call me in the morning. I bet you'll have things under control in no time.

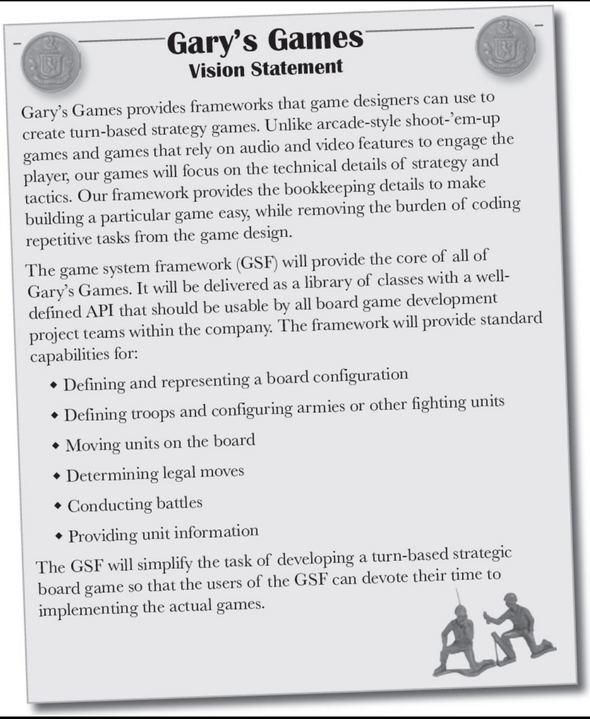
Analysis is even more important with large software... and in most cases, you start by analyzing individual pieces of functionality, and then analyzing the interaction of those pieces.

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

5

5

Here's the big problem we're going to be working on for the next few weeks.



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.

6

6

Sharpen your pencil

What should we do first?

Below are several things that you might start out doing to get going on Gary's Games. Check the boxes next to the things you think we should start with.

Talk to Gary.

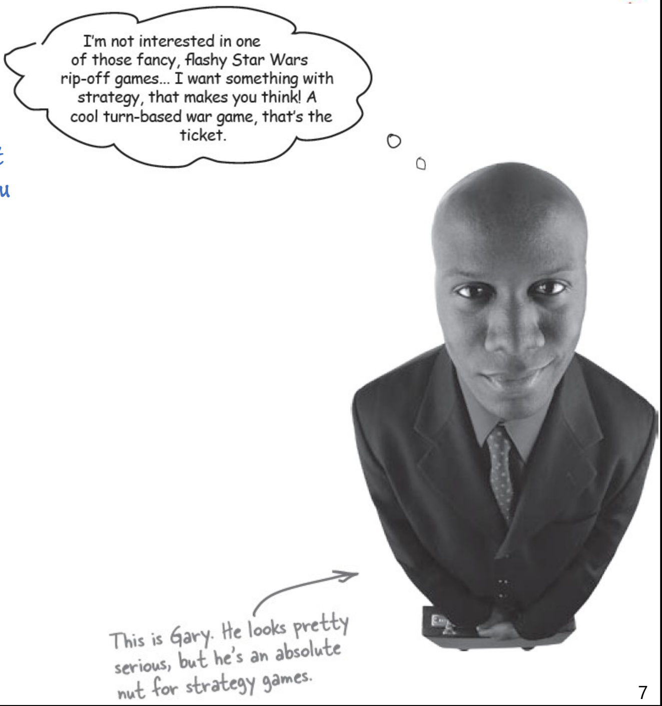
Gather requirements.

Start a class diagram.

Talk to people who might use the framework.

Write use cases.

Start a package diagram.


I'm not interested in one of those fancy, flashy Star Wars rip-off games... I want something with strategy, that makes you think! A cool turn-based war game, that's the ticket.
This is Gary. He looks pretty serious, but he's an absolute nut for strategy games.

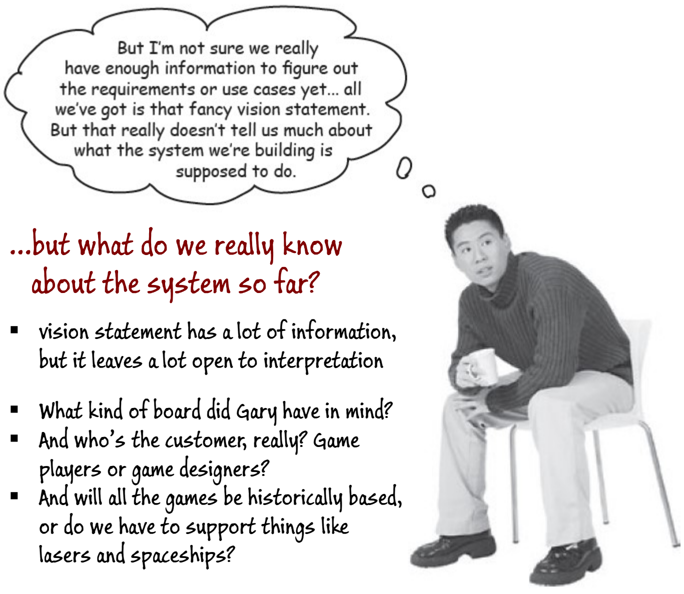
7

7

Hey, this is an easy one. We start out by writing out the requirements and use cases, like we did with Doug's Dog Doors.
One of the programmers on your team.

Requirements and use cases are a good place to start...

- figure out what a system is supposed to do
- adding functionality bit by bit


But I'm not sure we really have enough information to figure out the requirements or use cases yet... all we've got is that fancy vision statement. But that really doesn't tell us much about what the system we're building is supposed to do.

...but what do we really know about the system so far?

- vision statement has a lot of information, but it leaves a lot open to interpretation
- What kind of board did Gary have in mind?
- And who's the customer, really? Game players or game designers?
- And will all the games be historically based, or do we have to support things like lasers and spaceships?

8

8

We need a lot more information

We've got to figure out what the system is supposed to do.
So how do we do *that*?

This is called **commonality**... what things are similar?

This is called **variability**... what things are different?

What is the system like?

Are there some things that you do know about that the system functions or behaves like?

What is the system not like?

determine what you don't need to worry about in your system.

So let's listen in on one of Gary's meetings, and see what we can find out...

9

9

Customer Conversation

We've already found some commonality! The system has an interface sort of like this Zork game.

Remember that old computer game, Zork? Everybody loved that thing, even though it was pure text.
Bob in marketing.
Bethany in design.
Susan and Tom in sales.

Tom: Yeah, Gary loves text-based games. And people are getting a little tired of all the fancy graphics in games like Star Wars episode 206 (or whatever the heck they're up to these days).

Here's some variability. The system is not a graphic-rich game.

Bethany: And we need all sorts of different time periods. We could have a Civil War version, with battles at Antietam and Vicksburg, and a World War I version over in Europe... players will love all the historical stuff, I'll bet.

Susan: Nice idea, Beth! I'll bet we can let game designers create add-on packs, too, so you could buy a World War II: Allies game, and then buy an add-on for other forces that the core game didn't include.

Bob: That's a cool marketing point, too... if our system supports different time periods, unit types, uniforms, and offensives, we're going to be able to sell this to almost anyone developing games.

Flexbililty is going to be key if we're going to support all these variations.

Bethany: Do you think we need to worry about battles that aren't historical? I mean, we could sell our system to the folks that make the fancy starship games, and let them create sci-fi battles, right?

10

10

Tom: Hmmm... I'll bet Gary would go for that, if they're still creating turn-based games. Why not clean up on that market as well as the history buffs?

Bob: Do you think we could market this as a system to create everything from online Risk to a modern-day Stratego? Those were both killer strategy board games back in the day... I'd love to sell our system to people that make those sorts of games.

A little more commonality... so we're really aiming at turn-based wargames.

Bethany: So let's talk details. We know we've got to sell this to lots of game designers, so we need it to be really flexible. I'm thinking we start with a nice square board, and fill it up with square tiles.

