

# Test-Driven Development

---

Kenneth M. Anderson  
University of Colorado, Boulder  
CSCI 4448/5448 — Lecture 28 — 04/21/11

© University of Colorado, 2011

# Credit where Credit is Due

---

- Some of the material for this lecture is taken from “Test-Driven Development” by Kent Beck
  - as such some of this material is copyright © Addison Wesley, 2003
- In addition, some material for this lecture is taken from “Agile Software Development: Principles, Patterns, and Practices” by Robert C. Martin
  - as such some materials is copyright © Pearson Education, Inc., 2003
- Finally, one of the examples is inspired by the Roman Numerals example that is featured in Dive into Python 3 <<http://diveintopython3.org/>> by Mark Pilgrim. The slides devoted to that example are thus distributed using the following license: <<http://creativecommons.org/licenses/by-sa/3.0/>>.

# Goals for this lecture

---

- Introduce the concept of Test-Driven Development (TDD)
- Present several examples
- More executive summaries!

# Test-Driven Development

---

- The idea is simple
  - No *production* code is written **except to make a failing test pass**
- Implication
  - You have to write test cases **before** you write code
- Note: use of the word “production”
  - which refers to code that is going to be deployed to and used by real users
- It does not say: “No code is written except...”

# Test-Driven Design in One Slide or Less

---

- This means that when you first write a test case, you may be testing code that does not exist
  - And since that means the test case will not compile, obviously the test case “fails”
    - After you write the skeleton code for the objects referenced in the test case, it will now compile, but also may not pass
- So, then you write the simplest code that will make the test case pass

# Example (I)

---

- Consider writing a program to score the game of bowling
- You might start with the following test

```
public class TestGame extends TestCase {  
    public void testOneThrow() {  
        Game g = new Game();  
        g.addThrow(5);  
        assertEquals(5, g.getScore());  
    }  
}
```

- When you compile this program, the test “fails” because the Game class does not yet exist. But:
  - You have defined two methods on the class that you want to use
  - You are designing this class from a client’s perspective

# Example (II)

---

- You would now write the Game class

```
public class Game {  
    public void addThrow(int pins) {  
    }  
    public int getScore() {  
        return 0;  
    }  
}
```

- The code now compiles but the test will still fail: getScore() returns 0 not 5
  - In Test-Driven Design, Beck recommends taking small, simple steps
  - So, we get the test case to compile before we get it to pass

# Example (III)

---

- Once we confirm that the test still fails, we would then write the simplest code to make the test case pass; that would be

```
public class Game {  
    public void addThrow(int pins) {  
    }  
  
    public int getScore() {  
        return 5;  
    }  
}
```

- The test case now passes!



# Example (IV)

---

- But, this code is not very useful!
- Lets add a new test case to enable progress

```
public class TestGame extends TestCase {  
    public void testOneThrow() {  
        Game g = new Game();  
        g.addThrow(5);  
        assertEquals(5, g.getScore());  
    }  
    public void testTwoThrows() {  
        Game g = new Game()  
        g.addThrow(5)  
        g.addThrow(4)  
        assertEquals(9, g.getScore());  
    }  
}
```

- The first test passes, but the second case fails (since  $9 \neq 5$ )
  - This code is written using JUnit; it uses reflection to invoke tests automatically

# Example (V)

---

- We have duplication of information between the first test and the Game class
  - In particular, the number 5 appears in both places
  - This duplication occurred because we were writing the simplest code to make the test pass
  - Now, in the presence of the second test case, this duplication does more harm than good
  - So, we must now refactor the code to remove this duplication

# Example (VI)

---

```
public class Game {  
    private int score = 0;  
    public void addThrow(int pins) {  
        score += pins;  
    }  
    public int getScore() {  
        return score;  
    }  
}
```

Both tests now pass. Progress!

# Example (VII)

---

- But now, to make additional progress, we add another test case to the TestGame class

...

```
public void testSimpleSpare() {  
    Game g = new Game()  
    g.addThrow(3); g.addThrow(7); g.addThrow(3);  
    assertEquals(13, g.scoreForFrame(1));  
    assertEquals(16, g.getScore());  
}
```

...

