Originality is Overrated: OO Design Principles & Iterating And Testing

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/5448 — Lecture 12 — 10/01/2009

Lecture Goals: Part One

- Review material from Chapter 8 of the OO A&D textbook
    - Object-Oriented Design Principles
        - Open-Closed Principle
        - Don’t Repeat Yourself
        - Single Responsibility Principle
        - Liskov Substitution Principle
        - Aggregation and Composition, Revisited
- Discuss the examples in Chapter 8
- Emphasize the OO concepts and techniques encountered in Chapter 8

Lecture Goals: Part Two

- Review material from Chapter 9 of the OO A&D textbook
  - Iteration is fundamental
    - Feature driven development
    - Use case driven development
  - Testing is fundamental
    - Test driven development
    - Proving yourself to the customer
  - Programming by Contract
  - Defensive Programming
- Discuss the examples in Chapter 9
- Emphasize the OO concepts and techniques encountered in Chapter 9

Originality is Overrated

- Corollary: “Only Steal from the Best” — various sources
- OO A&D has been performed in various forms and in various contexts for nearly 40 years
    - Over that time, designers have developed a set of principles that ease the task of creating OO designs
- If you apply these principles in your own code, you will be “stealing” from the best that the OO A&D community has to offer
    - The same is true of Design Patterns

OO Principles: What We've Seen So Far

- We've seen the following principles in action over the past eight lectures
  - **Classes are about behavior**
    - Emphasize the behavior of classes over the data of classes
      - Don't subclass for data-related reasons
  - **Encapsulate what varies**
    - Provides info. hiding, limits impact of change, increases cohesion
  - **One reason to change**
    - Limits impact of change, increases cohesion
  - **Code to an Interface**
    - Promotes flexible AND extensible code
      - Code applies across broad set of classes, subclasses can be added in a straightforward fashion (including at run-time)

New Principles

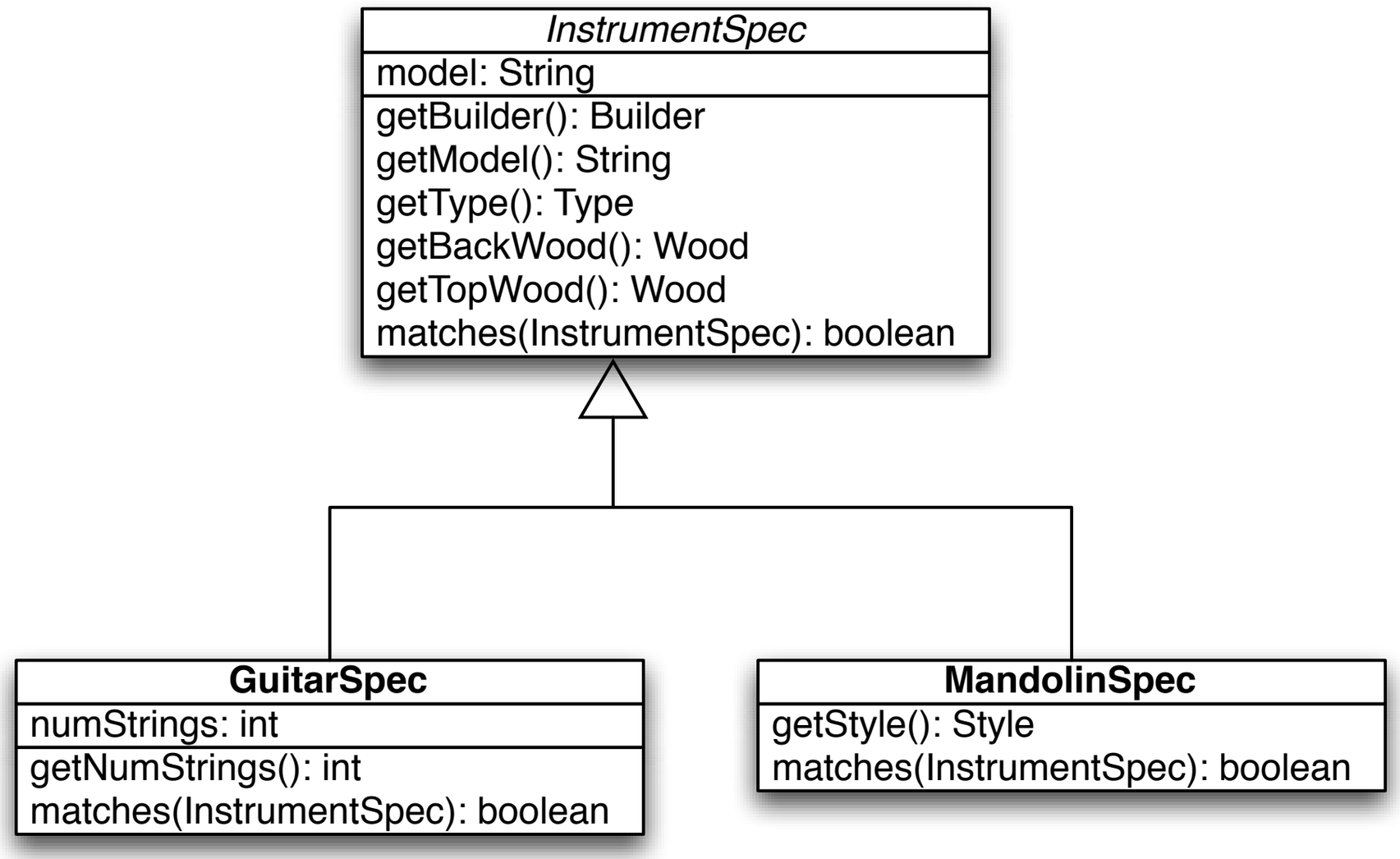
- **Open-Closed Principle (OCP)**
  - Classes should be open for extension and closed for modification
- **Don’t Repeat Yourself (DRY)**
  - Avoid duplicate code by abstracting out things that are common and placing those things in a single location
- **Single Responsibility Principle (SRP)**
  - Every object in your system should have a single responsibility, and all the object’s services should be focused on carrying it out
- **Liskov Substitution Principle (LSP)**
  - Subtypes must be substitutable for their base types