Tom: We can let the game designers pick how many tiles on the board, right? They can choose a height and width, or something like that?

OK, now we're starting to get some ideas about actual features of the game system.

Bethany: Yeah. And then we should support all different types of terrains: mountains, rivers, plains, grass...

Susan: ...maybe space or craters or asteroid or something for the space games...

Bob: Even underwater tiles, like seaweed or silt or something, right?

Bethany: Those are great ideas! So we just need a basic tile that can be customized and extended, and a board that we can fill with all the different tiles.

Susan: Do we have to worry about all those movement rules and things that these games usually have?

Tom: I think we have to, don't we? Don't most of these strategy games have all sorts of complicated rules, like a unit can only move so many tiles because he's carrying too much weight, or whatever?

Strategy games again... we definitely have some commonality with that type of game to pay attention to.

Bethany: I think most of the rules depend on the specific game, though. I think we should leave that up to the game designers who use our framework. All our framework should do is keep track of whose turn it is to move, and handle basic movement stuff.

Susan: This is great. We can build a framework for challenging, fun strategy games, and make a ton of money, too.

Bob: This is starting to sound pretty cool! Let's get this to Gary and those software guys he's hired, so they can get started.

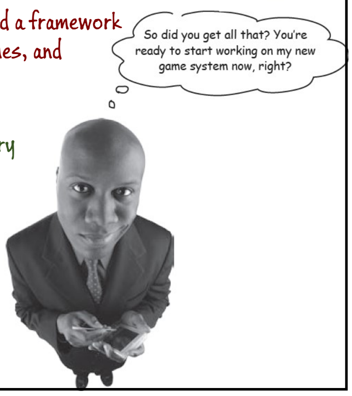

So did you get all that? You're ready to start working on my new game system now, right?

Figure out the features

Let's take that information and figure out the ***features*** of the system.

Bethany said the game system should support different time periods. That's a feature of the game system.

Bethany: And we need all sorts of <u>different time periods</u>. We could have a Civil War version, with battles at Antietam and Vicksburg, and a World War I version over in Europe... players will love all the historical stuff, I'll bet.

Here's another feature: different types of terrain. This single feature will probably create several individual requirements.

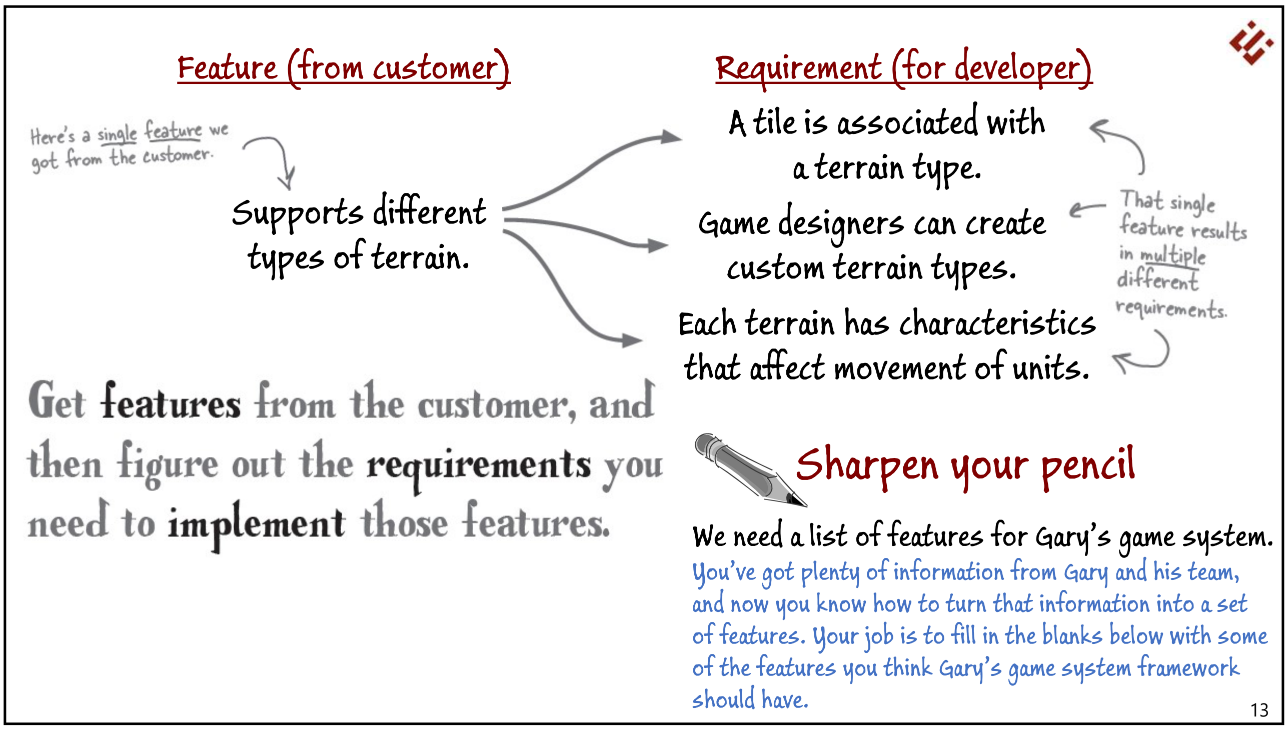
Bethany: Yeah. And then we should support all <u>different types of terrains</u>: mountains, rivers, plains, grass...
Susan: ...maybe space or craters or asteroid or something for the space games...
Bob: Even underwater tiles, like seaweed or silt or something, right?

But what is a feature, anyway?

- A feature is just a *high-level description* of something a system needs to do. You usually get features from talking to your customers
- A lot of times, you can take one feature, and come up with several different requirements that you can use to satisfy that feature.
- So figuring out a system's features is a great way to start to get a handle on your requirements.

Starting with the features of a system is really helpful in big projects—like Gary's game system—when you don't have tons of details, and just need to get a handle on where to start.

12



Feature (from customer) / Requirement (for developer)

Here's a single feature we got from the customer.

Supports different types of terrain.

- A tile is associated with a terrain type.
- Game designers can create custom terrain types.
- Each terrain has characteristics that affect movement of units.

That single feature results in multiple different requirements.

Get features from the customer, and then figure out the requirements you need to implement those features.