- We're back to the code not compiling due to scoreForFrame()
  - We'll need to add a method body for this method and give it the simplest implementation that will make all three of our tests cases pass

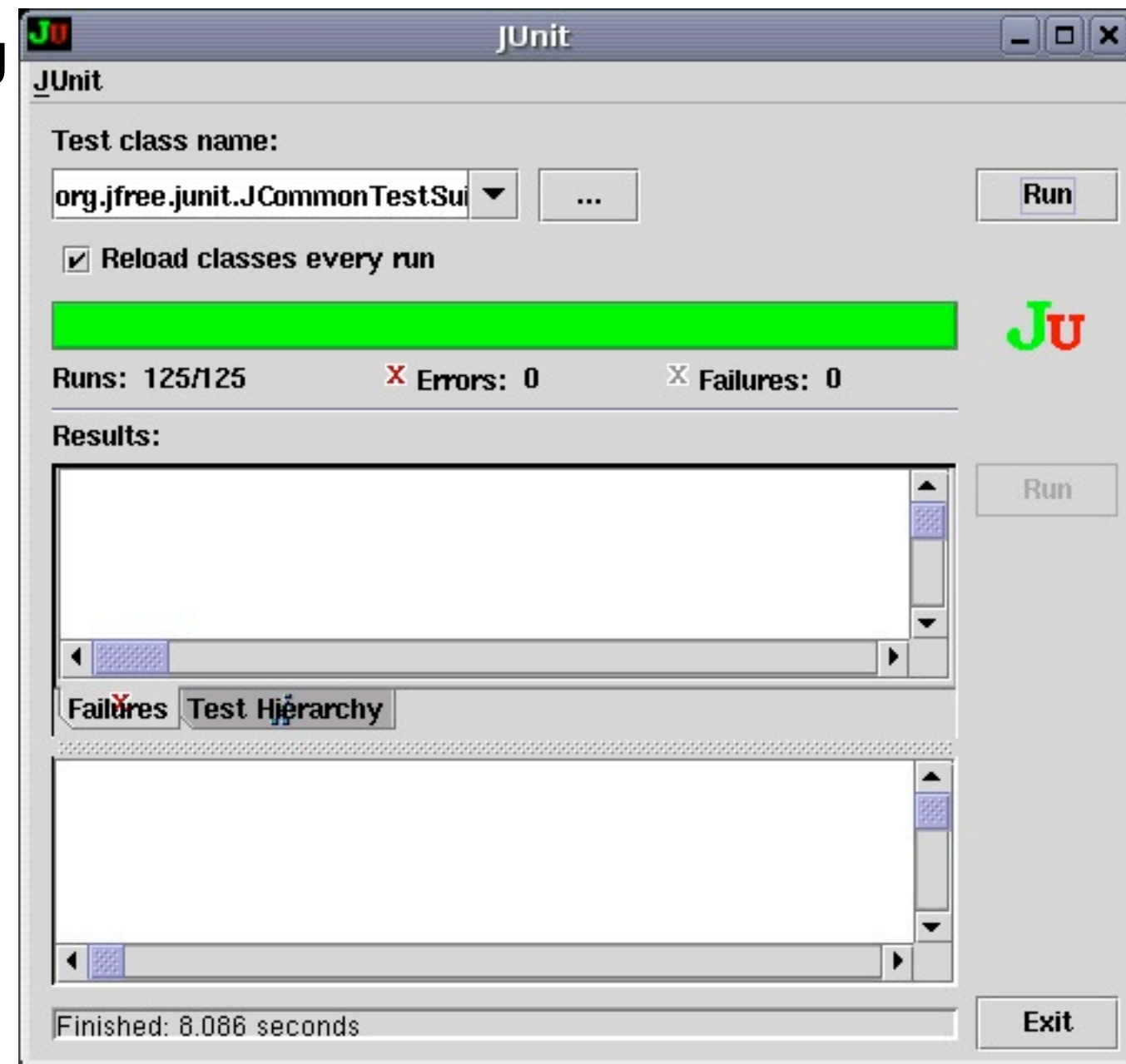
# TDD Life Cycle

---

- The life cycle of test-driven development is
  - Quickly add a test
  - Run all tests and see the new one fail
  - Make a simple change
  - Run all tests and see them all pass
  - Refactor to remove duplication
- This cycle is followed until you have met your goal;
  - note that this cycle simply adds testing to the “add functionality; refactor” loop covered in the our two lectures on refactoring