Open-Closed Principle

- Classes should be open for extension and closed for modification
- Basic Idea:
  - Prevent, or heavily discourage, changes to the behavior of existing classes
    - especially classes that exist near the root of an inheritance hierarchy
  - If a change is required, create a subclass and allow it to extend/override the original behavior
    - This means you must carefully design what methods are made public and protected in these classes; private methods cannot be extended
- Why is this important?
  - Limits impact on code that follows “Code to an Interface” principle
    - If you change the behavior of an existing class, a lot of client code may need to be updated

Example

- We've seen one example of the Open-Closed Principle in action
  - InstrumentSpec.matches() being extended by GuitarSpec and MandolinSpec



Is this just about Inheritance?

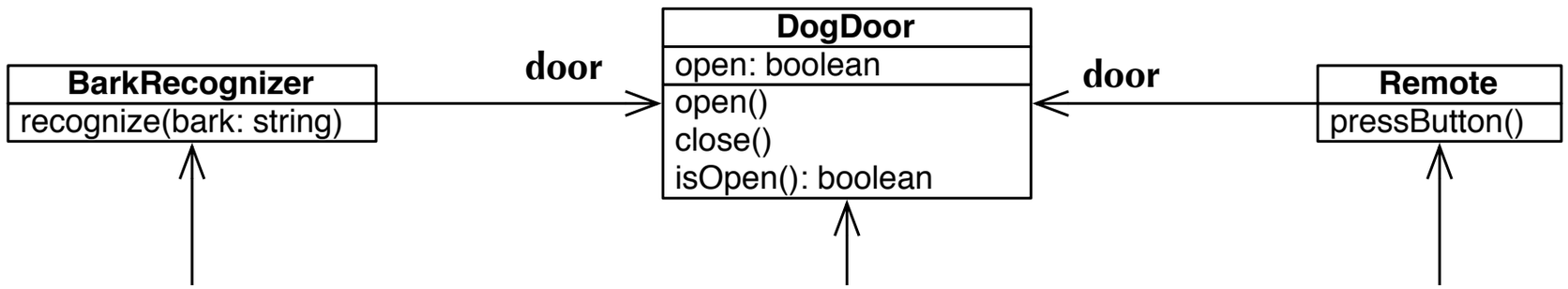
- Inheritance is certainly the easiest way to apply this principle
    - but its not the only way
- In looking at Design Patterns, we'll see that composition and delegation offer more flexibility in extending the behavior of a system
    - Inheritance still plays a role but its more background than foreground
- The key point of the OCP is to get you to **be reluctant to change working code**, look for opportunities to extend, compose and/or delegate your way to achieve what you need first

Don’t Repeat Yourself

- Avoid duplicate code by abstracting out things that are common and placing those things in a single location
- Basic Idea
  - Duplication is Bad!
  - At all levels of software engineering: Analysis, Design, Code, and Test
- We want to avoid duplication in our requirements, use cases, feature lists, etc.
- We want to avoid duplication of responsibilities in our code
- We want to avoid duplication of test coverage in our tests
- Why?
  - Incremental errors can creep into a system when one copy is changed but the others are not
  - Isolation of Change Requests: We want to go to ONE place when responding to a change request

Example (I)

- Duplication of Code: Closing the Door in Chapter 2



  - We had the responsibility for closing the door automatically in our “dog door” example originally living in the RemoteControl Class.

- When we added a BarkRecognizer Class to the system, it opened the door automatically but failed to close the door

  - We could have placed a copy of the code to automatically close the door in BarkRecognizer but that would have violated the DRY principle

- Instead, we moved the responsibility to the shared Door class

Example (II)

- DRY is really about ONE requirement in ONE place
  - We want each responsibility of the system to live in a single, sensible place
- New Requirements for the Dog Door System: Beware of Duplicates
  - The dog door should alert the owner when something inside the house gets too close to the dog door
  - The dog door will open only during certain hours of the day
  - The dog door will be integrated into the house's alarm system to make sure it doesn't activate when the dog door is open
  - The dog door should make a noise if the door cannot open because of a blockage outside
  - The dog door will track how many times the dog uses the door
  - When the door closes, the house alarm will re-arm if it was active before the door opened

Example (III)

- New Requirements for the Dog Door System: Beware of Duplicates
  - The dog door should alert the owner when something inside the house gets too close to the dog door
  - The dog door will open only during certain hours of the day
  - The dog door will be integrated into the house's alarm system to make sure it doesn't activate when the dog door is open
  - The dog door should make a noise if the door cannot open because of a blockage outside
  - The dog door will track how many times the dog uses the door
  - When the door closes, the house alarm will re-arm if it was active before the door opened

Example (III)

- New Requirements for the Dog Door System
  - The dog door should alert the owner when something is too close to the dog door
  - The dog door will open only during certain hours of the day
  - The dog door will be integrated into the house's alarm system
  - The dog door will track how many times the dog uses the door
- Duplicates Removed!