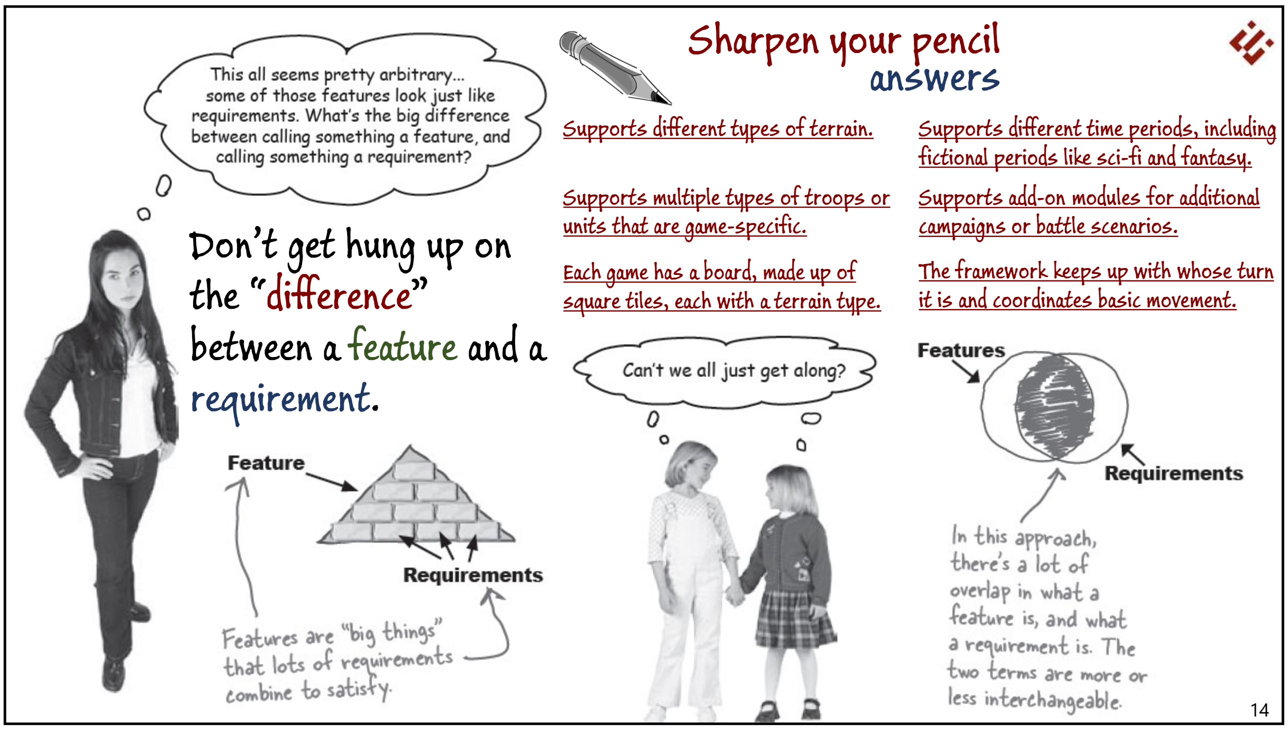
Sharpen your pencil

We need a list of features for Gary's game system.

You've got plenty of information from Gary and his team, and now you know how to turn that information into a set of features. Your job is to fill in the blanks below with some of the features you think Gary's game system framework should have.

13

13



Sharpen your pencil answers

Supports different types of terrain.

Supports different time periods, including fictional periods like sci-fi and fantasy.

Supports multiple types of troops or units that are game-specific.

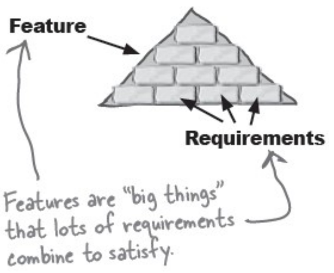
Supports add-on modules for additional campaigns or battle scenarios.

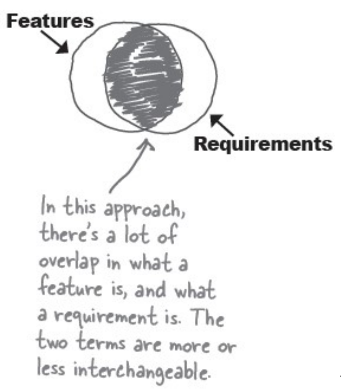
Each game has a board, made up of square tiles, each with a terrain type.

The framework keeps up with whose turn it is and coordinates basic movement.

This all seems pretty arbitrary... some of those features look just like requirements. What's the big difference between calling something a feature, and calling something a requirement?

Don't get hung up on the "difference" between a feature and a requirement.


Feature
Requirements
Features are "big things" that lots of requirements combine to satisfy.


Features
Requirements
In this approach, there's a lot of overlap in what a feature is, and what a requirement is. The two terms are more or less interchangeable.

14

14

OK, so we've got the feature and requirement thing figured out. **Now** we can write some use cases, right?

Use cases don't always help you see the big picture.

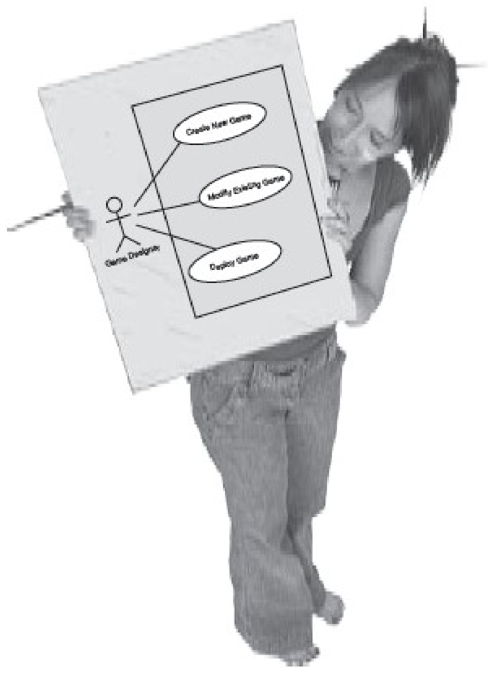
When you're working on a system, it's a good idea to defer details as long as you can...
You won't get caught up in the *little things* when you should be working on the *big things*.

Always defer details as long as you can.

So what **are** we supposed to do now? You've been telling us we need to know what the system is supposed to do for like 6 weeks now, and suddenly use cases aren't a good idea? What gives?

You still need to know what your system is supposed to do... but you need a BIG-PICTURE view.

15



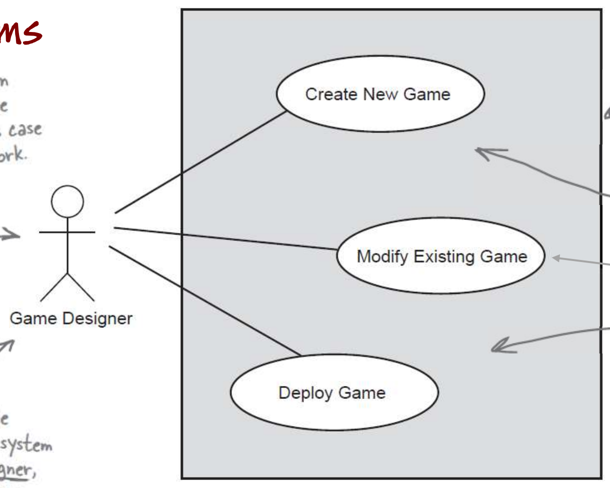
Ever hear that a picture is worth a thousand words?

Let's see if we can show what the system is supposed to do.

16

Use case diagrams

This stick figure is an actor. He acts on the system, which in this case is the game framework.



This big box represents the system. What's inside the box is the system; what's outside uses the system. So the box is the system boundary.

Each of these ovals represents a single use case in the system.

Remember, the actor on this system is a game designer, not a game player.

This use case diagram might not be the most detailed set of blueprints for a system, but it tells you everything the system needs to do, in a simple, easy-to-read format. Use cases are much more detail-oriented, and don't help you figure out the big picture like a good use case diagram does.

17

17

OK, this is just plain **stupid**. What good does that diagram do us? Do we really need to draw a picture to figure out that game designers are going to create and modify games?

Use case diagrams are the blueprints for your system.

Remember, our focus here is on the *big picture*. That use case diagram helps you keep your eye on the fundamental things that your system *must* do.

But what about all those features we worked so hard to figure out? They don't even show up on the use case diagram!

Use your feature list to make sure your use case diagram is complete.

Take your use case diagram, and make sure that all the use cases you listed will cover all the features you got from the customer.

18

18

Feature Magnets

Here's the list of features we came up with

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, and coordinates basic movement.

```
The framework supports
different types of terrain.
```

```
The framework supports
different time periods.
```

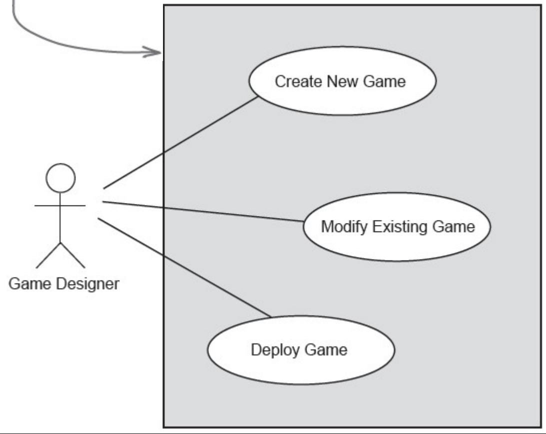
```
The framework supports
multiple unit types.
```

```
The framework supports
add-on modules.
```

```
The framework provides a
board made up of tiles, each
with a terrain type.
```

```
The framework keeps up
with whose turn it is, and
coordinates basic movement.
```

Here's our use case diagram, the blueprint for our system.

