Refactoring

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/5448 — Lecture 27 — 04/19/11

Credit where Credit is Due

- Some of the material for this lecture is taken from “Refactoring: Improving the Design of Existing Code” by Martin Fowler; as such, some material is copyright © Addison Wesley, 1999

Goals for this lecture

- Introduce the concept of Refactoring and cover a few examples

- Present more executive summaries of graduate student presentations

What is Refactoring

- Refactoring is the process of changing a software system such that
  - the external behavior of the system does not change
    - e.g. functional requirements are maintained
  - but the internal structure of the system is improved
- This is sometimes called
  - “Improving the design after it has been written”

(Very) Simple Example

- Consolidate Duplicate Conditional Fragments (page 243); This

```
if (isSpecialDeal()) {
    total = price * 0.95;
    send()
} else {
    total = price * 0.98;
    send()
}
```

- becomes this

```
if (isSpecialDeal()) {
    total = price * 0.95;
} else {
    total = price * 0.98;
}
send();
```

(Another) Simple Example

- Replace Magic Number with Symbolic Constant

```
double potentialEnergy(double mass, double height) {
        return mass * 9.81 * height;
}
```

- becomes this

```
double potentialEnergy(double mass, double height) {
        return mass * GRAVITATIONAL_CONSTANT * height;
}
static final double GRAVITATIONAL_CONSTANT = 9.81;
```

In this way, refactoring formalizes good programming practices

Refactoring is thus Dangerous!

- Manager's point-of-view
  - If my programmers spend time "cleaning up the code" then that's less time implementing required functionality (and my schedule is slipping as it is!)
- To address this concern
  - Refactoring needs to be **systematic**, **incremental**, and **safe**

Refactoring is Useful Too

- The idea behind refactoring is to acknowledge that it will be difficult to get a design right the first time and, as a program's requirements change, the design may need to change
    - refactoring provides techniques for evolving the design in small incremental steps
- Benefits
    - Often code size is reduced after a refactoring
    - Confusing structures are transformed into simpler structures
        - which are easier to maintain and understand

A “cookbook” can be useful

- Refactoring: Improving the Design of Existing Code
    - by Martin Fowler (and Kent Beck, John Brant, William Opdyke, and Don Roberts)
- Similar to the Gang of Four’s Design Patterns
    - Provides “refactoring patterns”

Principles in Refactoring

- Fowler's definition
  - Refactoring (noun)
    - a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior
  - Refactoring (verb)
    - to restructure software by applying a series of refactorings without changing its observable behavior

Principles, continued

- The purpose of refactoring is
    - to make software easier to understand and modify
- contrast this with performance optimization
    - again functionality is not changed, only internal structure;
    - however performance optimizations often involve making code harder to understand (but faster!)

Principles, continued

- When you systematically apply refactoring, you wear two hats
  - adding function
    - **functionality is added** to the system **without** spending any time **cleaning the code**
  - refactoring
    - **no functionality is added**, but the code is **cleaned up**, made easier to understand and modify, and sometimes is reduced in size

Principles, continued

- How do you make refactoring safe?
  - First, use refactoring “patterns”
    - Fowler’s book assigns “names” to refactorings in the same way that the GoF’s book assigned names to patterns
  - Second, test constantly!
    - This ties into the agile design paradigm
      - you write tests **before** you write code
      - after you refactor, you run the tests and check that they all pass
        - if a test fails, the refactoring broke something **but you know about it right away** and can fix the problem before you move on

Why should you refactor?

- Refactoring improves the design of software
  - without refactoring, a design will “decay” as people make changes to a software system
- Refactoring makes software easier to understand
  - because structure is improved, duplicated code is eliminated, etc.
- Refactoring helps you find bugs
  - Refactoring promotes a deep understanding of the code at hand, and this understanding aids the programmer in finding bugs and anticipating potential bugs
- Refactoring helps you program faster
  - because a good design enables progress

When should you refactor?

- The Rule of Three
    - Three “strikes” and you refactor
        - refers to duplication of code
- Refactor when you add functionality
    - do it before you add the new function to make it easier to add the function
    - or do it after to clean up the code after the function is added
- Refactor when you need to fix a bug
- Refactor as you do a code review

Problems with Refactoring

- Databases
  - Business applications are often tightly coupled to underlying databases
    - code is easy to change; databases are not
  - Changing Interfaces (!!)
    - Some refactorings **require that interfaces be changed**
      - if you own all the calling code, no problem
      - if not, the interface is “published” and can’t change
- Major design changes **cannot** be accomplished via refactoring
  - This is why agile design says that software devs. need courage!