# TDD Life Cycle, continued

- Kent Beck likes to perform TDD using a testing framework, such as JUnit.
- Within such frameworks
  - failing tests are indicated with a “red bar”
  - passing tests are shown with a “green bar”
- As such, the TDD life cycle is sometimes described as
  - “red bar/green bar/refactor”



# JUnit: Red Bar...

- When a test fails:
  - You see a red bar
  - Failures/Errors are listed
  - Clicking on a failure displays more detailed information about what went wrong



# Example Background: Multi-Currency Money

---

- Lets design a system that will allow us to perform financial transactions with money that may be in different currencies
  - e.g. if we know that the exchange rate from Swiss Francs to U.S. Dollars is 2 to 1 then we can calculate expressions like
    - $5 \text{ USD} + 10 \text{ CHF} = 10 \text{ USD}$
  - or
    - $5 \text{ USD} + 10 \text{ CHF} = 20 \text{ CHF}$



# Starting From Scratch

---

- Lets start developing such an example
- How do we start?
  - TDD recommends writing a list of things we want to test
  - This list can take any format, just keep it simple
  - Example
    - $\$5 + 10 \text{ CHF} = \$10$  if rate is 2:1
    - $\$5 * 2 = \$10$

# First Test

---

- The first test case looks a bit complex, lets start with the second
  - $5 \text{ USD} * 2 = 10 \text{ USD}$
- First, we write a test case

```
public void testMultiplication() {  
    Dollar five = new Dollar(5);  
    five.times(2);  
    assertEquals(10, five.amount)  
}
```

# Discussion on Test Case

---

```
public void testMultiplication() {  
    Dollar five = new Dollar(5);  
    five.times(2);  
    assertEquals(10, five.amount)  
}
```

- What benefits does this provide?
  - target class plus some of its interface
    - we are designing the interface of the Dollar class by thinking about how we would want to use it
  - We have made a testable assertion about the state of that class after we perform a particular sequence of operations

# What's Next?

---

- We need to update our test list
  - The test case revealed some things about Dollar that we will want to address
    - We are representing the amount as an integer, which will make it difficult to represent values like 1.5 USD; how will we handle rounding of fractional amounts?
    - Dollar.amount is public; violates encapsulation
    - What about side effects?; we first declared our variable as “five” but after we performed the multiplication it now equals “ten”

# Update Testing List

---

- The New List
  - $5 \text{ USD} + 10 \text{ CHF} = 10 \text{ USD}$
  - $\$5 * 2 = \$10$
  - make “amount” private
  - Dollar side-effects?
  - Money rounding?
- Now, we need to fix the compile errors
  - no class Dollar, no constructor, no method: times(), no field: amount

# First version of Dollar Class

---

```
public class Dollar {  
    public Dollar(int amount) {  
    }  
  
    public void times(int multiplier) {  
    }  
  
    public int amount;  
}
```

- Now our test compiles and fails!

# Too Slow?

---

- Note: we did the simplest thing to make the test compile;
- now, we are going to do the simplest thing to make the test pass
- Is this process too slow?
  - **YES**, as you get familiar with the TDD life cycle you will gain confidence and make bigger steps
  - **NO**, taking small simple steps avoids mistakes;
    - beginning programmers try to code too much before invoking the compiler;
    - they then spend the rest of their time debugging!

# How do we make the test pass?

---

- Here's one way

```
public void times(int multiplier) {  
    amount = 5 * 2;  
}
```

- The test now passes, we received a “green bar”!
- Now, we need to “refactor to remove duplication”
  - But where is the duplication?



# Refactoring

---

- To remove the duplication of the test data and the hard-wired code of the times method, we think the following
- “We are trying to get a 10 at the end of our test case and we’ve been given a 5 in the constructor and a 2 was passed as a parameter to the times method”
  - So, lets connect the dots...