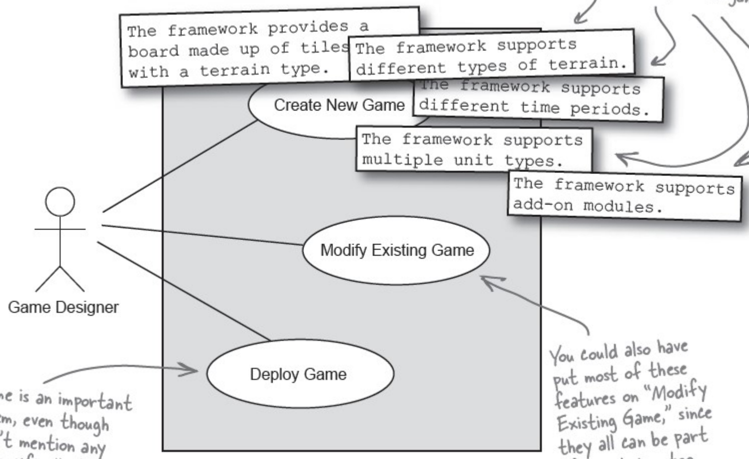


Each feature should be attached to one of the use cases in the system.

Feature magnets.

Feature Magnets Solutions

Almost all of the features have to do with a game designer creating a new game.



```
The framework provides a
board made up of tiles
with a terrain type.
```

```
The framework supports
different types of terrain.
```

```
The framework supports
different time periods.
```

```
The framework supports
multiple unit types.
```

```
The framework supports
add-on modules.
```

Deploying the game is an important piece of the system, even though the customer didn't mention any features related specifically to it.

You could also have put most of these features on "Modify Existing Game," since they all can be part of a redesign, too.

But there's one feature still left...what up with that?

How is this feature related to the system? And what actors are involved? And are we missing some use cases in our diagram? ***What do you think?***

```
The framework keeps up
with whose turn it is, and
coordinates basic movement.
```

?

We know this is a feature, but why doesn't it have a place in our blueprints?

The Little Actor A small Socratic exercise in the style of The Little Lisper

What system are you designing?	A game framework, duh!
So what is the point of the framework?	To let game designers build games.
So the game designer is an actor on the system?	Yes. I've got that in my use case diagram.
And what does the game designer do with the framework?	Design games. I thought we established that!
Is the game the same as the framework?	Well, no, I suppose not.
Why not?	The game is complete, and you can actually play it. All the framework provides is a foundation for the game to be built on.
So the framework is a set of tools for the game designer?	No, it's more than that. I mean, the feature I'm stuck on is something the framework handles for each individual game. So it's more than just tools for the designer.

21

21

The Little Actor A small Socratic exercise in the style of The Little Lisper

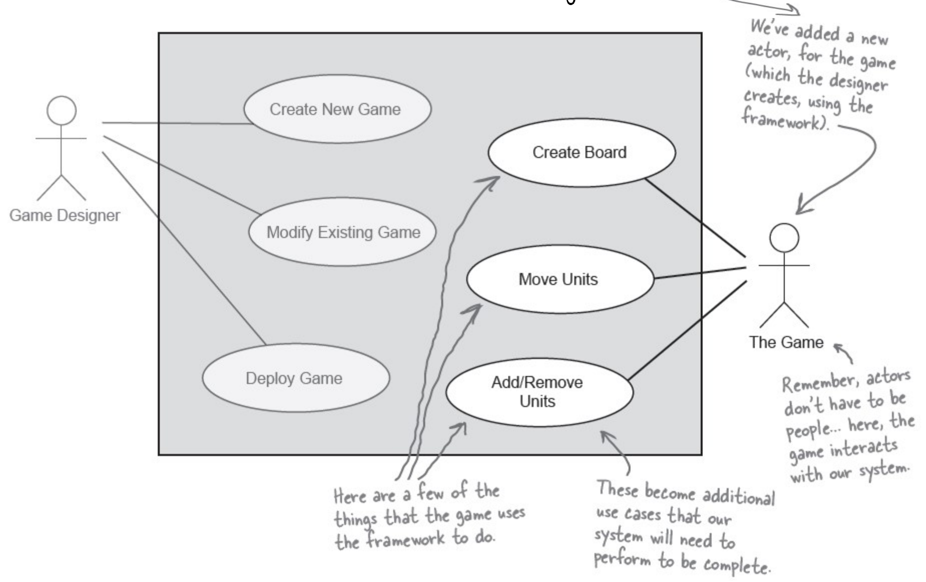
Interesting. So the framework is part of the game, then?	Well, I guess so. But it's like a lower level, like it just provides some basic services to the game. The game sort of sits on top of the framework.
So the game actually uses the framework?	Yes, exactly.
Then the game actually uses the system you're building?	Right, that's just what I said. Oh, wait... then...
...if the game uses the system, what is it?	An actor! The game is an actor!

22

22

Actors are people, too (well, not always)

It turns out that in addition to the game designer, the game itself is an actor on the framework you're building. Let's see how we can add a new actor to our use case diagram:

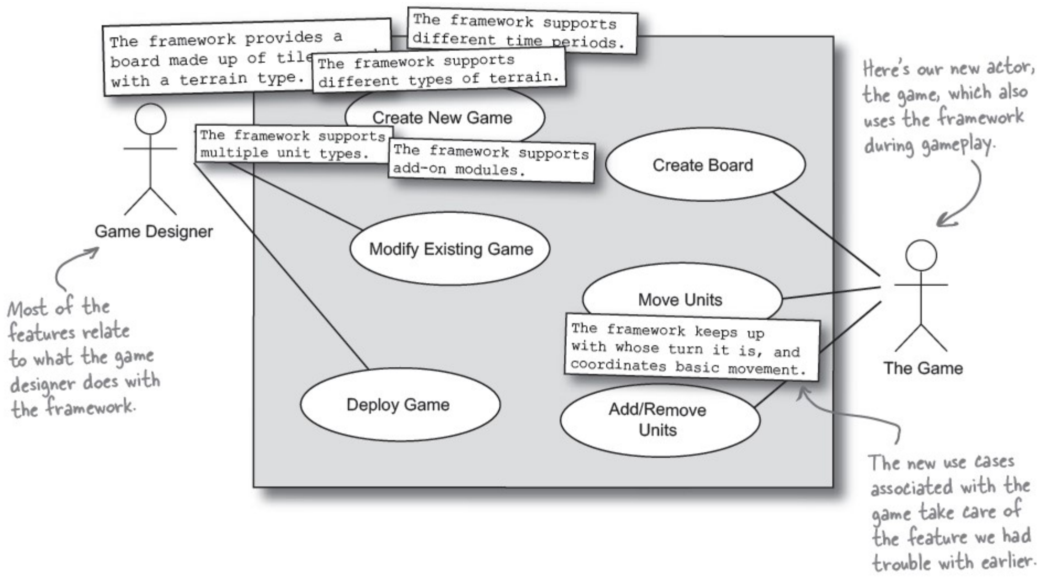


We've added a new actor, for the game (which the designer creates, using the framework).
Game Designer
Create New Game
Modify Existing Game
Deploy Game
Create Board
Move Units
Add/Remove Units
The Game
Remember, actors don't have to be people... here, the game interacts with our system.
Here are a few of the things that the game uses the framework to do.
These become additional use cases that our system will need to perform to be complete.