Refactoring: Where to Start?

- How do you identify code that needs to be refactored?
  - Fowler uses an olfactory analogy (attributed to Kent Beck)
  - Look for “Bad Smells” in Code
    - A very valuable chapter in Fowler’s book
    - It presents examples of “bad smells” and then suggests refactoring techniques to apply

Bad Smells in Code

- **Duplicated Code**
  - bad because if you modify one instance of duplicated code but not the others, you (may) have introduced a bug!
- **Long Method**
  - long methods are more difficult to understand
    - performance concerns with respect to lots of short methods are largely obsolete

Bad Smells in Code

- **Large Class**
  - Large classes try to do too much, which reduces cohesion
- **Long Parameter List**
  - hard to understand, can become inconsistent if the same parameter chain is being passed from method to method
- **Divergent Change**
  - symptom: one type of change requires changing one subset of methods; another type of change requires changing another subset
  - Related to cohesion

Bad Smells in Code

- **Shotgun Surgery**
  - a change requires lots of little changes in a lot of different classes
- **Feature Envy**
  - A method requires lots of information from some other class
    - move it closer!
- **Data Clumps**
  - attributes that clump together (are used together) but are not part of the same class

Bad Smells in Code

- **Primitive Obsession**
  - characterized by a reluctance to use classes instead of primitive data types
- **Switch Statements**
  - Switch statements are often duplicated in code; they can typically be replaced by use of polymorphism (let OO do your selection for you!)
- **Parallel Inheritance Hierarchies**
  - Similar to Shotgun Surgery; each time I add a subclass to one hierarchy, I need to do it for all related hierarchies
    - Note: some design patterns encourage the creation of parallel inheritance hierarchies (so they are not always bad!)

Bad Smells in Code

- **Lazy Class**
  - A class that no longer “pays its way”
    - e.g. may be a class that was downsized by a previous refactoring, or represented planned functionality that did not pan out
- **Speculative Generality**
  - “Oh I think we need the ability to do this kind of thing someday”
- **Temporary Field**
  - An attribute of an object is only set/used in certain circumstances;
    - but an object should need all of its attributes

Bad Smells in Code

- **Message Chains**
  - a client asks an object for another object and then asks that object for another object etc. Bad because client depends on the structure of the navigation
- **Middle Man**
  - If a class is delegating more than half of its responsibilities to another class, do you really need it? Involves trade-offs, some design patterns encourage this (e.g. Decorator)
- **Inappropriate Intimacy**
  - Pairs of classes that know too much about each other's implementation details (loss of encapsulation; change one class, the other has to change)

Bad Smells in Code

- **Data Class (information holder)**
  - These are classes that have fields, getting and setting methods for the fields, and nothing else; they are data holders, but objects should be about data AND behavior
- **Refused Bequest**
  - A subclass ignores most of the functionality provided by its superclass
  - Subclass may not pass the “IS-A” test
- **Comments** (!)
  - Comments are sometimes used to hide bad code
    - “…comments often are used as a deodorant” (!)

The Catalog

- The refactoring book has 72 refactoring patterns!
  - I’m only going to cover a few of the more common ones, including
    - Extract Method
    - Replace Temp with Query
    - Move Method
    - Replace Conditional with Polymorphism
    - Introduce Null Object
    - Separate Query for Modifier
    - Introduce Parameter Object
    - Encapsulate Collection

Extract Method

- You have a code fragment that can be grouped together
- Turn the fragment into a method whose name explains the purpose of the fragment
- Example, next slide

Extract Method, continued

```
void printOwing(double amount) {
    printBanner()
    //print details
    System.out.println(“name: ” + _name);
    System.out.println(“amount: ” + amount);
}
```

```
==========================================
```

```
void printOwing(double amount) {
    printBanner()
    printDetails(amount)
}

void printDetails(double amount) {
    System.out.println(“name: ” + _name);
    System.out.println(“amount: ” + amount);
}
```

Replace Temp with Query

- You are using a temporary variable to hold the result of an expression
  - Extract the expression into a method;
  - Replace all references to the temp with the expression.
  - The new method can then be used in other methods

- Example, next slide

Replace Temp with Query, continued

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.98;