# First version of Dollar Class

---

```
public class Dollar {  
    public Dollar(int amount) {  
        this.amount = amount;  
    }  
    public void times(int multiplier) {  
        amount = amount * multiplier;  
    }  
    public int amount;  
}
```

- Now our test compiles and passes, and we didn't have to cheat!

# One loop complete!

---

- Before writing the next test case, we update our testing list
  - 5 USD + 10 CHF = 10 USD
  - ~~\$5 \* 2 = \$10~~
  - make “amount” private
  - Dollar side-effects?
  - Money rounding?

# One more example

---

- Lets address the “Dollar Side-Effects” item and then move on to another example
- Lets write the next test case
  - When we called the times operation our variable “five” was pointing at an object whose amount equaled “ten”; not good
    - the times operation had a side effect which was to change the value of a previously created “value object”
    - Think about it, as much as you might like to, you can’t change a 5 dollar bill into a 500 dollar bill; the 5 dollar bill remains the same throughout multiple financial transactions

# Next test case

---

- The behavior we want is

```
public void testMultiplication() {  
    Dollar five = new Dollar(5);  
    Dollar product = five.times(2);  
    assertEquals(10, product.amount);  
    product = five.times(3);  
    assertEquals(15, product.amount);  
    assertEquals(5, five.amount);  
}
```

# Test fails

---

- The test fails because it won't compile;
- We need to change the signature of the times method; previously it returned void and now it needs to return Dollar

```
public Dollar times(int multiplier) {  
    amount = amount * multiplier;  
    return null;  
}
```

- The test compiles but still fails; as Kent Beck likes to say “Progress!”

# Test Passes

---

- To make the test pass, we need to return a new Dollar object whose amount equals the result of the multiplication

```
public Dollar times(int multiplier) {  
    return new Dollar(amount * multiplier);  
}
```

- Test Passes;
- Cross “Dollar Side Effects?” off the testing list; second loop complete!
- There was no need to refactor in this situation

# Discussion of the Example

---

- There is still a long way to go
  - only scratched the surface
- But
  - we saw the life cycle performed twice
  - we saw the advantage of writing tests first
  - we saw the advantage of keeping things simple
  - we saw the advantage of keeping a testing list to keep track of our progress
- Plus, as we write new code, we will know if we are breaking things because our old test cases will fail if we do;
  - if the old tests stay green, we can proceed with confidence



# Roman Numerals (I)

---

- Let's develop a class that can manipulate roman numerals
  - Roman numerals can express integers from 1 to 3999
- They do this using the following set of symbols that map to the following values
  - I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, M = 1000
- There are rules concerning how these characters can be combined
  - For instance, the 10s characters (X,C,M) can be repeated up to three times
  - The 5s characters (V, L, D) cannot be repeated
  - Character sequences can be additive (III = 3) or subtractive (IX = 9)
    - Can be complex 99 is written as XCIX (100-10 + 10-1)

# Roman Numerals (II)

---

- We start by developing a testing list
  - able to convert legal roman numerals to integers
  - able to convert integers in the range 1 to 3999 into roman numerals
  - able to add two roman numerals, checking for boundary conditions
  - able to subtract two roman numerals, checking for boundary conditions
- We will not complete the example but we'll make progress on a few of these

# Test Case: Create a Roman Numeral, Get Its Value

---

- Let's use Python's Unit Test framework
  - We write the test case as if all the code we need is available

```
1  import roman
2  import unittest
3
4  class TestRomanNumerals(unittest.TestCase):
5
6      def testCreateAndGetVale(self):
7          thousand = roman.RomanNumeral("M")
8          self.assertEqual(thousand.value(), 1000)
9
10 if __name__ == "__main__":
11     unittest.main()
12
```

# Several Failures on the Path to Green

---

- module import fail: no file named roman.py → create one
- no class called RomanNumeral → create one
- wrong number of arguments for constructor → add self and value arguments
- no method called value() → create a “blank” one
- test now runs and reports failure!! → write simplest code to make it work
- test passes but contains duplication → add another test case to make it fail
  - end of step 2, onto step 3 directory
- original test passes, but new test fails → write simplest code to make it work
  - note, because of the tests, this is no longer trivial code to write

