2007 Concurrent Programming
Part I — Java Programming in the Large
Introductory Notes and Reading List

Dr. Graham Roberts
room 6.17, phone 33711
email: G.Roberts@cs.ucl.ac.uk
URL: http://www.cs.ucl.ac.uk/staff/G.Roberts/2007/index.hml

1. Course Overview

2007 has two parts. Part I, taught by myself, continues on from 1008 in the 1st year and looks at writing and testing larger Java programs. Part II, taught by Dr. Kevin Bryson, will move on to look at threads and concurrent programming. These notes mostly focus on Part I of the course, highlighting some key issues you need to think and learn about and the reading list.

First, however, some details about the course organisation.

1.1 Course Organisation

Part I will consist of 10 lectures on general Java programming, while Part II has 20 lectures on concurrent programming using Java. The lectures will be used to provide context and background, as well as introducing the key ideas and concepts needed to do the coursework. Exercises and coursework will provide lots of programming practice. Background reading is important. A key feature of 2007 is that you get to do a lot of programming!

1.2 Assessment and Coursework

The course will be assessed by 15% coursework and 85% exam. At least 35% of the coursework must be completed and submitted (regardless of what marks may be given). Otherwise, a student will be deemed to have not completed enough of the coursework to be passed, and be marked as Not Complete (NC).

The exam will be 2.5 hours in length. There is one past exam paper available for 2007, from last year (see website). Some questions from past papers for the course COMP2B11, which preceded 2007, are also relevant.

The details of the coursework will be announced as the course progresses. However, you can expect two mini-projects requiring you to write some larger programs. It is up to you to do the programming and to do the reading/research to find out how to write the programs. Some of the material will be covered in the lectures but not all.

As in the first year, the mini-projects will be graded and should be submitted both on paper and electronically. Printed work should include a signed coversheet (available from the main Students page on the CS website) and be handed in to the departmental office before the advertised deadline.

You should submit *properly presented* source code for each mini-project. By properly presented I mean that the code should be readable and understandable by other people. This is more than just getting the indentation right; variable and method names should be carefully chosen, methods should be structured to clearly show their intent and the whole should clearly

communicate the purpose of the code. Comments should be included as necessary to highlight design and structure, and convey design information that may not be obvious from looking at the source code. User manuals, systems manuals and other documentation are *not* needed but the source code *must* be properly presented. Testing is essential. Marking will pay attention to the proper use of design and presentation idioms.

Plagiarism (copying other people's work) will be treated very seriously. Anybody found to have copied work and submitted it in their own name will be subject be reported to the Chair of the Examination Board.

Note that electronically submitted work will be automatically scanned for plagiarism — this is highly effective and plagiarism will be detected if it has occurred.

Remember, your ability at doing the coursework is a good indicator as to whether you will pass the exam. If you do a poor job on the coursework due to laziness you will find the exam very hard.

1.3 Lab Classes

There are timetabled lab classes during which demonstrators will be available to provide help and feedback on programming coursework. See the online timetables for times/places.

1.4 Java Version and Tools

The course will be using the Java 2 Platform, Standard Edition (J2SE) version 5.0 (visit http://java.sun.com for the latest information and downloads). This version includes significant enhancements over v1.4 used for courses last year.

We will also be using TestNG 2.5.5 (a unit testing framework), the Ant build tool v. 1.6, Subversion 1.2 and the Eclipse IDE v3.1. JEdit, the Tomcat web server and Together Control Center are available on lab machines.

Learning how to use the tools, especially TestNG, Eclipse, Subversion and Ant, will be your responsibility. You must spend some time reading the documentation and, more importantly, using the tools to become familiar with them.

The software for Windows, Mac OS X and Linux machines will be available on CD from the departmental office (not from me!), or can be downloaded directly if you have a suitable internet connection.

