Bringing Order to Chaos

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/6448 — Lecture 12 — 10/04/2007

Lecture Goals

- Review material from Chapter 7 of the OO A&D textbook
  - How do you use software architecture to guide the development process?
  - The three Q's of architecture
  - Risk and how to reduce it
  - Commonality Analysis
  - Discuss the Chapter 7 Example: Gary's Game Framework
  - Emphasize the OO concepts and techniques encountered in Chapter 7
  - Review Homework 1

Bringing Order to Chaos

- At the start of a development project for a large, complex software system
  - You don’t have a lot of information
  - You don’t know a lot about your domain
- And yet…
  - You are expected to make
    - implementation decisions (concerning expensive pieces of middleware)
    - decisions regarding non-functional requirements
    - progress on decomposing the problem domain into manageable pieces
    - progress on identifying the features of the system under design

How Does One Start?

- Indeed after doing the work that we described in the last lecture…
  - domain analysis
    - talking to customer
    - developing feature list
    - developing use case diagram
      - which requires hard work like identifying all of the different types of users an their major tasks
  - decomposing “big problem” into “smaller modules”
- You may be feeling overwhelmed with the information you have so far
  - or have doubts about its accuracy (which then leads to paralysis)
- So, how does one identify what to do next?

Answer: Software Architecture

- It's not enough to identify the individual pieces of a big problem
  - You also need to know how those pieces fit together
  - You need to have a mechanism for prioritizing your work on those pieces
- The missing piece is the software architecture for your system
  - A software architecture **provides a structure to the design** of a software system, **highlights the most important parts** of your system, and the **relationships between those parts**
- A second definition
  - A software architecture is the **organizational structure** of a system, including its **decomposition into parts**, **their connectivity**, **interaction mechanisms**, and the **guiding principles and decisions** you use in the design of a system
- An architecture can take a chaotic mess and turn it into a well-ordered app!

Back to the Three Step Process

- Our three step OO A&D process
  - Make sure your software does what the customer wants it to do
  - Apply basic OO principles to add flexibility
  - Strive for a maintainable, reusable design
- points the way towards what to do first: Focus on Functionality
  - As such, we will start with what the customer wants to do
  - And for large systems that means looking at the feature list

The Feature List

Features for Gary's Game System

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

But now the question is which of these are the most important?

The Three Qs of Architecture

- We are trying to identify the important functionality of our system
  - the features that are **architecturally significant**
- To identify important pieces of functionality, **ask 3 questions of each feature**
  - Is the feature part of the **essence** of the system?
    - The essence of a system is what that system is at its most basic level
    - “If I don’t implement this feature, would the system meet its goals?”
  - What does the feature **mean**?
    - Not having a clear idea might indicate that the feature can take a lot of time to get right; best to start on it early
  - How will I implement the feature?
    - Focus on features that **seem really hard to implement** so they don’t bog you down later in the development cycle

The Essence of Gary's Game Framework

- After applying the 3 Qs of architecture, the book focuses on these features
  - The board for the game (the essence)
  - Game-Specific Units (the essence and what does this mean?)
  - Framework coordinate basic movement (Can we implement this?)
- Having identified these features, we can now examine them in depth
  - and this will serve to finally get the project moving… after these features get fleshed out, it will be easier to move on to the remaining features/ modules and flesh them out as well

Why are these features important? Risk

- The importance of the three Qs of architecture is that these questions help you to identify the major risks associated with your development process
  - The reason that the three features on the previous slide are architecturally significant is that they all introduce **RISK** to your project (i.e. the risk that the project will fail due to this particular feature/requirement)
    - If we don't know what some feature means, we have a risk of not meeting schedules and deadlines as we perform a lot of work to discover its meaning
    - If a feature seems hard to implement, there is a risk we won't figure it out or it will take a really long time to accomplish
    - If the core features are not in place, there's a risk that the customer won't like the finished product
- The key task during this phase of transitioning from large, complex problems to smaller, more manageable, problems is **reducing risk**