Example (IV)

- Ruby on Rails makes use of DRY as a core part of its design
  - focused configuration files; no duplication of information
  - for each request, often single controller, single model update, single view
- But, prior to Ruby on Rails 1.2, there was duplication hiding in the URLs used by Rails applications
  - `POST /people/create    # create a new person`
  - `GET /people/show/1     # show person with id 1`
  - `POST /people/update/1  # edit person with id 1`
  - `POST /people/destroy/1 # delete person with id 1`

Example (V)

- The duplication exists between the HTTP method name and the operation name in the URL
  - `POST /people/create`
- Recently, there has been a movement to make use of the four major “verbs” of HTTP
  - PUT/POST == create information (create)
  - GET == retrieve information (read)
  - POST == update information (update)
  - DELETE == destroy information (destroy)
- These verbs mirror the CRUD operations found in databases
  - Thus, saying “create” in the URL above is a duplication

Example (VI)

- In version 1.2, Rails eliminates this duplication for something called “resources”
- Now URLs look like this:
  - POST /people
  - GET /people/1
  - PUT /people/1
  - DELETE /people/1
- And the duplication is **logically** eliminated
  - Disclaimer: … but not actually eliminated… Web servers do not universally support PUT and DELETE “out of the box”. As a result, Rails uses POST
    - POST /people/1
      Post-Semantics: Delete

Single Responsibility Principle

- Every object in your system should have a single responsibility, and all the object's services should be focused on carrying it out
  - This is obviously related to the "One Reason to Change" principle
  - If you have implemented SRP correctly, then each class will have only one reason to change
- The "single responsibility" doesn't have to be "small", it might be "manage units" in Gary's Game System Framework
- We've encountered SRP before
  - SRP == high cohesion
  - "One Reason To Change" **promotes** SRP
  - DRY is often used to achieve SRP

Return to Textual Analysis

- One way of identifying high cohesion in a system is to do the following
  - For each class C
    - For each method M
      - Write “The C Ms itself”
  - Examples
    - The Automobile drives itself
    - The Automobile washes itself
    - The Automobile starts itself
- If any one of these sentences doesn’t make sense then investigate further. You may have discovered a service that belongs to a different responsibility of the system and should be moved to a different class
  - This may require first creating a new class before performing the move

SRP in Action

- We've seen SRP used in several places over the last eight lectures
    - Automatically closing the door in the dog door example
    - InstrumentSpec handling all instrument-related properties in Rick's Guitars
    - Instrument handling all inventory-related properties in Rick's Guitars
    - Board handling board-related services in the Game System Framework
    - Unit handling all property-related functionality in the Game System Framework
- Essentially any time we've seen a highly cohesive class!

Liskov Substitution Principle

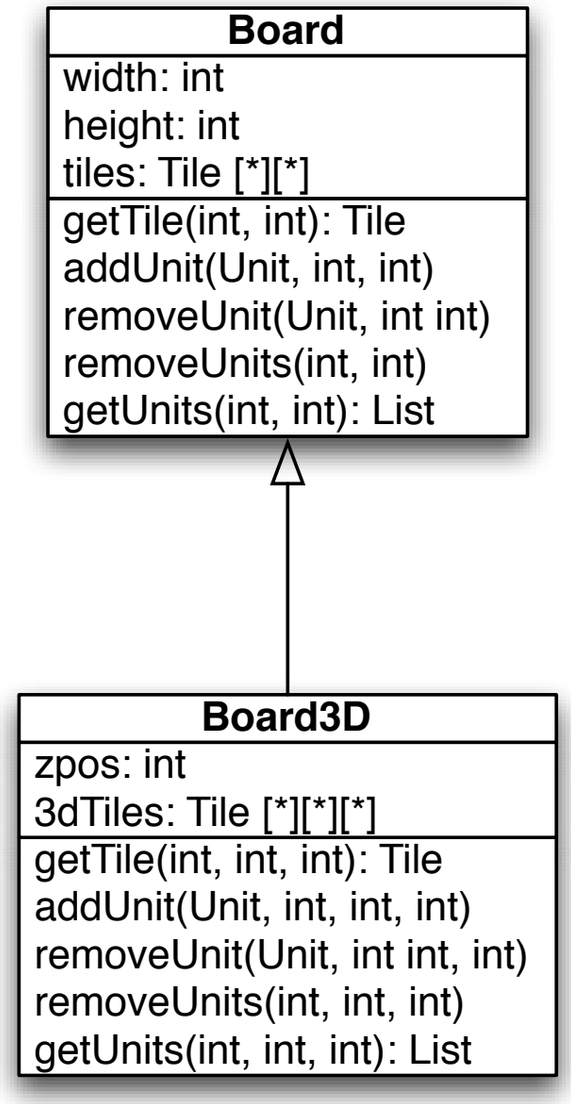
- Subtypes must be substitutable for their base types
- Basic Idea
  - Instances of subclasses do not violate the behaviors exhibited by instances of their superclasses
    - They may constrain that behavior but they do not **contradict** that behavior
- Named after Barbara Liskov who co-authored a paper with Jeannette Wing in 1993 entitled ***Family Values: A Behavioral Notion of Subtyping***
  - *Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T.*
- Properties that hold on superclass objects, hold on subclass objects
  - **Return to Rectangle/Square**: *WidthAndHeightMayBeDifferent(Rectangle)* equals true for Rectangles and equals false for Square