1.5 Web Pages

The course will have its own set of pages on the department's teaching web. My pages are at:

http://www.cs.ucl.ac.uk/staff/G.Roberts/2007/index.html

These notes and other support and manual material will be put into the course web pages during the course of the term.

There is also a page for Java support within the department at:

http://www.cs.ucl.ac.uk/teaching/java/index.html

This has links to the on-line Java documentation — the Javadoc. You should know what this contains as you will need to refer to it frequently. There is a great deal of Java information available on the web, so it is worth spending a bit of time browsing around to find the most useful sites. Email me any URL's which you feel should be added to the local pages for easy reference.

2. Java Programming

The main aims of Part I of the course are:

- To provide more programming practice and learn more about Java.
- Looking at how Graphical User Interfaces (GUIs) are implemented.
- To gain insight into writing larger sized programs (500+ lines of code).
- Get an overview of Java project organisation, using the Ant build tool and using

Subversion for version control.

- To introduce test-driven development (TDD) and refactoring.

Part I of this course follows on from 1007/1008, which were about 'Programming in the Small' — learning to program in the first place and then writing small programs only a few tens or hundreds of lines long. In contrast to the 1st year courses, 2007 is about 'Programming in the Large', which is an appropriate subtitle for the course. Larger programs are capable of doing interesting and useful things but need significant design effort, and new ways of managing their development. Larger programs are also often written by a group of people rather than a single person.

Beyond programming in the large, there is "Programming in the Huge", for want of a better name[1]. Much commercial software (such as that produced by companies like Microsoft) falls into this category. Programs will contain millions of lines of code and require a very great deal of design and implementation effort. This scale of development is beyond the scope of this course but the ideas presented do scale up, becoming even more important.

The content of 2007 is related to that of 2009 Software Engineering and Human Computer Engineering, and you will find much of what you learn in 2009 to be directly of use in 2007. In particular, 2009 will cover the Unified Modelling Language (UML) in detail.

2.1 Serving your Apprenticeship

During your first year you learnt about the Java programming language and practised using it via a series of exercises and courseworks. This course will allow you to continue to practice and develop your programming skills, and also to start writing larger and more interesting programs. In particular, it will introduce you to concurrency and concurrent programming.

Why more programming? Programming is primarily a skill that has to be learnt via a long apprenticeship. You have made the first steps and now is the time to continue developing your skills. Programming is much more than just knowing about a particular programming language, it is also about knowing how to design and structure code, so that it works properly, is maintainable and, most importantly, so that its structure is appropriate to the application you are trying to build. As you write larger programs these issues will become more apparent and important.

Programming is also about knowing how and when to use programming tools. Tools include language compilers, editors, debuggers, build managers, version control, manuals and code libraries. You need to get to know what all the tools are, what they can do and when they should be used.

2.2 Test-Driven Development

Testing of an application is traditionally done as a separate activity after coding. There is overwhelming evidence that this is the wrong approach, greatly increases the cost of software development and results in buggy, unreliable code. Moreover, as time spent designing and coding often exceeds estimates, testing is frequently either cut back or not done at all.

An important element of 2007 will be a close look at test-first programming, or test-driven development (TDD) as it is increasingly being known. With this approach, test code is written *before* the code to be tested. This ensures that any application code written must be tested and will be written in such a way that it can be tested. Testing then drives the design process.

As code is developed, a corresponding test suite — a collection of tests — will also be developed. This can be repeatedly run to check that the application code works (called regression testing), and that modifications and new additions don't break existing code. The philosophy is test everything repeatedly, find errors as soon as possible and fix them immediately.

1. Actually a good name is mega-programming!

Testing is most effective if tests can be run automatically, so we will be making use of a tool called TestNG (see http//testng.org/doc/index.html). This provides a framework for writing and running test code, making it straightforward to develop and run test suites.

