Iterating And Testing

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/6448 — Lecture 13 — 10/07/2008

© University of Colorado, 2008
Lecture Goals

• 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
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 terrain type.
6. The framework keeps up with whose turn it is and coordinates basic movement

Here’s our Unit class from Lec. 11

<table>
<thead>
<tr>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>type: String</td>
</tr>
<tr>
<td>properties: Map</td>
</tr>
<tr>
<td>setType(String)</td>
</tr>
<tr>
<td>getType(): String</td>
</tr>
<tr>
<td>setProperty(String, Object)</td>
</tr>
<tr>
<td>getProperty(String): Object</td>
</tr>
</tbody>
</table>

Let's try feature driven development, starting with **feature four** of the Game System Framework.
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!)*
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. Questions?
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 case 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()

```java
public Object getProperty(String property) throws IllegalAccessException {
    if (properties == null) {
        throw new IllegalAccessException("Unit properties not initialized.");
    }
    Object value = properties.get(property);
    if (value == null) {
        throw new IllegalAccessException("Non-Existent Property.");
    }
    return value;
}
```
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

<table>
<thead>
<tr>
<th>Description</th>
<th>Input</th>
<th>Expected Output</th>
<th>Starting State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Unit to Group</td>
<td>Unit with Id of 100</td>
<td>UnitGroup with single Unit</td>
<td>UnitGroup with no members</td>
</tr>
<tr>
<td>Get Unit by ID</td>
<td>100</td>
<td>Unit with Id of 100</td>
<td>UnitGroup containing Unit 100</td>
</tr>
<tr>
<td>Get All Units</td>
<td>N/A</td>
<td>List of Units</td>
<td>UnitGroup with &gt;1 members</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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

• 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 15: Putting It All Together
  • Read Chapter 10 of the OO A&D book
• Lecture 16: OO Design Methods
  • None; Will compare textbook’s life cycle with other design methods