Well-Designed Inheritance

- LSP is about well-designed inheritance
  - When I put an instance of a subclass in a place where I normally place an instance of its superclass
    - the functionality of the system must remain **correct**
    - (not necessarily the **same**, but correct)

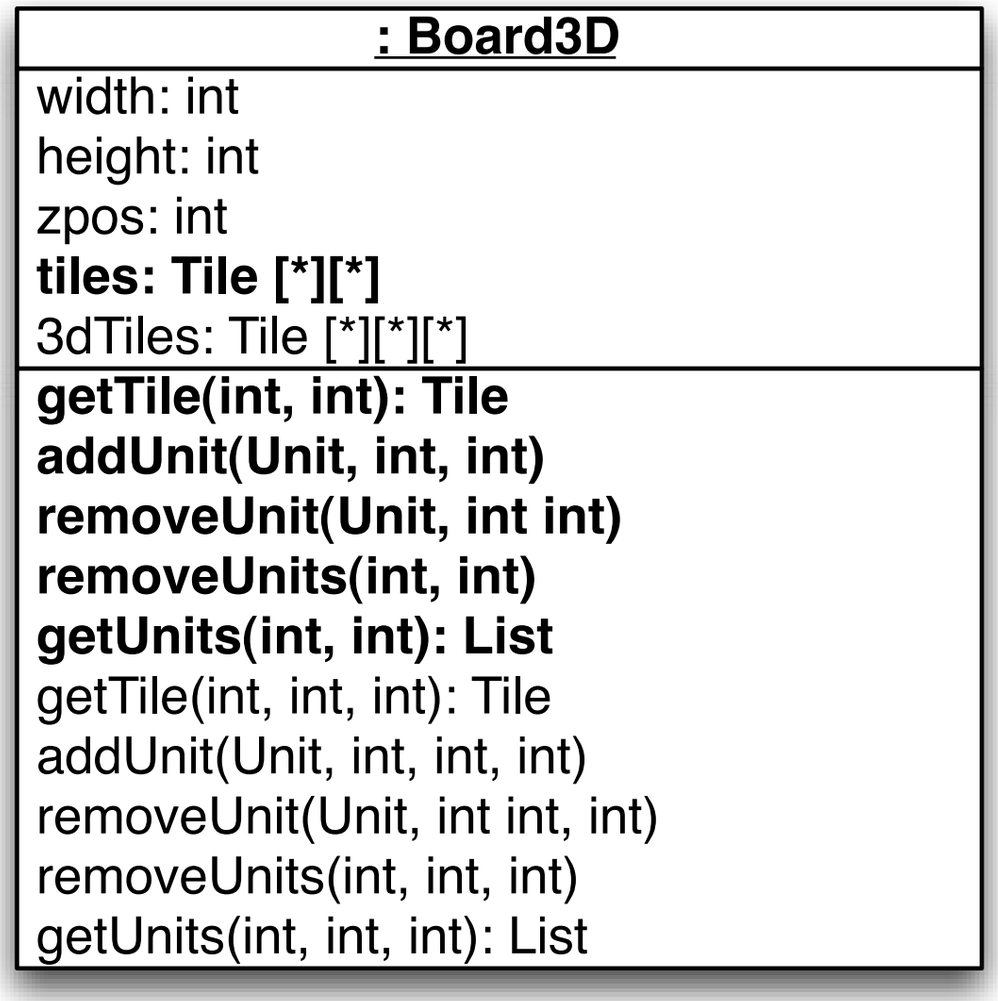
Bad Example (I)

- The book provides an example of misusing inheritance (and violating the LSP)
  - Extend Board to produce Board3D



Bad Example (II)

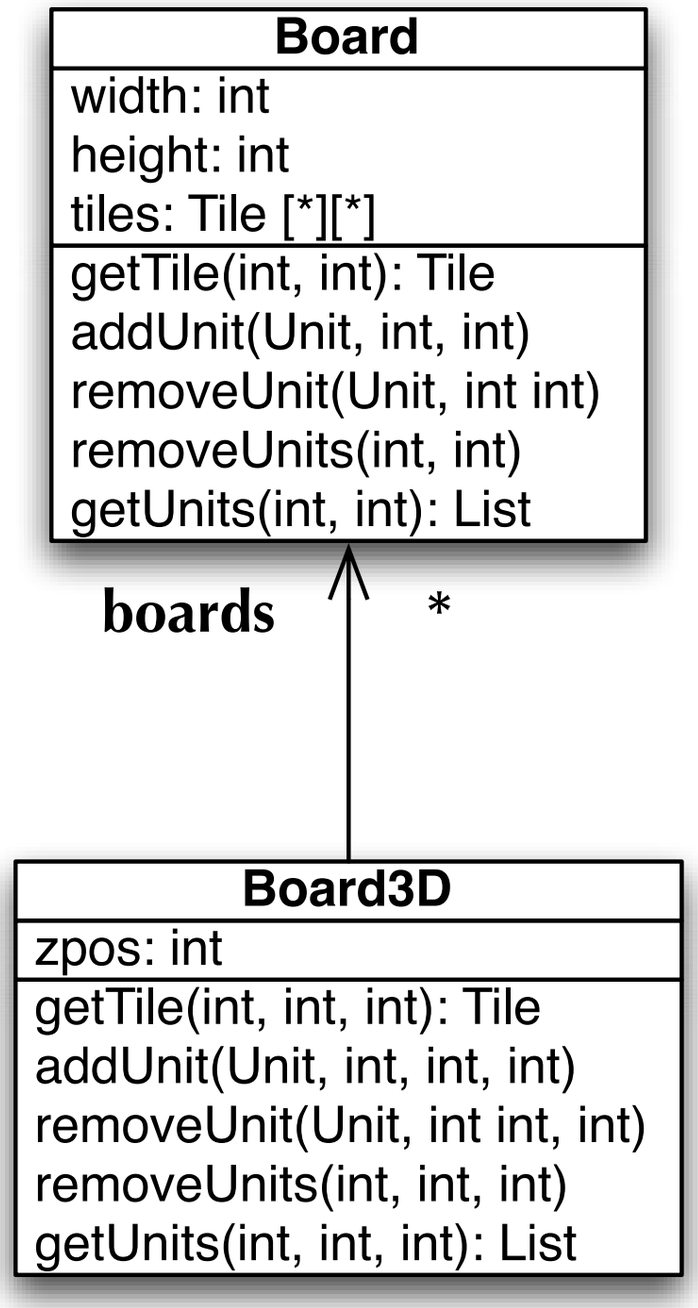
- But this means that an instance of Board3D looks like this:
  - Each attribute and method in bold is meaningless in this object
  - Board3D is getting nothing useful from Board except for width and height!!
  - We certainly could NOT create a Board3D object and hand it to code expecting a Board object!
  - As a result, this design violates the LSP principle
  - How to fix?