================================

if (basePrice() > 1000)
    return basePrice() * 0.95;
else
    return basePrice() * 0.98;
…
double basePrice() {
    return _quantity * _itemPrice;
}
```

Move Method (I)

- A method is using more features (attributes and operations) of another class than the class on which it is defined
    - Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether

Move Method (II)

```
 1 class Account {
 2     ...
 3     double overdraftCharge() {
 4         if (_type.isPremium()) {
 5             double result = 10;
 6             if (_daysOverdrawn > 7 ) {
 7                 result += (_daysOverdrawn - 7) * 0.85;
 8             }
 9             return result;
10         } else {
11             return _daysOverdrawn * 1.75;
12         }
13     }
14
15     double bankCharge() {
16         double result = 4.5;
17         if (_daysOverdrawn > 0) {
18             result += overdraftCharge();
19         }
20         return result;
21     }
22
23     private AccountType _type;
24     private int _daysOverdrawn;
25 }
26
```

A class to manage a bank account. There are currently two types of accounts: standard and premium.

Currently we have only two types of accounts, **standard** and **premium**, but it is anticipated that we will be adding new account types and that each type will have a different rule for calculating an overdraft charge.

As such, we'd like to **move the method** overdraftCharge() **to** the AccountType class.

Move Method (III)

- When moving a method to a new class, we examine its code to see if makes use of internal attributes of its original class
  - In this case, `overdraftCharge()` makes use of `_daysOverdrawn`
- All such attributes become parameters to the method in its new home. (If the method already had parameters, the new parameters get tacked on to the end of its existing parameter list.)
  - In this case, `_daysOverdrawn` will stay in the `Account` class and be passed as a parameter to `AccountType.overdraftCharge()`.
- Note, also, that since we are moving this method to the `AccountType` class, all calls to its methods that previously required a variable reference, can now be made directly
  - Thus, `_type.isPremium()` becomes simply `isPremium()` in the method's new home

Move Method (IV)

```
1  class AccountType {
2      ...
3      double overdraftCharge(int daysOverdrawn) {
4          if (isPremium()) {
5              double result = 10;
6              if (daysOverdrawn > 7 ) {
7                  result += (daysOverdrawn - 7) * 0.85;
8              }
9              return result;
10         } else {
11             return daysOverdrawn * 1.75;
12         }
13     }
14     ...
15 }
16
```

Here is the method in its new home. It has a daysOverdrawn parameter, which is used instead of _daysOverdrawn, throughout the method. _type.isPremium() is now just isPremium(), as advertised.

Move Method (V)

```
1  class Account {
2      ...
3      double overdraftCharge() {
4          return _type.overdraftCharge(_daysOverdrawn);
5      }
6
7      double bankCharge() {
8          double result = 4.5;
9          if (_daysOverdrawn > 0) {
10             result += overdraftCharge();
11         }
12         return result;
13     }
14
15     private AccountType _type;
16     private int _daysOverdrawn;
17 }
18
```

Back in the Account class, we update overdraftCharge() to delegate to the overdraftCharge() method in the AccountType class. Or, we could…

Move Method (VI)