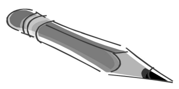
Do these new use cases take care of the feature we couldn't find a place for?

The framework keeps up with whose turn it is, and coordinates basic movement.

Use case diagram... check! Features covered... check!



The framework provides a board made up of tiles with a terrain type.
The framework supports different time periods.
The framework supports different types of terrain.
The framework supports multiple unit types.
The framework supports add-on modules.
Game Designer
Create New Game
Modify Existing Game
Deploy Game
Create Board
Move Units
The framework keeps up with whose turn it is, and coordinates basic movement.
Add/Remove Units
The Game
Here's our new actor, the game, which also uses the framework during gameplay.
Most of the features relate to what the game designer does with the framework.
The new use cases associated with the game take care of the feature we had trouble with earlier.



Sharpen your pencil

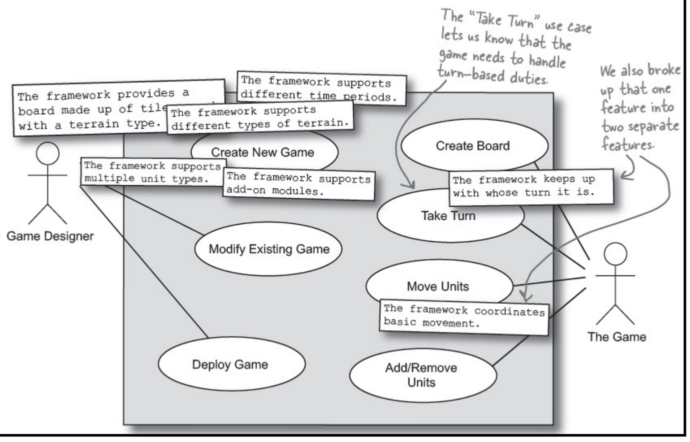
```
The framework keeps up
with whose turn it is, and
coordinates basic movement.
```

That last feature is still a little funny...

The second part of that last feature, about movement, fits in with the "Move Units" use case... but what about keeping up with whose turn it is to move? It seems like there's something still missing from our use case diagram. It's your job to figure out two things:.

1. Who is the actor on "The framework keeps up with whose turn it is?"
   The game is still the actor... it's using the framework to handle managing whose turn it is.

2. What use case would you add to support this partial feature?
   We need a use case for "Take Turn" where the framework handles basic turn duties, and lets the custom game handle the specifics of that process.



The "Take Turn" use case lets us know that the game needs to handle turn-based duties.
We also broke up that one feature into two separate features.
The framework provides a board made up of tiles with a terrain type.
The framework supports different time periods.
The framework supports different types of terrain.
Create New Game
Create Board
The framework supports multiple unit types.
The framework supports add-on modules.
The framework keeps up with whose turn it is.
Take Turn
Game Designer
Modify Existing Game
Move Units
The framework coordinates basic movement.
The Game
Deploy Game
Add/Remove Units

So what exactly have we done?

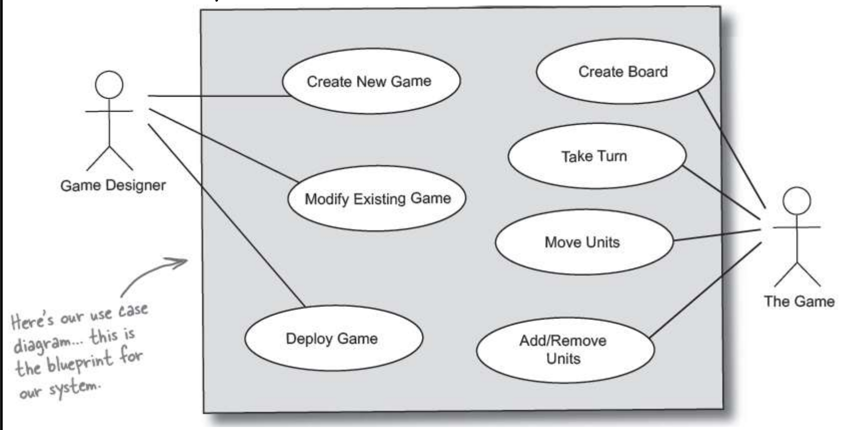
Use a feature or requirement list to capture the BIG THINGS that your system needs to do.

Draw a use case diagram to show what your system IS without getting into unnecessary detail.

Here's our feature list... the system has to do these things.

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.



26

Cubicle Conversation

Isn't it about time we started actually talking about code? I mean, I get that we need a feature list, and use case diagrams, and all that, but at some point we have to actually build something, you know?
Jim
Frank

Domain analysis lets you check your designs, and still speak the **customer's language.**

Frank: I don't know, Jim. I think we have been talking about code.

Jim: How do you figure that? I mean, what line of code is "framework supports different types of terrain" really going to turn into?

Frank: You're talking about those features we figured out, right? Well, that's not just one line of code, but it certainly is a big chunk of code, right?

Jim: Sure... but when do we get to talk about what classes we need to write, and the packages we put those classes into?

Frank: We're getting to that, definitely. But the customer really doesn't understand what most of that stuff means... we'd never be sure we were building the right thing if we started talking about classes and variables.

Jim: What about class diagrams? We could use those to show what we're going to code, couldn't we?

Frank: Well, we could... but do you think the customer would understand that much better? That's really what domain analysis is all about. We can talk to the customer about their system, in terms that they understand. For Gary, that means talking about units, and terrain, and tiles, instead of classes, objects, and methods.

27

27

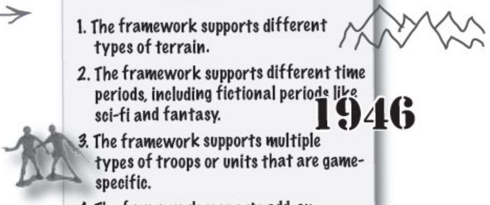
Let's do a little domain analysis!

These features are using terms that the customer understands.

The domain here is game systems.

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.



This whole feature list is a form of analysis, just like we've been doing earlier

Let's put all these things we've figured out about the game system together, in a way that Gary, our customer, will actually understand.

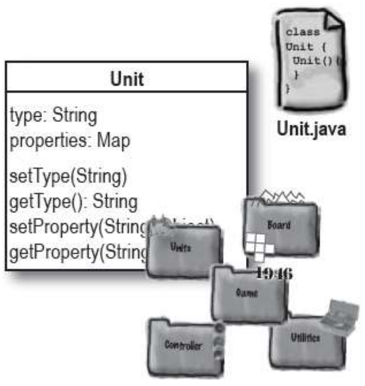
This is a process called *domain analysis*, and just means that we're describing a problem using terms the customer will understand.

the Scholar's Corner

domain analysis. The process of identifying, collecting, organizing, and representing the relevant information of a domain, based upon the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within a domain.

28

What most people give the customer...

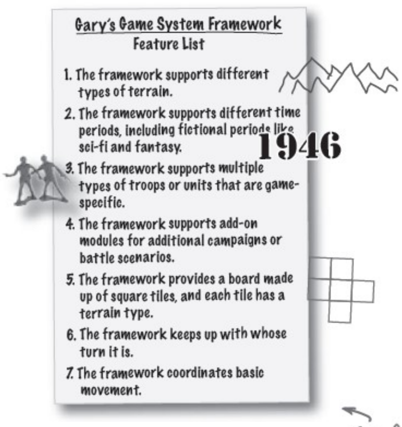


These are class and package diagrams, and code-level details about how you'll build Gary's game framework.

What the heck is this? I have no idea if this is what I want.

Gary's totally lost, because he's not a programmer! You didn't speak his language.

What we're giving the customer...