Delegation to the Rescue! (Again)

- You can understand why the Game System Framework thought they could extend Board when creating Board3D
  - Board has a lot of useful functionality and a Board3D should try to reuse that functionality as much as possible
  - However, the Board3D has no need to CHANGE that functionality and the Board3D doesn't really behave in the same way as a board
    - For instance, a unit on "level 10" may be able to attack a unit on "level 1"; such functionality doesn't make sense in the context of a 2D board
- Thus, if you need to use functionality in another class, but you don't want to change that functionality, consider using **delegation** instead of inheritance
  - Inheritance was simply the wrong way to gain access to the Board's functionality
  - Delegation is when you hand over the responsibility for a particular task to some other class or method

New Class Diagram



Board3D now maintains a list of Board objects for each legal value of “zpos”

It then delegates to the Board object to handle the requested service

```
public Tile getTile(int x, int y, int z) {
    Board b = boards.get(z);
    return b.getTile(x,y);
}
```

Note: book gets UML diagram wrong on page 405. The “3dTiles: Tile [*][*][*] attribute is eliminated with this new design

Another Take on Composition

- The book defines composition as
  - Composition allows you to use behavior from a family of classes, and to change that behavior at runtime
    - Their definition is essentially equivalent to the Strategy design pattern
- Delegation is useful when the behavior of the object you're delegating to never changes
  - Delegation is still used in composition, but the object that you are delegating to can change at run-time
- Example: Unit and Weapon
  - A unit can invoke the attack() method on its Weapon; as the game progresses, the unit may switch among different weapons at will
  - The unit is composing its "attack behavior" out of a number of Weapon instances; existence dependency applies; delete unit ⇒ delete weapons

Another Take on Aggregation

- In composition, the object composed of other behaviors owns those behaviors. When the object is destroyed, so are all of its behaviors
    - The behaviors in a composition do not exist outside of the composition itself
- If this is not what you want, then use aggregation: composition without the abrupt ending
    - Aggregation is when one class is used as part of another class, but still exists outside of that other class
    - The book uses an example of a Unit that can arrive at a building and leave its weapons there in storage, the relationship between Unit and Weapon is now an aggregation

Implication: Use Inheritance Sparingly

- Delegation, composition, and aggregation all offer alternatives to inheritance when you need to reuse the behavior of another class
    - Only use inheritance when
        - an IS-A relationship exists between the superclass and the subclass
            - AND the subclass behaves like a superclass (i.e. maintains the properties of the superclass in its behavior)
- If you favor delegation, composition, and aggregation **over** inheritance, your software will usually be more flexible and easier to maintain, extend, and reuse
    - This was the subject of a religious war during the 90s
        - Unlike “emacs vs. vi”, the war is over and delegation won!

Wrapping Up: Chapter 8

- We've added four new OO principles to our toolkit
  - Apply these principles and you'll see a marked increase in the flexibility and extensibility of your OO designs
  - Indeed, one of the "secrets" of design patterns is that they invariably lead to code that exhibit these principles
- We've also seen that inheritance is a tool to be used sparingly
  - Favor delegation, composition, and aggregation to gain run-time flexibility
  - Use inheritance when the subclass's semantics and behavior fit neatly with its superclass
- On to Chapter 9…

Software Life Cycles (aka Design Methods)

- Software life cycles break up the development process into stages
  - Each stage has a goal and an associated set of tasks and documents
- Traditional stages:
  - analysis, design, imp, test, deploy, maintenance, retirement
- To move forward in a life cycle, two things are fundamental
  - Iteration
    - You won’t get it right the first time; Enables Divide and Conquer
  - Testing
    - How do you show your customer that progress is being achieved?