```
1  class Account {
2      ...
3      double bankCharge() {
4          double result = 4.5;
5          if (_daysOverdrawn > 0) {
6              result += _type.overdraftCharge(_daysOverdrawn);
7          }
8          return result;
9      }
10
11     private AccountType _type;
12     private int _daysOverdrawn;
13 }
14
```

… get rid of the overdraftCharge() method in Account entirely. In that case, we move the call to AccountType.overdraftCharge() to bankCharge()

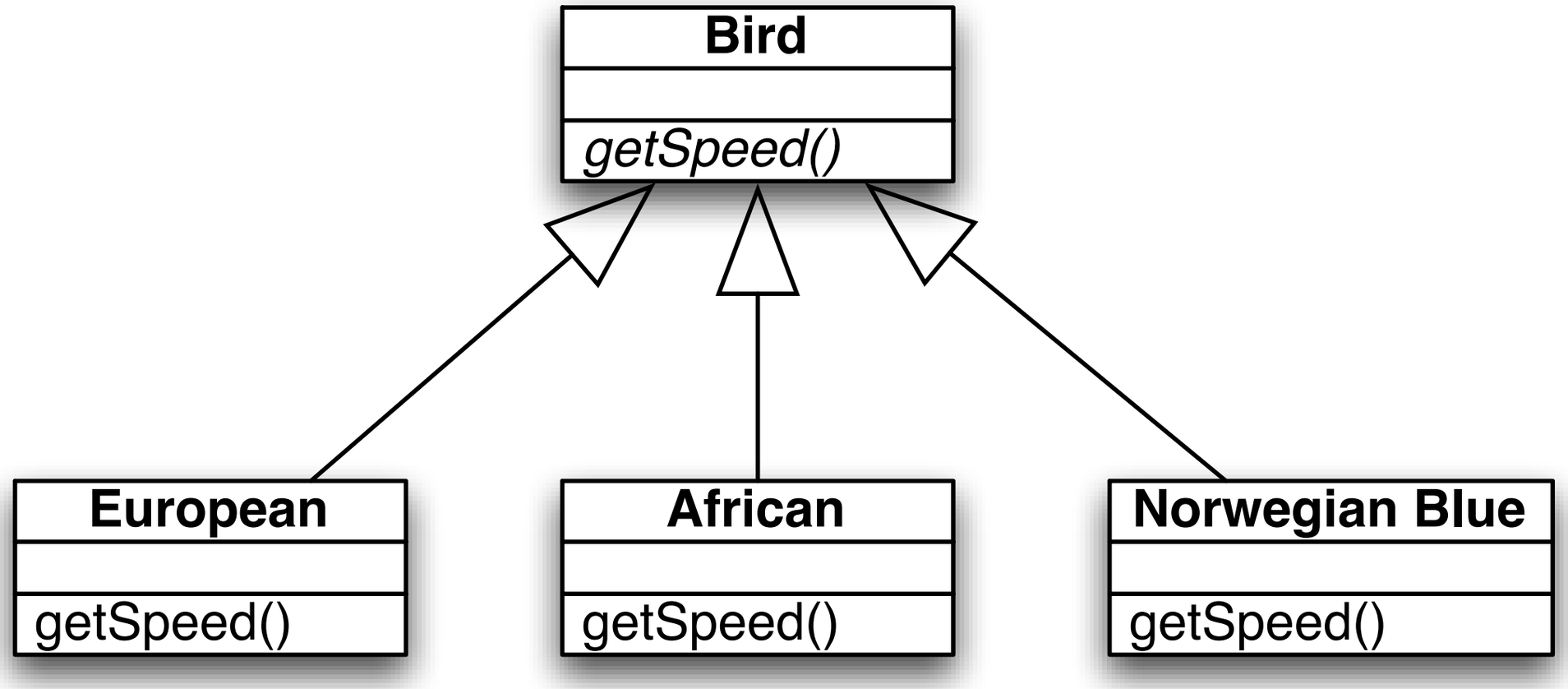
Replace Conditional with Polymorphism (I)

- You have a conditional that chooses different behavior depending on the type of an object
    - Move each “leg” of the conditional to an overriding method in a subclass. Make the original method abstract

Replace Conditional with Polymorphism (II)

```
double getSpeed() {
    switch (_type) {
        case EUROPEAN:
            return getBaseSpeed();
        case AFRICAN:
            return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
        case NORWEGIAN_BLUE:
            return (_isNailed) ? 0 : getBaseSpeed(_voltage);
    }
    throw new RuntimeException("Unknown Type of Bird")
}
```

Replace Conditional with Polymorphism (III)



With this configuration, you can now write code that looks like this:

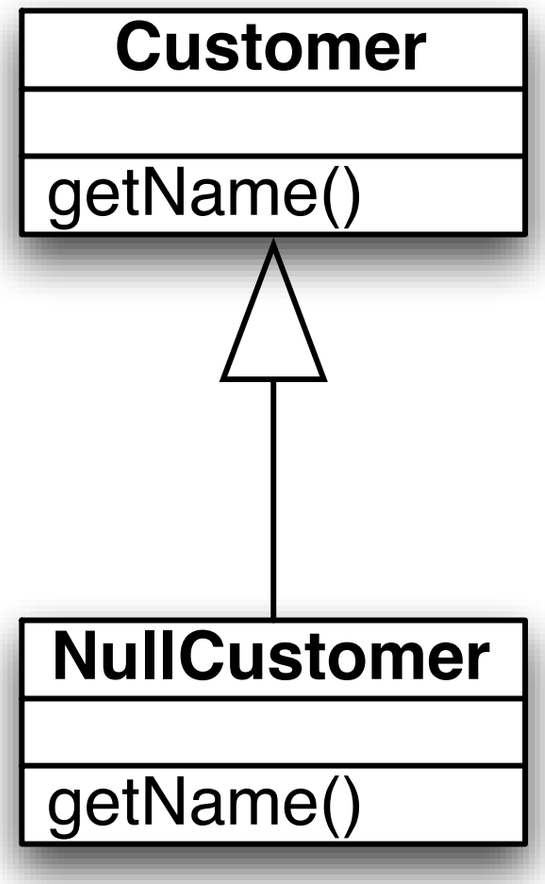
```
void printSpeed(Bird[] birds) {
    for (int i=0; i < birds.length; i++) {
        System.out.println("" + birds[i].getSpeed());
    }
}
```

and everything will work correctly via polymorphism and will be easy to extend: just add a new subclass to support a new type of bird

Introduce Null Object (I)

- Repeated checks for a null value (see below)
- Rather than returning a null value from findCustomer() return an instance of a “null customer” object