# Making Progress; But Long way to go

---

- We now have a class that can successfully handle Roman Numerals that consist only of “M” characters
  - And, we haven’t fully completed any of the items on our test list
  - We have lots of different directions we could go in
    - Add tests to check that we handle bad input
    - Add tests to add support for other roman numeral characters
    - Add tests to add basic support for addition or subtraction
    - etc.
  - Let’s focus on bad input to see the test-code-refactor loop one more time

# Test Case: Handle Bad Input

---

- Let's add test cases that handle (step 4 directory in sample code)
  - wrong input types (being handed a number or array rather than a string)
  - wrong values (producing a value that is outside the legal set of values)
- Then, we'll add a test case that can handle basic addition
- Note: all of the previous test cases remain
  - Once we get a test to pass, we never allow them to fail again

# Several Failures on the Path to Green (Again)

---

- add test case to handle non-string args to the constructor
  - Here we want to give it bad input and see if it raises an exception
    - All such tests will currently fail since the constructor just accepts whatever it is given
  - Start by passing a number, check to see if it raises an exception  $\Rightarrow$  fail
  - Add code to check for int  $\Rightarrow$  pass; now pass collection  $\Rightarrow$  fail
    - Make it pass but then erase code written so far and now write code to raise exception whenever a non-string is passed
      - This is the refactor step, as we were adding duplication based on the types of the parameters passed in between code and test case
- End of step 4; now make sure that we test the contents of the string
  - accept “M”, “MM”, and “MMM” for now, all else should fail

# Test Case: Handle Addition

---

- All we'll be able to do is handle  $1000 + 1000$  and  $1000 + 2000$ 
  - but this will ensure that we've got the basics in place
    - can handle correct additions
    - can flag additions that produce numbers outside the legal range
- Getting to Green
  - Add a sum method that follows the "value" pattern seen above
  - Generates ValueError if the value goes outside of the legal range
  - First a test case to handle an illegal addition
  - Then a test case to handle a legal addition
    - We'll encounter familiar steps
      - fails because there is no sum method
      - fails because it doesn't throw an exception
      - etc.



# End of Example

---

- Still a long way to go, but you should now have the feel of what test-driven development is like
  - Start with a system that needs a new feature
  - Write a test that documents what the expected results of the feature are
  - Add simplest code to make test pass
  - Make test more complicated, or add new test to reveal duplication
  - Once duplication is found, refactor to produce general code
  - Loop until feature is implemented and all tests pass

# Principles of TDD

---

- Testing List
  - keep a record of where you want to go;
    - Beck keeps two lists, one for his current coding session and one for “later”; You won’t necessarily finish everything in one go!
- Test First
  - Write tests before code, because you probably won’t do it after
  - Writing test cases gets you thinking about the design of your implementation;
    - does this code structure make sense?
    - what should the signature of this method be?

# Principles of TDD, continued

---

- Assert First
  - How do you write a test case?
    - By writing its assertions first!
  - Suppose you are writing a client/server system and you want to test an interaction between the server and the client
    - Suppose that for each transaction
      - some string has to have been read from the server, and
      - the socket used to talk to the server should be closed after the transaction
  - Lets write the test case

# Assert First

---

```
public void testCompleteTransaction {  
    ...  
    assertTrue(reader.isClosed());  
    assertEquals("abc", reply.contents());  
}
```

- Now write the code that will make these asserts possible

# Assert First, continued

---

```
public void testCompleteTransaction {  
    Server writer = Server(defaultPort(), "abc")  
    Socket reader = Socket("localhost", defaultPort());  
    Buffer reply = reader.contents();  
    assertTrue(reader.isClosed());  
    assertEquals("abc", reply.contents());  
}
```

- Now you have a test case that can drive development
  - if you don't like the interface above for server and socket, then write a different test case
  - or refactor the test case, after you get the above test to pass

# Principles of TDD, continued

---

- Evident Data
  - How do you represent the intent of your test data
  - Even in test cases, we'd like to avoid magic numbers; consider this rewrite of our second "times" test case

```
public void testMultiplication() {  
    Dollar five = new Dollar(5);  
    Dollar product = five.times(2);  
    assertEquals(5 * 2, product.amount);  
    product = five.times(3);  
    assertEquals(5 * 3, product.amount);  
}
```

- Replace the "magic numbers" with expressions

# Summary

---

- Test-Driven Design is a “mini” software development life cycle that helps to organize coding sessions and make them more productive
  - Write a failing test case
  - Make the simplest change to make it pass
  - Refactor to remove duplication
  - Repeat!