Goal: Make Customer Happy

- We've given you lots of "tools" over the last nine lectures
  - OO Principles
  - Requirements, Analysis, and Design Techniques
  - Simple Software Life Cycle
    - aka "the three steps to writing great software"
  - Software Architecture Techniques
    - feature lists, use case diagrams, decomposing problem space
- **None of them matter, if you can't keep your customer happy**
  - Iteration and testing provide the means for externalizing results to the customer, demonstrating concrete progress
    - The book equates progress with "test cases applied to working code"

Iteration (I)

- The key question is how do you “organize” your iterations
- Two Approaches
  - Feature Driven Development
    - Pick a specific feature in your app; then plan, analyze, and develop that feature **to completion** in **one** iteration
  - Use Case Driven Development
    - Pick a scenario in a use case (one path) and write code to support that **complete scenario** through the use case in **one** iteration
      - If it makes sense to tackle the entire use case, then do so
- The former focuses on functionality; the latter focuses on tasks
  - The former will often be limited to a single class or a small set of classes
  - The latter may touch a lot of classes on multiple layers of your architecture

Iteration (II)

- Both feature driven development and use case driven development
  - depend on good requirements (which come from the customer)
  - deliver what the customer wants
- In feature driven development, you start with your feature list then
  - pick a feature
  - implement it
  - repeat (until done)
- In use case driven development, you start with your use case diagram
  - pick a use case and write it
  - implement it
  - repeat (until done)

Iteration (III)

- Feature driven development is more granular
  - Works well when you have a lot of different features with minimal overlap
  - Allows you to show working code faster (smaller chunks)
- Use case driven development is more “big picture”
  - Works well when your app has lots of tasks and actors it must support
  - Allows you to show the customer bigger pieces of functionality (i.e. tasks) after each iteration
  - Is user centric; focus is on a single task for a single user on each iteration
- Iterations will likely be shorter for feature driven development (days; weeks) than use case driven development (weeks; months)
  - Consider that in use case driven development, during your FIRST iteration, you may have to develop a user interface, controller classes, model classes, and handle persistence!

Example: Feature Driven Development

Features for Gary's Game System

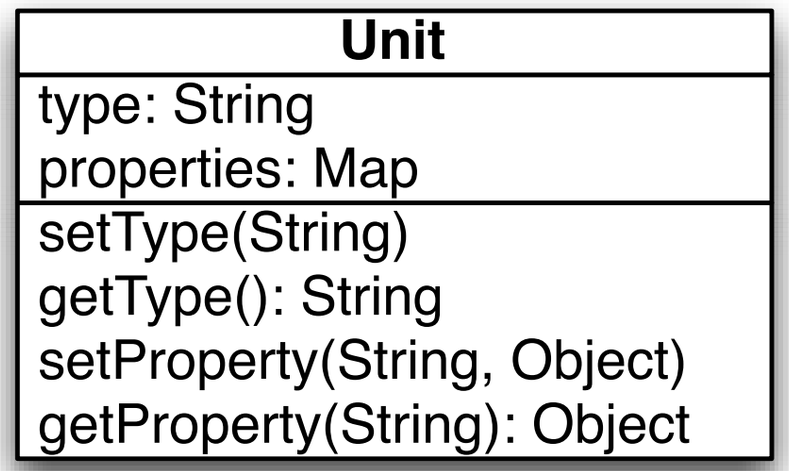
1. Supports different time periods, including fictional periods like sci-fi and fantasy
2. Supports add-on modules for additional campaigns or battle scenarios
3. Supports different types of terrain
4. Supports multiple types of troops or units that are game-specific
5. Each game has a board, made up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

Lets try feature driven development, starting with **feature four** of the Game System Framework

Here's our Unit class from Lec. 11

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

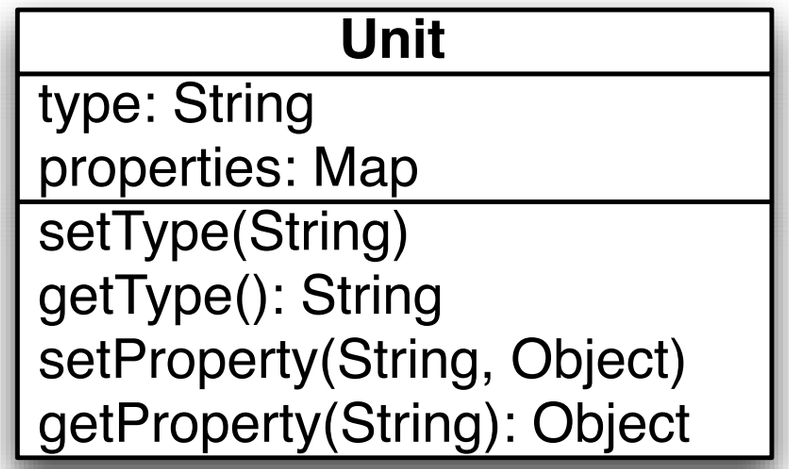
Is it complete?