```
...
Customer c = findCustomer(...);
...
if (customer == null) {
    name = “occupant”
} else {
    name = customer.getName()
}
if (customer == null) {
...
```



Introduce Null Object (II)

```
public class NullCustomer {
  public String getName() { return “occupant”;}
}
==============================
Customer c = findCustomer(...);
name = c.getName();
```

- The conditional goes away entirely!!
- In Fowler’s book, this technique is presented as a refactoring; in other contexts, its presented as a design pattern
  - Either way, this technique is very useful!

Separate Query for Modifier

- Sometimes you will encounter code that does something like this
  - getTotalOutstandingAndSetReadyForSummaries()
- It is a query method but it is also changing the state of the object being called
  - This change is known as a “side effect” because it’s not the primary purpose of the method
- It is generally accepted practice that queries should not have side effects so this refactoring says to split methods like this into:
  - getTotalOutstanding()
  - setReadyForSummaries()
- Try as best as possible to avoid side effects in query methods

Introduce Parameter Object

- You have a group of parameters that go naturally together
    - Stick them in an object and pass the object
- Imagine methods like
    - amountInvoicedIn(start: Date; end: Date);
    - amountOverdueIn(start: Date; end: Date);
- This refactoring says replace them with something like
    - amountInvoicedIn(dateRange: DateRange)
- The new class starts out as a data holder but will likely attract methods to it

Encapsulate Collection

- A method returns a collection
  - Make it return a read-only version of the collection and provide add/remove methods
- Student class with
  - getCourses(): Map;
  - setCourses(courses: Map);
- Change to
  - getCourses(): ReadOnlyList
  - addCourse(c : Course)
  - removeCourse(c : Course)

Changing the externally visible collection, too, is a good idea to protect clients from depending on the internals of the Student class

Refactoring Example in Dive into Python 3

- Mark Pilgrim has created an excellent introduction to Python 3 at
  - <http://diveintopython3.org/>
- In one chapter, he discusses refactoring
  - <http://diveintopython3.org/refactoring.html>
- However, he doesn't use refactoring patterns and he changes the previous system all in one step;
- He can do this because he wrote test cases for the original system in this chapter:
  - <http://diveintopython3.org/unit-testing.html>
- Highly recommended!

Wrapping Up

- Refactoring is a useful technique for making non-functional changes to a software system that result in
  - better code structures
    - Example: There's a book out there called "Refactoring to Patterns"
  - less code
    - Many refactorings are triggered via the discovery of duplicated code
      - The refactorings then show you how to eliminate the duplication
- Bad Smells
  - Useful analogy for discovering places in a system "ripe" for refactoring

SUMMARY

- The Java Concurrency Framework provides a set of safe and robust services that allow Java programmers to easily create code that will be able to take advantage of concurrent programming
- This presentation will introduce the framework and show examples of how the framework can be used in multithreaded programs

by Jay Daugherty

Executive Summary ActionScript 3.0

- This presentation assumes the audience has the basic understanding of object-oriented programming and has some exposure to other OOP languages.

- This presentation will briefly review ActionScript 3.0 language and syntax features and will highlight new features that are not common in other similar OOP languages.

- This presentation should be viewed because not only is it simple and understandable, it directs the audience to two really intricate and expansive ActionScript based web projects.

ACTIONSCRIPT 3.0

- An object oriented programming language developed by Adobe Systems
- Used in applications and websites targeting the Adobe Flash Player
- Features of ActionScript 3.0:
  - Core Language Features (defines fundamentals of AS3 language)
  - Flash API Features (AS3 provides a new set of APIs)
- Syntactically similar to JavaScript, can be run in any Flash Authoring environment
- Like any other programming language, provides basic programming fundamentals such as variables and constants, data types, operators, conditional and looping statements, etc.
- Provides better support, than the legacy ActionScript versions, for OO principles such as inheritance, polymorphism, encapsulation and interfaces
- ActionScript 3.0 classes comprise of three characteristics:
  - Properties
  - Methods
  - Events
- For every class, ActionScript creates a special class object that allows sharing of both behavior and state