Feature 1: The board

- We want to consider the design of the board
  - an essential part of the framework
- Since we don’t have a lot of detail from the user, we can’t create a use case
  - but we may have enough information to develop a single scenario
    - recall that a single use case will have one or more paths (aka scenarios)
- A scenario for the board is developed
  - create the board
  - move player 1’s and player 2’s units such that a battle takes place
    - take into consideration the terrain that they encounter
  - perform the battle
  - remove units that lost the battle from the board

From Scenario to Code

```
1  public class Board {
2
3      private int width, height;
4      private List tiles;
5
6      public Board(int width, int height) {
7          this.width = width;
8          this.height = height;
9          initialize();
10     }
11
12     private void initialize() {
13         tiles = new ArrayList(width);
14         for (int i=0; i<width; i++) {
15             tiles.add(i, new ArrayList(height));
16             for (int j=0; j<height; j++) {
17                 ((ArrayList)tiles.get(i)).add(j, new Tile());
18             }
19         }
20     }
21
22     public Tile getTile(int x, int y) {
23         return (Tile)((ArrayList)tiles.get(x-1)).get(y-1);
24     }
25
26     public void addUnit(Unit unit, int x, int y) {
27         getTile(x, y).addUnit(unit);
28     }
29
30     public void removeUnit(Unit unit, int x, int y) {
31         getTile(x, y).removeUnit(unit);
32     }
33
34     ...
35 }
36
```

Board is placed in a Java package called board (not shown)

The creation of the board is moved out of the constructor and into its own method

Place a Tile() at each coordinate and offer a means for retrieving tiles

Delegate the handling of Units to the Tile class (not all Unit methods shown)

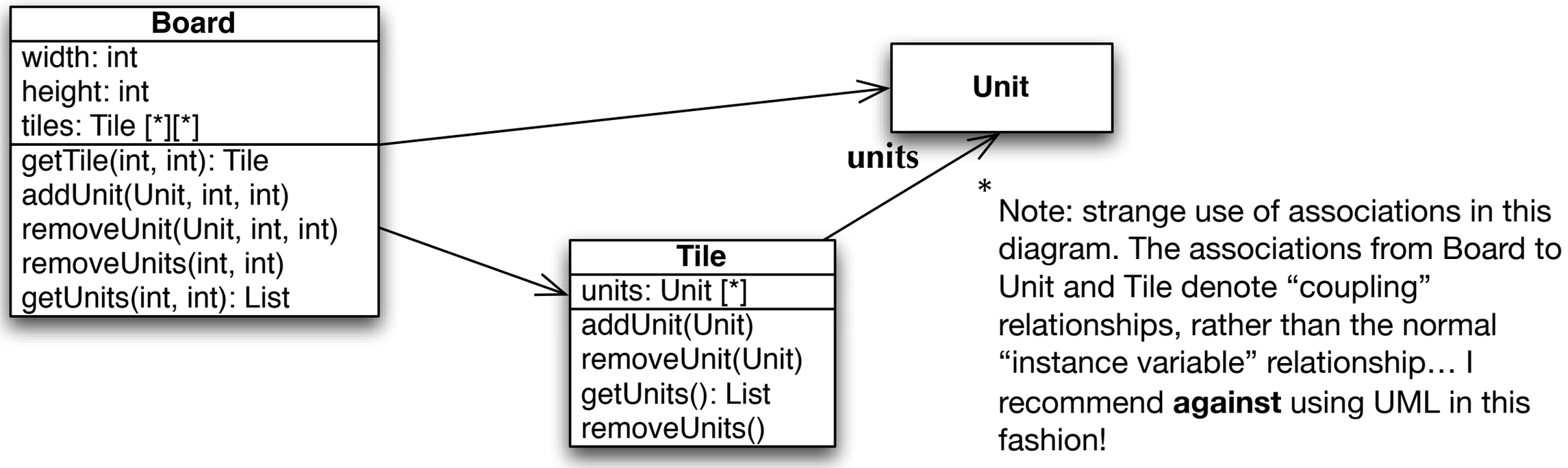
Create skeleton classes for Tile and Unit… they are outside our focus for now

Focus on one feature at a time!