Gary's Game System Framework
Feature List

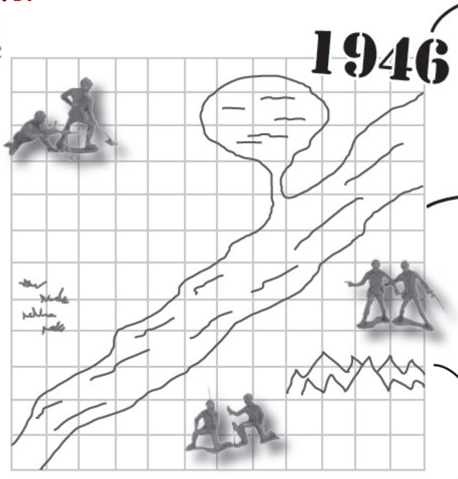
1. The framework supports different types of terrain.
2. The framework supports different time periods, including fictional periods like sci-fi and fantasy.
3. The framework supports multiple types of troops or units that are game-specific.
4. The framework supports add-on modules for additional campaigns 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.

Our feature list, in language the customer understands.

Gary's thrilled, because he understand what you're building, and knows it will do what he wants it to do.

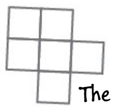
Now divide and conquer

Here's a rough drawing of some of the core parts of the game framework.



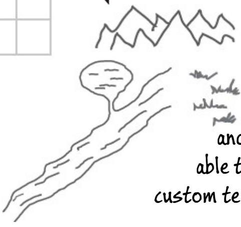
Time Periods

We may not need to do much here... as long as we support different terrains, unit types, and weapons, this should come naturally.



Tiles

The framework needs to have a basic tile, and each tile should be able to support terrain types, units, and probably handle battles, too.



Terrain Types

Each tile should support at least one terrain type, and game designers should be able to create and use their own custom terrain types, from grass to lakes to asteriod dunes.

Units

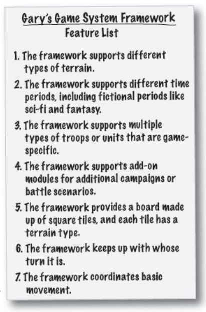
We need a way to represent a basic unit, and let the game designers extend that to create game-specific units.

We can break the large framework down into several smaller, more manageable, pieces.

31

31

The Big Break-Up

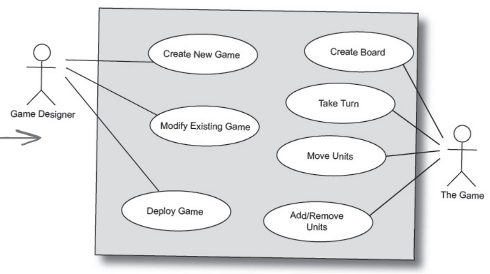


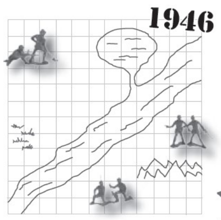
Gary's Game System Framework
Feature List

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.

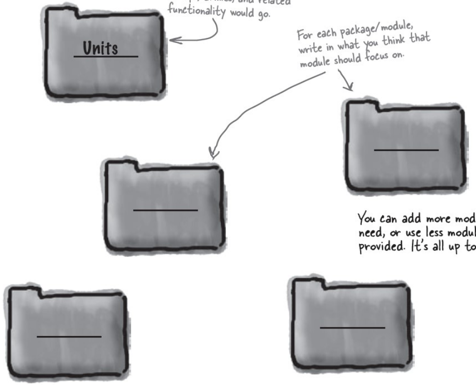
You need to address all the features in the system...

...as well as the functionality laid out in your use case diagram.





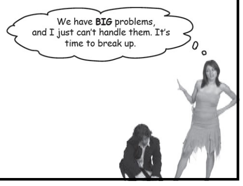
Here's the game board to remind you of some of the major areas to focus on... but remember, this isn't everything!



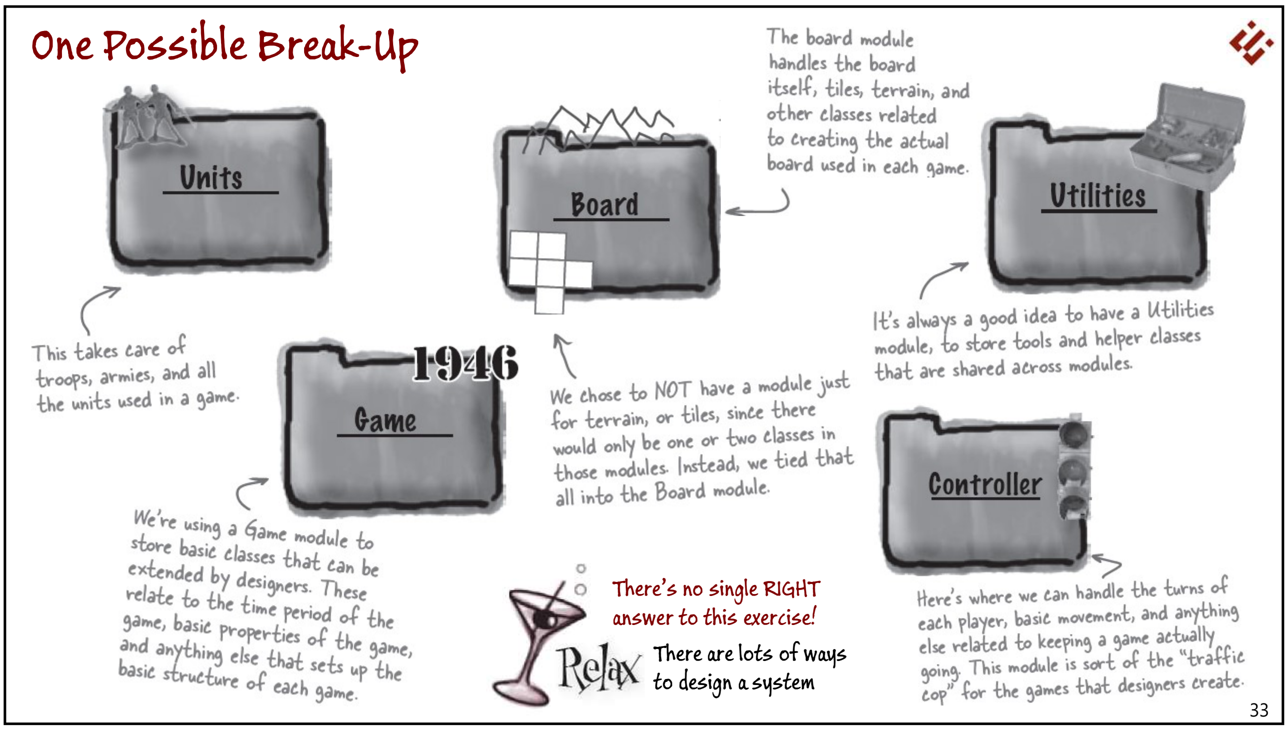
We've added a "Units" module to get you started. This would be where classes representing troops, armies, and related functionality would go.

For each package/module, write in what you think that module should focus on.

You can add more modules if you need, or use less modules than we've provided. It's all up to you!


We have **BIG** problems, and I just can't handle them. It's time to break up.

32



One Possible Break-Up

The board module handles the board itself, tiles, terrain, and other classes related to creating the actual board used in each game.

This takes care of troops, armies, and all the units used in a game.

It's always a good idea to have a Utilities module, to store tools and helper classes that are shared across modules.

We chose to NOT have a module just for terrain, or tiles, since there would only be one or two classes in those modules. Instead, we tied that all into the Board module.

We're using a Game module to store basic classes that can be extended by designers. These relate to the time period of the game, basic properties of the game, and anything else that sets up the basic structure of each game.