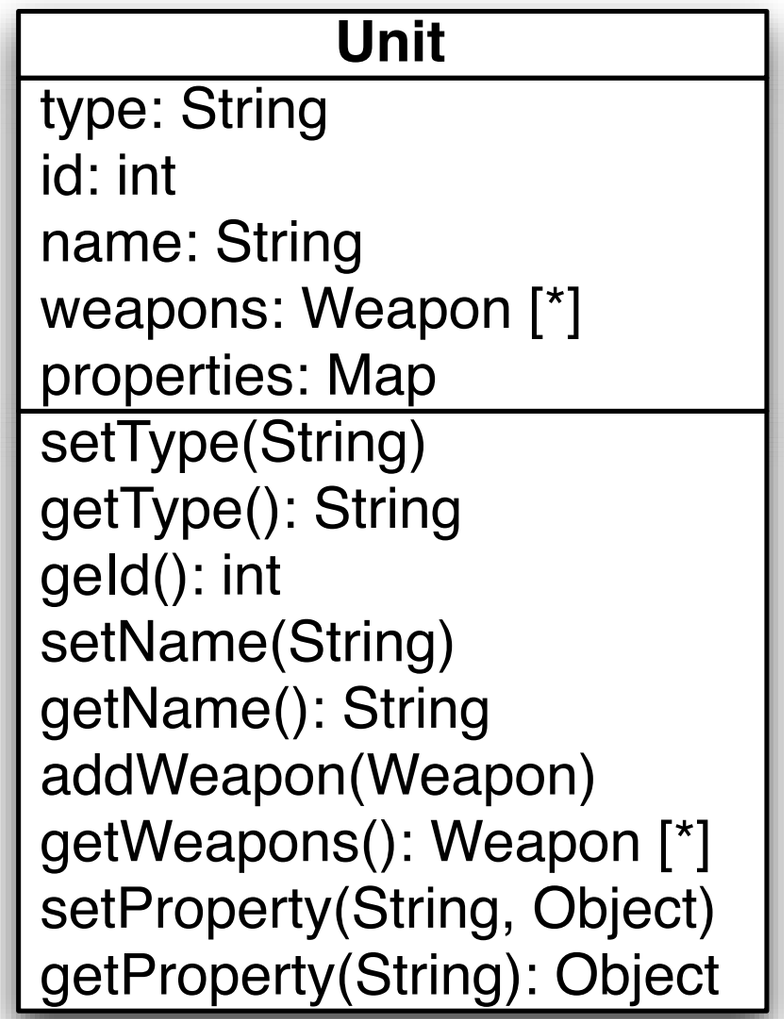
The book returns to Gary's vision statement and discovers that our initial design work missed some requirements!

1. Each unit has game-specific properties (**done, well maybe**)
2. Each unit can move from one tile to another on the board (**punt**)
3. Units can be grouped together into armies (**whoops!**)

Skipping A Section



vs.



The book has a section on whether Unit needs to be redesigned, pulling properties common to all Units out of the properties Map.

The point is to examine the trade-offs with each of these design choices and to emphasize the need to always evaluate your past design decisions.

How to Show Progress?

- Tests!
  - As we iterate on our design/code, we can demonstrate progress to our customer with test cases applied to working code
- Different types of tests
  - **Unit tests**: applied to individual classes
  - **Integration tests**: applied to groups of classes that interact to implement a particular scenario or task
  - **System tests**: applied to the entire system to determine if it meets its requirements
- Test driven development
  - All production code is written to make failing test cases pass
  - Means: write test case first, have it fail, then write code that makes it pass

Tests for Unit

- Test that you can set a property to a particular value and then retrieve that specific value for that property at a later time
- Test that a property value can be changed
- Test retrieving a value for an undefined property
  - Need to define what happens in this instance
- You should test your software for every possible usage (that you can identify)
- Be sure to test incorrect usage; it will help you **design your approach to error handling** and it will help you catch bugs early

Anatomy of a Test Case/Test Run

- The parts of a test care are
  - A name
  - A description
  - A specified input
  - An expected output
- The parts of a test run are
  - Code to execute test cases
  - A pass/fail value for each test
- Test process
  - Run test cases, fix problems, repeat until all tests pass

Demonstration

- Source for Unit
- Source code for each test
- Source code for testing framework

- Note: book has an excellent method for developing your test suite
  - Table based approach that uses the columns
    - id, description, input, expected output, and starting state
    - With respect to latter, test cases typically require initialization
      - e.g. to test client-server interaction, a server must be initialized

How do we predict expected output?

- Most of the time it falls out from the functionality
- But, sometimes, it depends on the **contract** of the class
    - especially with respect to error handling
- Programming by Contract (aka Design by Contract)
    - When you program by contract, you and your software's users are agreeing that your software will behave in a certain way
        - Such as returning "null" for non-existent properties
        - We could throw an exception instead
    - Programming by Contract is about trusting programmers to use your API correctly
- Unit's Contract
    - Hey, you look pretty smart. I'm going to return null values for non-existent properties. You can handle the null values, OK?

The alternative? Don’t Trust Your Users

- Defensive programming
  - If you don’t trust your software’s users, you must adopt a coding style called defensive programming
    - in which all input is suspect and errors are handled via exceptions
  - Defensive programming assumes the worst and tries to protect itself against misuse and/or bad data
    - Sometimes this is appropriate, for instance, when your software is available to the general public via a Web browser
- Defensive version of getProperty()