Don’t get distracted with features that won’t reduce the risks of your project

Reflection: Rapid Prototying

- Writing code for the board class at this stage of the game is an example of rapid prototyping
  - It helps answer the question
    - do I really understand what the board has to do?
  - We could have also chosen to just create a class diagram and outline Board's attributes and methods and associations, like this:



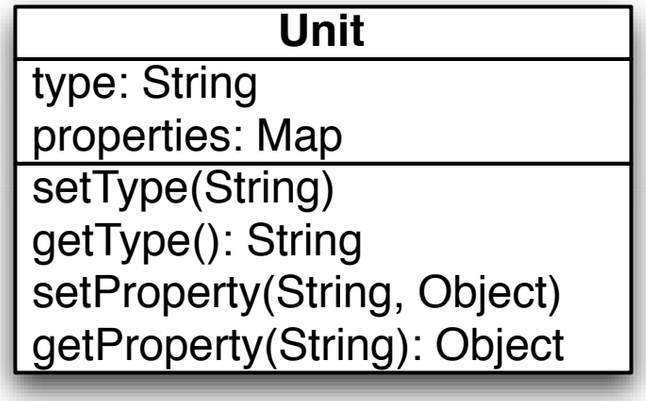
Feature 2: Game Specific Units

- We focus on the “game specific units” feature next for several reasons
  - Its a key feature but currently way underdeveloped (see previous diagram)
  - It is related to the Board concept and since architecture includes specifying the relationships between major system components, we need to define Units further such that we can begin to consider the relationship between Board and Unit
- Game Specific Units
  - Different game designers have different ideas about Units
    - some want attack, defense, and experience properties
    - some want lots of weapons
    - some want units to have names and track relationships with other units
    - all want different types of units: land, air, and space

Commonality Analysis

- For framework design, we want to identify the shared properties of all the different requests received from the game designers
  - Shared aspects go in the framework
  - Variable aspects go outside the framework but should still be accessible to the framework via inheritance, polymorphism, interfaces, etc.
- The common need in this instance is a bit abstract but can be stated succinctly
  - They need a Unit class with a variable number of atts of different types
    - Sound familiar?
  - This is exactly the same requirement that our InstrumentSpec class had
    - When its the number and type of properties that is the “thing that varies”, make the properties dynamic via a collection class
  - We want to avoid, therefore, an approach in which Unit is sub-classed

Solution: Just like Instrument Spec



Note: no need for code this time. The implementation of this class is obvious!

Provides Scalability

Number of Unit Types	Separate Subclasses	Dynamic Properties
3	4	1
25	26	1
100	101	1

To Code or Not to Code

- The book brings up an excellent point on page 364
  - They have a discussion about why they wrote code for the Board class but skipped writing code for the Unit class, noting that **reducing risk is key**
    - Do some initial design work to verify that a feature is **essential**, **well understood**, and **feasible**
      - If you need code, then write it but if you do not, then avoid it!
- They then ask the question “So, there’s not a lot of code involved in OO A&D, is there?”. Their response is excellent.
  - OO A&D is **ALL** about code. Its about getting your design right, having it be clean, well structured, flexible and extensible… because if you get that right then the code will be EASY to write and maintain
  - Sometimes the **best way** to **write great code** is to **hold off** on writing code **as long as you can**!

Feature 3: Coordinating Movement

- They repeat the process used for feature 2 for feature 3
  - Ask the customer
  - Perform Commonality Analysis
  - Develop a Design for the feature (gain confidence you can implement it)
- After doing these two steps, they acquire this information:

What's common?	What's variable?
Check if move is legal	**The algorithm to check the legality of a move is different for every game**
Unit's attributes determine how far it can move	**The number and specific properties used for this are different for every game**
Other factors (such as terrain) will affect movement	**The other factors that affect movement are different for every game**

What do they decide? They Punt!

- In the book, they decide to punt on this feature and declare it out-of-scope
  - Lame! (In my not so humble opinion…)
- They say: “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”
- But I think they missed out on an opportunity to further define the Board and Unit classes (note: the book specifically discounts my approach)
  - In particular, if they had made Unit an abstract base class, they could have game designers add subclasses that represent movement rules for different types of Units (losing some of the previous scalability)
    - The abstract interface: canMove(Tile), determinePath(), etc.
    - The Board class could employ a Strategy pattern to allow Units to retrieve the other factors that might affect movement, etc.
- The framework’s complexity goes up, but in exchange it does more for you!