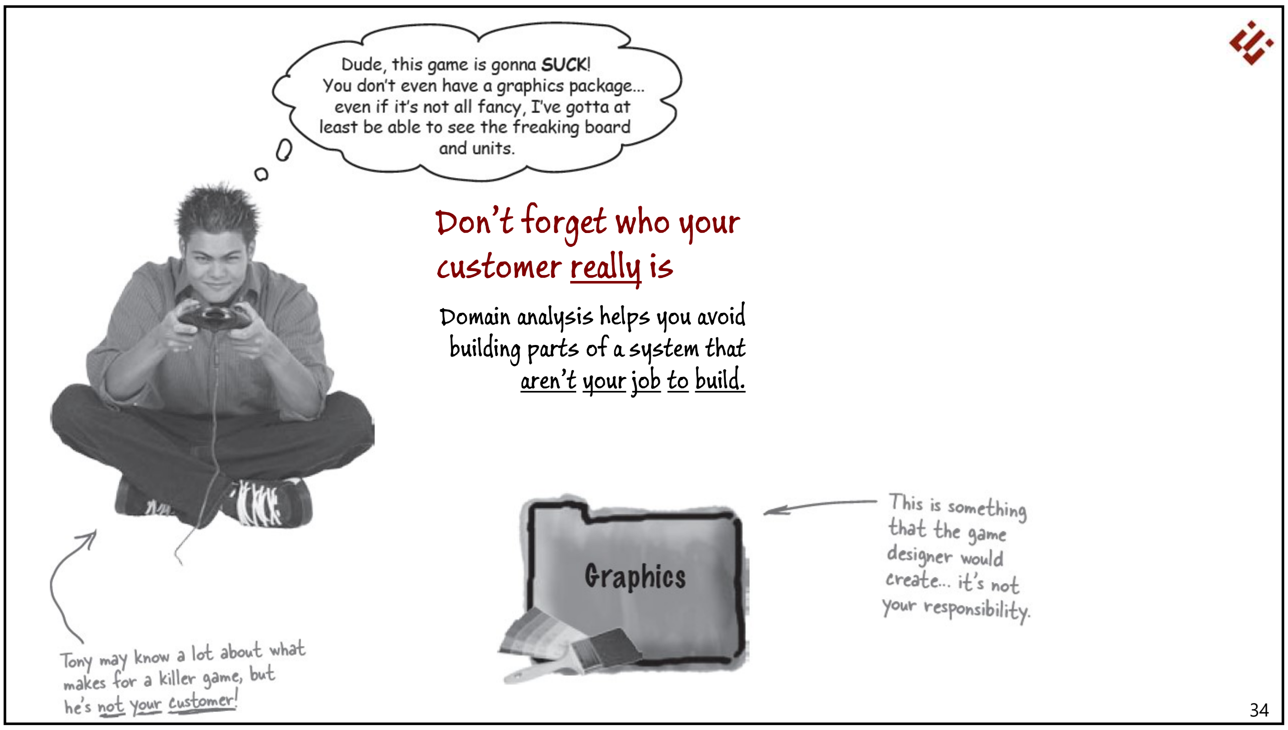
There's no single RIGHT answer to this exercise!

Relax There are lots of ways to design a system

Here's where we can handle the turns of each player, basic movement, and anything else related to keeping a game actually going. This module is sort of the "traffic cop" for the games that designers create.

33

33



Dude, this game is gonna SUCK! You don't even have a graphics package... even if it's not all fancy, I've gotta at least be able to see the freaking board and units.
Graphics

Don't forget who your customer really is

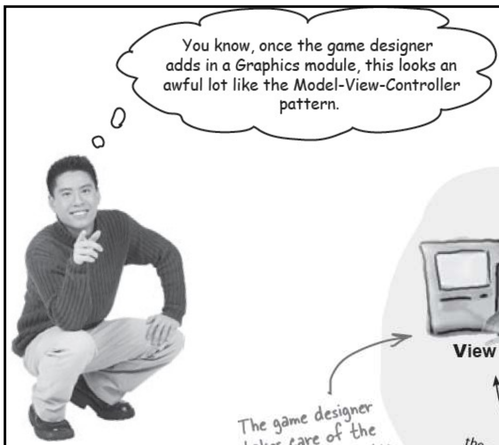
Domain analysis helps you avoid building parts of a system that aren't your job to build.

This is something that the game designer would create... it's not your responsibility.

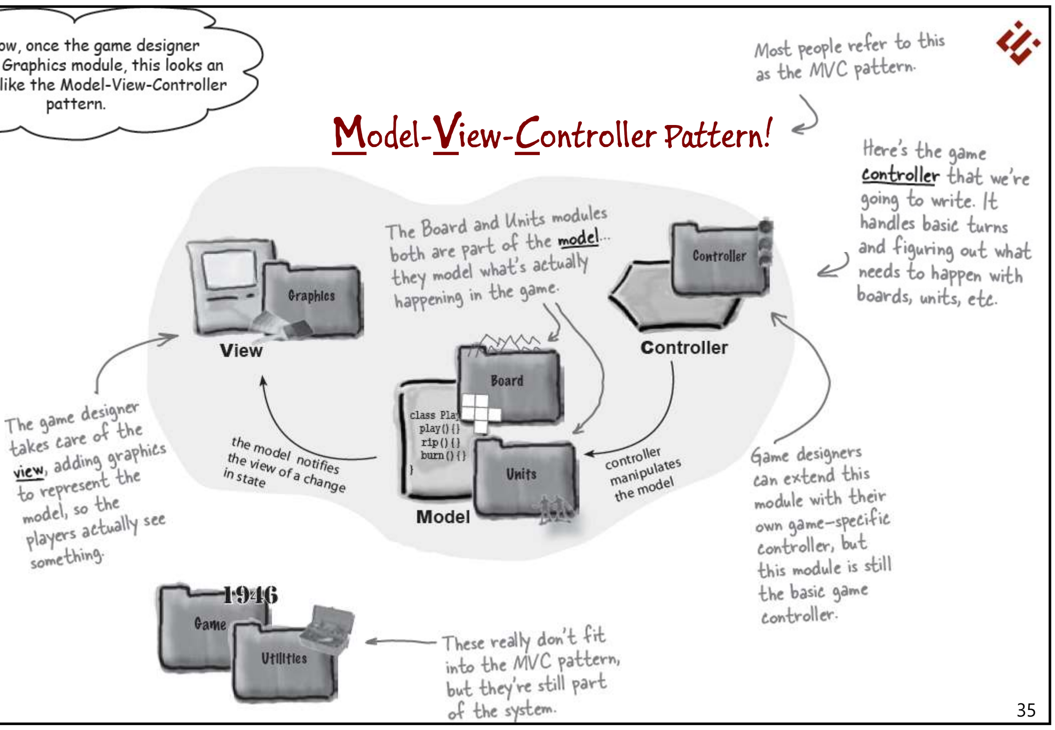
Tony may know a lot about what makes for a killer game, but he's not your customer!

34

34


You know, once the game designer adds in a Graphics module, this looks an awful lot like the Model-View-Controller pattern.

Model-View-Controller Pattern!


Most people refer to this as the MVC pattern.
Here's the game controller that we're going to write. It handles basic turns and figuring out what needs to happen with boards, units, etc.
The Board and Units modules both are part of the model... they model what's actually happening in the game.
Graphics
View
Controller
Controller
Board
class Pla
play() {}
rip() {}
burn() {}
}
Units
Model
the model notifies the view of a change in state
controller manipulates the model
The game designer takes care of the view, adding graphics to represent the model, so the players actually see something.
Game designers can extend this module with their own game-specific controller, but this module is still the basic game controller.
1946
Game
Utilities
These really don't fit into the MVC pattern, but they're still part of the system.