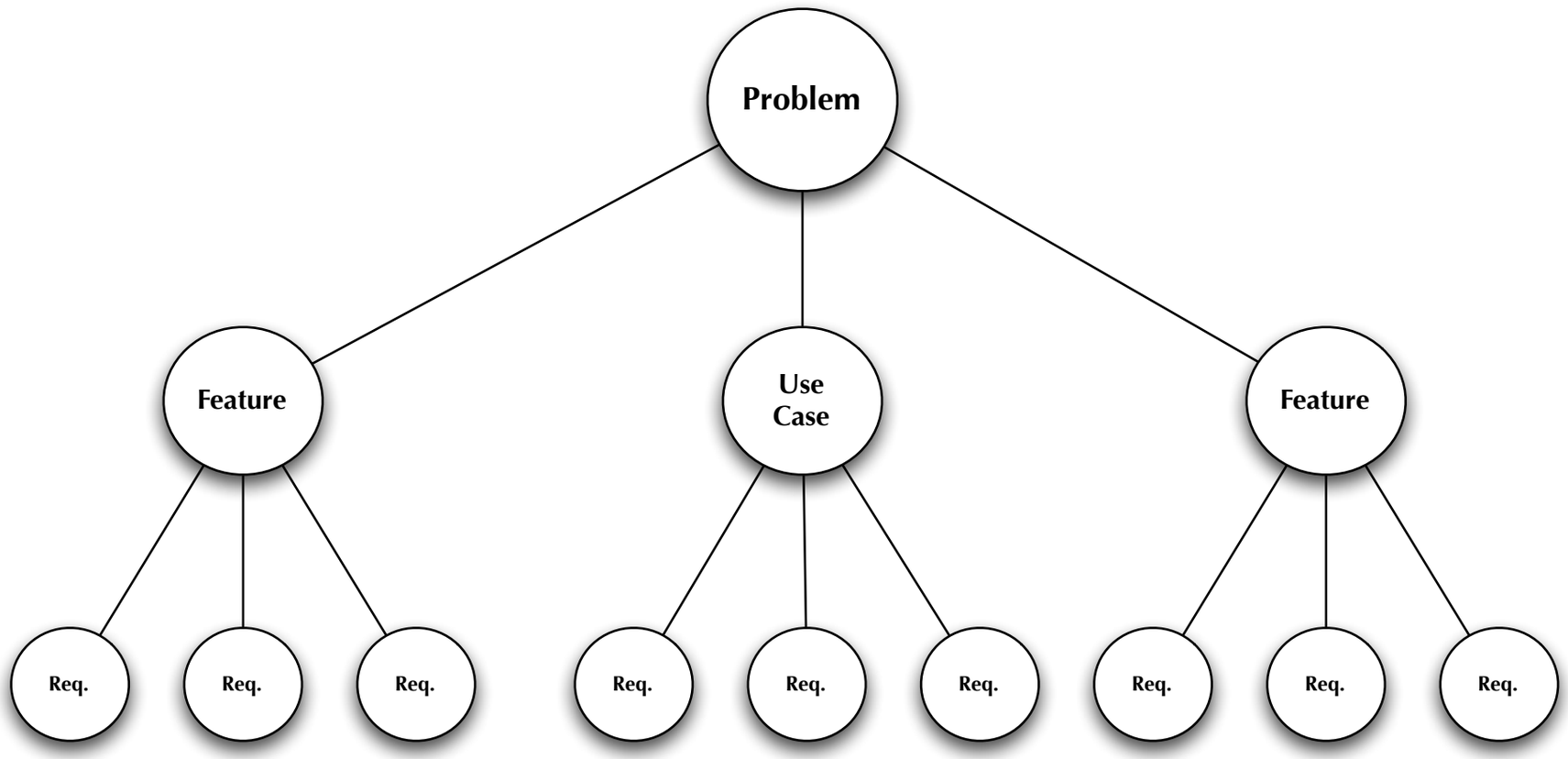
```
1  public Object getProperty(String property) throws IllegalAccessException {
2      if (properties == null) {
3          throw new IllegalAccessException("Unit properties not initialized.");
4      }
5      Object value = properties.get(property);
6      if (value == null) {
7          throw new IllegalAccessException("Non-Existent Property.");
8      }
9      return value;
10 }
```

The Trade-Offs

- When you are programming by contract, you're working with client code to **agree** on how you'll handle problem situations
    - Pick a style and stick with it
- When you're programming defensively, you're making sure the client gets a "**safe**" response, **no matter what the client wants** to have happen
- This style results in more work for the client programmer
    - API code contains checked exceptions that require explicit exception handlers
    - API results are carefully examined and validated before used

Feature 2: Unit Movement

- We already decided to punt on unit movement back in lecture 11
  - Since we made that decision at the architecture level, the same decision applies at the feature level
  - This is typical in problem decomposition, decisions made at higher levels can influence the work and decisions made at lower levels



Feature 3: Supporting Unit Groups

- Create UnitGroup Class
- Create Test Cases

Unit Group
units: Map
addUnit(Unit) removeUnit(int) removeUnit(Unit) getUnit(int): Unit getUnits(): Unit[*]

Description	Input	Expected Output	Starting State
Add Unit to Group	Unit with Id of 100	UnitGroup with single Unit	UnitGroup with no members
Get Unit by ID	100	Unit with Id of 100	UnitGroup containing Unit 100
Get All Units	N/A	List of Units	UnitGroup with >1 members
…	…	…	…

Wrapping Up: Chapter 9

- Software Life Cycles
    - Iteration and testing are fundamental to achieving progress
- Development Approaches
    - Use case driven development: implement single use case, repeat
    - Feature driven development: implement single feature, repeat
    - Test driven development: write a test first, watch it fail, write code, watch test pass
- Programming Practices
    - Programming by Contract: agreement about how software behaves
    - Defensive Programming: Trust No One; extensive error/data checking

Coming Up Next

- Lecture 13: Putting It All Together
    - Read Chapter 10 of the OO A&D book
- Lecture 14: Midterm Review
    - Midterm will be held on Tuesday, Oct. 13th