# Reflections

---

- Test-Driven Design builds on the practices of Agile Design Methods
  - If you decide to adopt it, not only do you “write code only to make failing tests pass” but you also get
    - an easy way to integrate refactoring into your daily coding practices
    - an easy way to introduce “integration testing/building your system every day” into your work environment
      - because you need to run all your tests to make sure that your new code didn’t break anything; this has the side effect of making refactoring safe
    - courage to try new things, such as unfamiliar design pattern, because now you have a safety net



# But how does it integrate with life cycles?

---

- With traditional software life cycles, TDD can be “test-driven development”
  - You’ll do requirements, use cases, class diagrams, etc. ➡ top down
  - Then TDD, coding from scratch to test your design ➡ bottom up
- With agile life cycles, TDD can be “test-driven design”
  - You create a new user story and use TDD to “discover” the classes that will help you implement that feature ➡ bottom up

# Testing Frameworks

---

- JUnit Tutorial: <<http://clarkware.com/articles/JUnitPrimer.html>>
- PyUnit: <<http://wiki.python.org/moin/PyUnit>>
- Unit testing in Objective-C and Xcode:
  - <<http://developer.apple.com/mac/articles/tools/unittestingwithxcode3.html>>
- Unit testing with C#: <<http://www.csunit.org/tutorials/tutorial7/>>
- Unit testing for Ruby:
  - <<http://www.ruby-doc.org/stdlib/libdoc/test/unit/rdoc/classes/Test/Unit.html>>

# Ruby on Rails

*Why should I care??*

*If your job is to create or manage web applications that capture and manipulate relational database from a web-based user interface, then Ruby on Rails may be the solution you've been looking for to make your web development life easier.*

*The result is an easy to use and cohesive framework that's rich in functionality.*

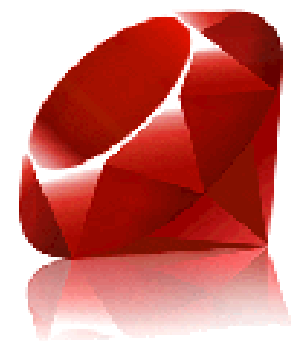
Ruby is a pure object-oriented scripted programming language with a super clean syntax that makes programming elegant and fun.

Rails is an open source Ruby framework for developing database-backed web applications.

Ruby on Rails provides

- High Productivity
- Reduced Development Time
- Easy to maintain and upgrade
- Built-in testing
- Friendly user community

Powerful  
&  
Beautiful  
Ruby  
Rails



Sahar Jambi, Spring 2011  
University of Colorado at Boulder

# What is PhoneGap?

presented by Lukas Jeter

PhoneGap is a framework that is based on the open standards of HTML5 and allows developers to use common web technologies (HTML, CSS, and JavaScript) to build applications for multiple mobile platforms from a single code base. This presentation will provide a brief overview of the features of PhoneGap, present the API's classes, show a sample of a core class, and give a short example of an HTML/JavaScript file that was compiled with the framework into an Android application.