2.3 Refactoring

An important part of the test-first approach is to keep code as simple and clean as possible and use *refactoring* to systematically transform it as new features are added. With this approach simplicity is favoured over more elaborate design, but the simplicity allows the code to be rapidly modified to incorporate more functionality and more advanced design structures when they come to be needed. Note, however, that simplicity is a relative term and doesn't imply that your code is trivial, lacks good design or is in any way sub-standard. Creating clean and simple code is a highly skilled practice.

Refactoring is based on a collection of simple rules describing strategies for modifying code when changes are needed. One of the course texts (see later for details) explains refactoring in detail and is essential reading — you *must* read it as the material will not be covered in any detail during the lectures.

The combination of test-first programming and refactoring leads to a highly reliable and robust design and coding process, and is a core element of the eXtreme Programming philosophy. Both test-first and refactoring require skill and attention to detail — they are not an easy way out or in any way an excuse for sloppy work. It will be important to put time and effort into understanding how they work and practising using them.

3. Course Texts

You need to read the course texts during the course of the academic year, preferably during term 1. The recommended text for getting a good overview of software development with Java is:

Software Design Using Java 2, by Kevin Lano et al, published by Palgrave MacMillan, 2002, 1-4039-0230-5

The book on refactoring is:

Refactoring - Improving the Design of Existing Code, by Martin Fowler et al., published by Addison Wesley, 1999, 0-201-48567-2

You must read this book — the material will not be covered in any detail during lectures and is assumed to be a core part of the course. This is an excellent book and should be on your essential reading list regardless of whether or not it is the course text!

The book on test-driven development is:

test-driven development, A Practical Guide, by David Astels, published by Prentice Hall/PTR, 2003, 0-13-101649-0

Again you must read this book — it covers many aspects of test-driven development including developing GUIs and has a complete example showing how a working application is developed.

The course will continue to make use of the 1007/8 book:

Developing Java Software 2nd Edition, by Russel Winder and Graham Roberts, published by John Wiley & Sons, 2000, 0-471-60696-0

I would also very strongly recommend that you read these as well:

Agile Software Development, by Robert C. Martin, published by Prentice Hall/PTR, 2002, ISBN 0-13-59744-5

Effective Java, by Josh Bloch, published by Addison Wesley, 2001, 0201310058

and

UML for Java Programmers, by Robert C. Martin, published by Prentice Hall/PTR, 2003, 0-13-1428489

(excellent book if you want to really understand how to use UML).

3.1 Background Reading List

The following books are all worth reading if you get the chance (a good programmer should aim to read 6-10 such books a year if he or she expects to develop their expertise and keep up to date).

Your ability to pass the course will be significantly enhanced if you make good use of the reading material available.

Unit Testing Books

Unit Testing in Java, by Johannes Link, published by Morgan Kaufmann, 2003, 1-55860-868-0

JUnit in Action, by Vincent Massol, published by Manning Press, 2003, 1930110995

Pragmatic Unit Testing in Java with JUnit, by Andy Hunt and Dave Thomas, published by The Pragmatic Programmers, 2003, 0974514012

JUnit Recipes, by J B Rainsberger, published by Manning Press, 2004, 1932394230

Agile Java - Crafting Code with Test-Driven Development, by Jeff Langr, published by Prentice Hall, 2005, 0-13-148239-4

Java Books

Core Java Foundation Classes, 2nd ed., by Kim Topley, published by Prentice Hall PTR, 2001, 013090581X

Graphic Java vol.2 Mastering The JFC, 3rd Edition, by David Geary, published by Prentice Hall PTR, 1999, 0-13-079667-0

Swing, 2nd edition, by Matthew Robinson & Pavel Vorobiev, published by Manning, 2003, 1-930110-88-X

The Java Programming Language 3rd Edition, by Ken Arnold, James Gosling and David Holmes, published by Addison Wesley, 2000, 0201704331

The Java Tutorial Continued - The Rest of the JDK, by Campione, Walrath, Huml, Tutorial Team, published by Addison Wesley, 1999, 0-201-48558-3