- ActionScript provides many top-level classes for applications using movie clips, date and time, user interactions, strings, arrays, XML, etc.
- This core classes ease the process of developing an application up to an great extent
- With ActionScript 3.0, developers can achieve excellent productivity and performance with content and applications that target Flash Player
- Because of the power and flexibility it provides, it is ideally suited for building rich internet applications, which have become an essential part of the web experience

Presentation Overview:

- Next 20-23 slides focuses on Android 3.0 framework for experienced developers
  - Platform technologies
  - New user features (with screenshots)
  - New Developer features (with screenshots)
  - API differences from the previous versions

by Saurabh Goel
Bonus Points: Executive summary had 10 slides; I picked this one to include. ☺

Need of the hour:

OH GOD!! How to make these guys talk

I know only DB and file protocols. I know how to manage all Network Elements

Network Management System

Hey! No issues. I can translate the DB, file protocols with the SNMP, CORBA protocols

Middleware Layer

I know only SNMP protocol

I know only CORBA protocol

Network Element (Cisco)

Network Element (Ericsson)

Executive Summary

- This is a beginners introduction to the java concurrency framework
- Some familiarity with concurrent programs is assumed
  - However the presentation does go through a quick background on concurrency
  - So readers unfamiliar with concurrent programming should still get something out of this
- The structure of the presentation is as follows
  - A brief history into concurrent programming is provided
  - Issues in concurrent programming are explored
  - The framework structure with all its major components are covered in some detail
  - Examples are provided for each major section to reinforce some of the major ideas

by Sidartha Gracias

Qt (“cute” or QueTee)

- Qt is a cross-platform application framework that is widely used to develop GUI applications.
- In runs on many OS platforms, including Windows, Linux, Solaris, FreeBSD, MeeGo, and Mac OSX, with a “write-once, compile anywhere” approach.
- Many well known applications are written using Qt, including Google Earth, Autodesk, KDE, Skype, VLC Media Player, Virtual Box, and Mathematica.
- This presentation covers the major abstractions and underlying features of the framework. A screencast is included which demonstrates building a simple demo application.

by Mark Grebe

Amit Gupta
amit.gupta@colorado.edu
CSCI-5448 Spring 2011

The Haiku Operating System

- We'll be taking a closer look at the HAIKU Operating System
- Its an open source, object oriented operating system, written in C++
- We'll learn about its history and origins from BeOS
- We'll overview its architecture & various subsystems
- We'll examine selected classes of some important subsystems
- We'll see interesting examples of powerful features that emerge out of its object oriented design
- We'll look at some interesting features of its file system
- Finally, we'll get a sense of the current state of this project

Symfony Framework

-Hemalatha Gurumoorthy

- This Presentation gives an introduction to symphony. Describes the architecture used in symphony.
- I have tried to give a basic understanding of framework, for my listener to be able to take if further from there.
- Symfony is a powerful tool which has excellent documentation and is open source.
- It provides good maintenance and extension support and is the best option to develop a complex application
- It has very well defined inbuild functions and is easily adaptible for complex class definitions.

Executive Summary

- The "C#" presentation gives a brief overview about the language. A handful of features that have been introduced to C# which have amazed the developers of the object-oriented world are covered with short & easy explanations and code examples. These features are considered important and helpful for developers to write easy, maintainable, short, and readable code. Therefore, these few slides will give you a head start in C#, and will provide some basic, but essential concepts to familiarize you about the language.

by Mazin Hakeem

One-page Domain-Driven Design overview

- DDD is the idea of a very tight coupling between a model of the domain, such as an activity diagram or use case, and the software.
- This connection is a very important concept that can help students succeed in their careers – I applied DDD principles during my career and received accelerated promotions, better raises, and many monetary awards.
- My presentation reinforces OO principles and design patterns that Professor Anderson has discussed over the course of the semester.

by Bill Irelan

Coming Up Next

- Lecture 28: Test-Driven Design