# Processing Visualization Language

By Rutvi Joshi

- A **Graphical Sketch Book** and Environment, to graphically teach the fundamentals of computer science
- But it has evolved into something much bigger
- Being a **Free and Open Source** software tool, has made it accessible and thus very popular
- The Language is strictly typed and **very similar to Java** but it does not include the advanced features of Java, it integrates them, making it **easier to learn**
- Sketches run as **Java Applets**. Thus, they can be put in Web pages and can be **Exported** as software application on Windows, Linux and MAC
- **Processing.js** is a recent addition to this family which makes your application that you developed run using web standards without any plug ins!
- The presentation is a quick tutorial to familiarize yourself with the language and the various options that can be explored



# Electronic Nursing Record System: An Object-Oriented approach

- ✱ using Object Oriented techniques in ENRS:
  - ✱ Shows how to overcome the problems in the implementation phase by using of OOAD, UML, and RUP) in the implementation phase.
  - ✱ How it will be easy to exchange the information with third parties.
  - ✱ How to protect information internally and eternally by using encapsulation.
- ✱ The Topic shows:
  - ✱ Importance of create standard terminology in the implementation of an electronic nursing record system.
  - ✱ Nursing data are traditionally recorded in both structured and free-text formats, which make it difficult to implement an ICNP-based ENRS.

# OpenSceneGraph

-Ryan Kroiss

- High performance 3D graphics toolkit
- OO framework on top of OpenGL
- Open source, cross platform
- Extensive use of design patterns
- Scalable, portable, extensible
- Commonly used for games, flight simulators, etc.
- Based on concept of scene graph
- Nodes in graph represent aspects of scene
  - Leaf nodes - rendered geometry
  - Non-leaf nodes - establish hierarchy and scene manipulations
- Node kits add extensions
- Huge variety of data loaders





# Executive Summary



- Processing is a free, open-source, Java-based framework and IDE that focuses on usability, creativity, and extensibility
- Processing has a large base of included and contributed libraries that extend its functionality into video, audio, hardware communication, file i/o, math and physics simulation, advanced 3D, and more
- Processing supports all Java-like object oriented definitions and techniques, as well as allowing use of standard and custom Java libraries within processing itself
- Processing can also be used in Eclipse using the Proclipsing plugin or by importing the Core Processing libraries into Eclipse

by Ben Leduc-Mills



# Topics Discussed

---

- History of Struts
- Basic features of struts 2.0
- Struts2 vs struts1.1
- Architecture of struts2.0
- MVC 2 Model Architecture and Overview
- Basic flow of struts2.0
- Core Components
- Pros and Cons



# Keynote: Domain Driven Design

- The Participants
- What is Domain Driven Design?
- Modeling a domain
- Ingredients of Effective Modeling
- Knowledge crunching & Knowledge-Rich Design
- Fractured Language Problem
- Solution – Ubiquitous language
- Confluence of Terminologies
- Fine tune a model using U Language
- Isolating the domain (Layered Architecture)
- Layer Policies
- Communication with higher layers
- Model expressed in software
- Entities
- Designing Entities
- Value Object
- Services
- Modules
- Non-Objects in an Object World
- Life Cycle of a Domain object
- Aggregates
- Factories
- Where the Factory should be positioned?
- Factories vs. Constructors
- Repositories
- Advantages of Repository
- Putting it all together :The Cargo tracking example

by Vivek Madhavan

# Hadoop MapReduce: Summary and Conclusion

- Hadoop MapReduce is a large scale, open source software framework dedicated to scalable, distributed, data-intensive computing
- The framework breaks up large data into smaller parallelizable chunks and handles scheduling
  - Maps each piece to an intermediate value
  - Reduces intermediate values to a solution
  - User-specified partition and combiner options
- Fault tolerant, reliable, and supports thousands of nodes and petabytes of data
- If you can rewrite algorithms into Maps and Reduces, and your problem can be broken up into small pieces solvable in parallel, then Hadoop's MapReduce is the way to go for a distributed problem solving approach to large datasets
- Tried and tested in production
- Many implementation options

by Casey McTaggart



# Summary

- ✦ JPA is a specification that a developer can code to in order to easily leverage ORM technologies
- ✦ There are a wide variety of vendors that implement the specification
  - ✦ Coding to the spec allows the developer to be flexible in their choice of vendor implementations with limited ripple throughout the codebase
- ✦ JPA greatly simplifies persistence of Objects through a small set of easily utilized annotations

**JPA by Aaron Schram**



# Coming Up Next

---

- Lecture 29: ORM and Hibernate
- Lecture 30: Dependency Injection and Spring; Semester Wrap Up