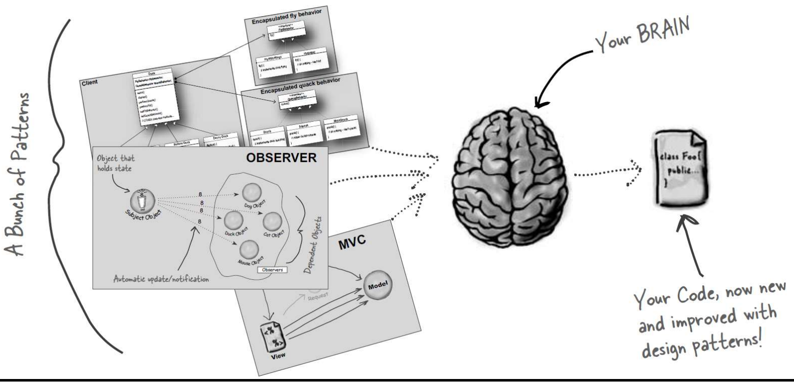
35

What's a design pattern? And how do I use one?

- Think of off-the-shelf libraries and frameworks.
  - Java APIs and all the functionality they give you: network, GUI, IO, etc.

THEY DON'T HELP
STRUCTURING YOUR CODE

Design patterns don't go directly into your code, they first go into your BRAIN.

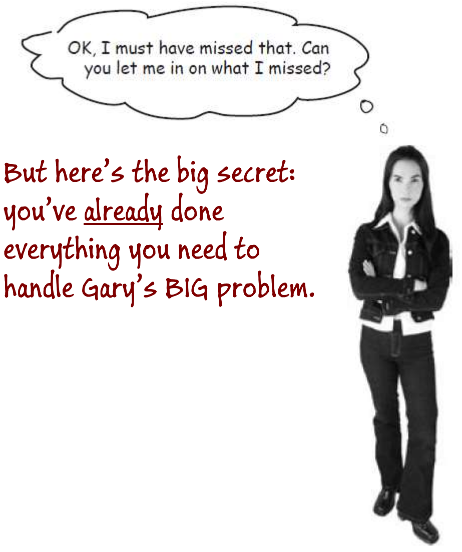

A Bunch of Patterns
Encapsulated fly behavior
Encapsulated quack behavior
Client
OBSERVER
Object that holds state
Automatic update/notification
Dependent Objects
MVC
Model
View
Your BRAIN
class Foo{ public... }
Your Code, now new and improved with design patterns!

Feeling a little bit lost?

We've done a lot of things this week, and some of them don't even seem to be related...

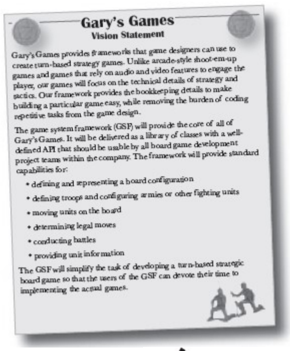
- Gathering features
- Domain analysis
- Breaking Gary's system into modules
- Figuring out Gary's system uses the MVC pattern.

But how does any of this really help us solve BIG problems?


OK, I must have missed that. Can you let me in on what I missed?

But here's the big secret: you've already done everything you need to handle Gary's BIG problem.

37


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

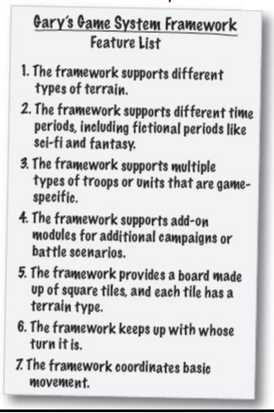
The GSF will simplify the task of developing a turn-based strategic board game so that the users of the GSF can devote their time to implementing the actual games.

We started out with this rather vague, far-reaching vision statement. Now that's a BIG problem.

1. We listened to the customer.

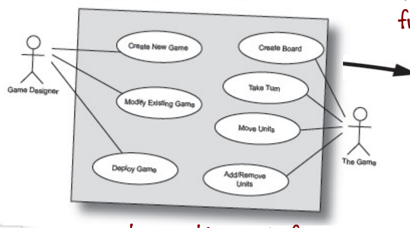
Using domain analysis, we made sure we understand what Gary wanted his system to do.

2. We made sure we understood the system.


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.

Once we knew what we were building, we created a use case diagram to help us understand the big picture.



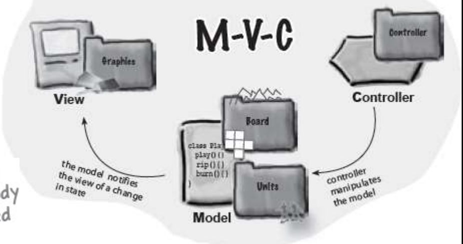
3. We drew up blueprints for the system we're building.

4. We broke the big problem up into smaller pieces of functionality.

Units
Board
1946
Game
Controller
Utilities

5. We apply design patterns to help us solve the smaller problems.

We even took a design pattern that we already understand, and applied it to our system.

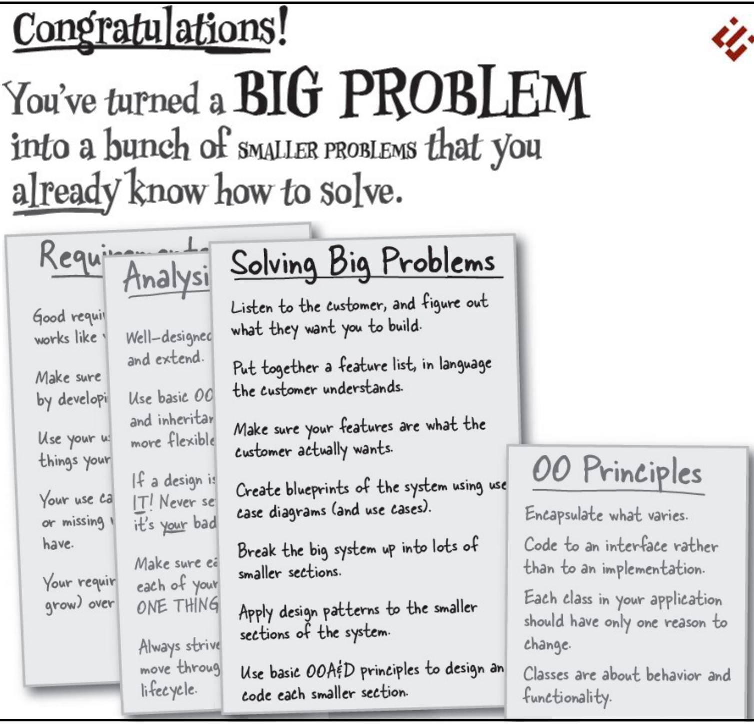

M-V-C
Graphics
View
Controller
Controller
Board
Units
Model
the model notifies the view of a change in state
controller manipulates the model

38

Tools for your toolbox

Bullet Points

- The best way to look at a big problem is to view it as a collection of smaller problems.
- Just like in small projects, start working on big projects by gathering features and requirements.
- Features are usually "big" things that a system does, but also can be used interchangeably with the term "requirements."
- Commonality and variability give you points of comparison between a new system and things you already know about.
- Use cases are detail-oriented; use case diagrams are focused more on the big picture.
- Your use case diagram should account for all the features in your system.
- Domain analysis is representing a system in language that the customer will understand.
- An actor is anything that interacts with your system, but isn't part of the system.



Congratulations!

You've turned a BIG PROBLEM into a bunch of SMALLER PROBLEMS that you already know how to solve.

Requi

Good requi
works like

Make sure
by developi

Use your u
things your

Your use ca
or missing
have.

Your requir
grow) over

Analysi

Well-designed
and extend.

Use basic OO
and inheritar
more flexible

If a design is
IT! Never se
it's your bad

Make sure ea
each of your
ONE THING

Always strive
move throug
lifecycle.

Solving Big Problems

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

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

Make sure your features are what the customer actually wants.

Create blueprints of the system using use case diagrams (and use cases).

Break the big system up into lots of smaller sections.

Apply design patterns to the smaller sections of the system.

Use basic OOA&D principles to design an code each smaller section.

OO Principles

Encapsulate what varies.

Code to an interface rather than to an implementation.

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

Classes are about behavior and functionality.

39