Classic Programmer Reaction

- “Great. So we’ve got a little sheet of paper with some check marks, a few classes that we know aren’t finished, and lots of UML diagrams. And I’m supposed to believe **this** is how you write great software.”
- Absolutely!
  - Remember this is just the start of the **first** iteration of **many**
  - When you start a project, you don’t have a lot of information and you can waste a lot of time if you don’t start by acquiring more detail in a systematic fashion
    - Domain Analysis, The Three Qs of Architecture, and Commonality Analysis can all help you to systematically whittle a large problem down to smaller more manageable chunks
  - The key point is to reduce risk with each step you take, so you can get to a point where you have a handle on the “big picture” and can start working on its constituent parts

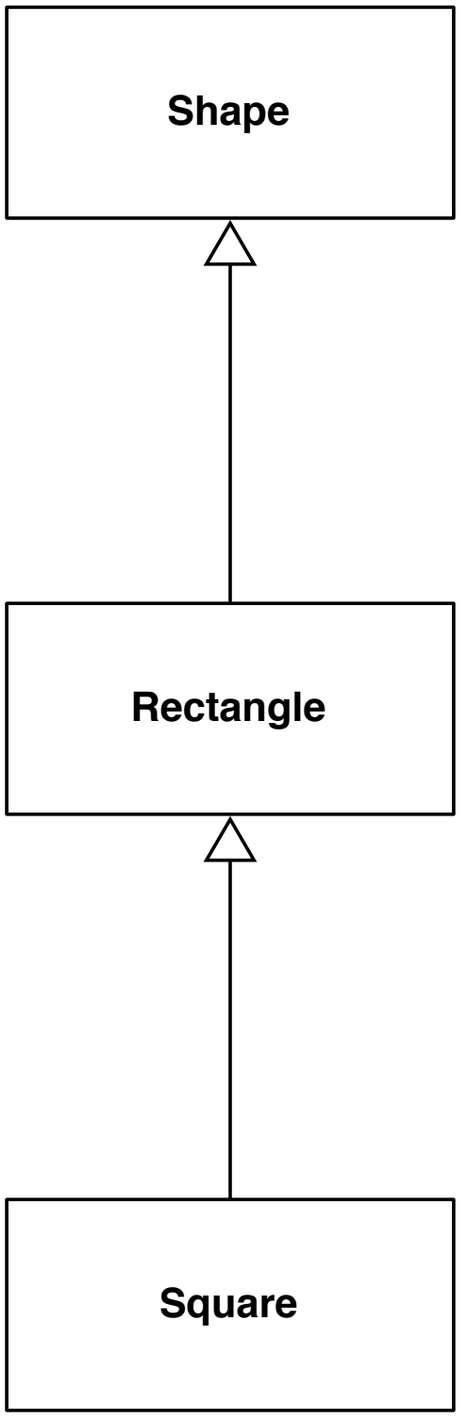
Wrapping Up

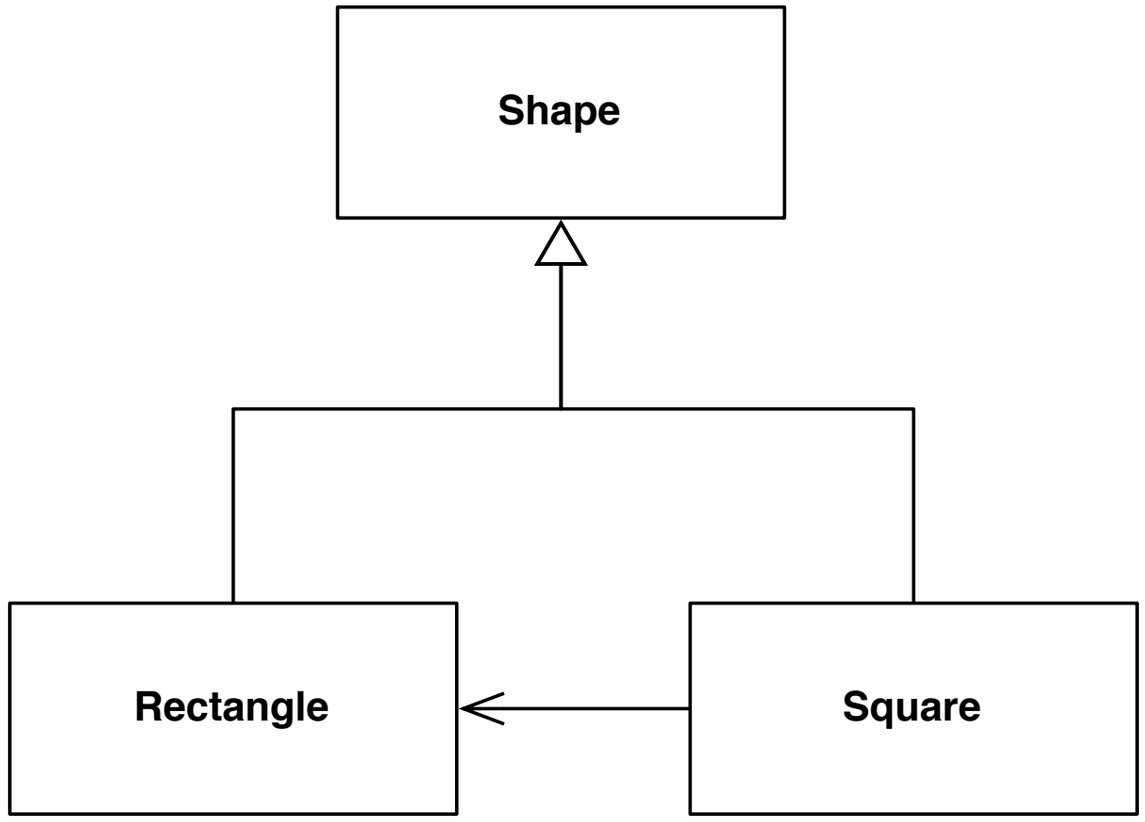
- Architecture helps you turn all your diagrams, plans, and feature lists into a well-ordered application
- Focus on features that are the essence of your system, that have unclear meanings, or that don't seem feasible
- If you don't need all of the detail of a use case, just write simple scenarios to help you identify the characteristics of a particular feature
  - These simple scenarios can be turned into the "main path" of a use case at a later stage in the software life cycle
- Be open to getting lots of feedback from your customer… if something is unclear, go back to them and get additional feedback so that you can move forward with your design

Review of Homework 1

- Question 1: Use of Inheritance versus Use of Delegation
- Question 2: Create a Class Diagram
- Question 3: Design and Implement a Simple OO System

Question 1: Inheritance versus Delegation





Semantics?

Questions

- Assume rectangle has width and height attributes with their associated getters and setters
  - getWidth(), getHeight(), setWidth(), setHeight()
- Square cannot reuse all of these methods. Why Not?
- Does this have any implication on using inheritance between Rectangle and Square?
  - Put another way, can using inheritance between these two classes cause any problems?
- How might the diagram on the right address these issues?
  - Do problems remain with this approach?

Question 2: Create a Class Diagram

- A structural computing system is made up of elements.
- There are two types of elements, atoms and collections.
- Atoms are used to store application-specific objects supplied by clients
- Collections are used to group other elements
- All elements have a unique id and a set of attributes.
- Each attribute has a name, which is a string, and a value, which can be one of any number of different types all of which share a common interface provided by an abstract class called AttributeValue.
- Elements are stored by a repository, which manages their persistence and which also can be used to search for specific elements via their attributes.

Question 3: Design/Implement Simple OO System

- Shape Environment
  - Canvas with at least a Report() method
  - Class Hierarchy of Shapes with various requirements
  - Main program with these requirements

    1.creates a canvas

    2.adds 5 to 10 shapes of various types to it.

    3.randomly select a shape and apply move() or scale() operation to it

    4.Repeat the previous step 19 times until 20 shapes have been randomly selected and either moved or scaled.

    5.find two shapes farthest apart along x axis and align them vertically

    6.find two shapes farthest apart along y axis and align them horizontally
- Your main program should invoke report() method after steps 2-6

Coming Up Next

- Lecture 13: Originality is Overrated
    - Read Chapter 8 of the OO A&D book
- Lecture 14: Testing And Iterating
    - Read Chapter 9 of the OO A&D book