Java in Practice - Design Styles and Idioms for Effective Java, by Nigel Warren and Philip Bishop, published by Addison Wesley, 1999, 0-201-36065-9

Java Pitfalls, by Michael C. Daconta, Eric Monk, J Paul Keller and Keith Bohnenberger, published by John Wiley & Sons, 2000, 0-471-36174-7

Java and XML 2nd Edition, by Brett McLaughlin, published by O'Reilly, 2001, 0596001975

Java Cookbook 2nd Edition, by Ian Darwin, published by O'Reilly, 2004, 0-596-00701-9

Java in a Nutshell 4th Edition, by David Flanagan, published by O'Reilly, 2002, 0-596-00283-1

Java Extreme Programming Cookbook, by Eric Burke and Brian Coyner, published by O'Reilly, 2003, 0-596-00387-0

Java Puzzlers, by Joshua Bloch and Neal Gafter, published by Addison Wesley, 2005, 0-321-33678-X

Better, Faster, Lighter Java, by Bruce A. Tate and Justin Gehtland, published by O'Reilly, 2004, 0-596-00676-4

Tools Books

Ant - The Definitive Guide, by Jesse Tilly & Eric M. Burke, published by O'Reilly, 2004, 0-596-00184-3

Eclipse in Action, David Gallardo et al, published by Manning, 2003, 1-930110-96-0 (how to use the eclipse programming environment)

Eclipse: A Java Developer's Guide, by Steve Holzner and Bill Rosenblatt, published by O'Reilly, 2004, 0596006411

Eclipse Cookbook, by Steve Holzner, published by O'Reilly, 2004, 0596007108

Java Tools for eXtreme Programming, by Richard Hightower and Nicholas Lesiecki, published by Wiley, 2002, 0-471-20708-X

Java Development with Ant, Erik Hatcher and Steve Loughran, published by Manning Press, 2002, 1930110588

Pragmatic Version Control using Subversion, by Mike Mason, published by Pragmatic Bookshelf, 2005, 0-97451-6-3

Programming and Design Books

These books deal with design and programming, in a variety of programming languages, not just in Java. They are all worth a read.

Extreme Programming Explained, by Kent Beck, published by Addison Wesley, 1999, 0-201-61641-6 (another book for your 'must-read' list)

Extreme Programming Explored, by William C. Wake, published by Addison Wesley, 2001, 0-201-73397-8

Extreme Programming in Practice, by James Newark and Robert C. Martin, published by Addison Wesley, 2001, 0-201-70937-6

The Practice of Programming, by Brian W. Kernighan and Rob Pike, published by Addison Wesley, 1999, 0-201-61586-X

Open Source Development with CVS, by Karl Fogel, published by Coriolis, 1999, 1-57610-490-7

Programming Pearls - Second Edition, by Jon Bentley, published by Addison Wesley, 2000, 0-201-65788-0

Bringing Design to Software, by Terry Winograd, published by Addison Wesley, 1996, 0-201-85491-0

Object Thinking, by David West, published by Microsoft Press, 2004, 0-7356-1965-4

Working Effectively with Legacy Code, by Michael C. Feathers, published by Prentice Hall, 2005, 0-13-117705-2

Pragmatic Project Automation, by Mike Clark, published by The Pragmatic Bookshelf, 2004, 0-9745140-3-9

Design Pattern Books

Design patterns are becoming an important part of a programmer's knowledge, so you need to be familiar with what they are, and the names and details of commonly used patterns. I recommend that you read at least one of these books — this is not something you should treat as optional, you need the pattern knowledge in these books.

Design Patterns Explained, by Alan Shalloway, James J. Trott, published by Addison Wesley, 2001, 0201715945 (strongly recommended).

Java Design Patterns, by James W. Cooper, published by Addison Wesley, 2000, 0-201-48539-7 (this is quite an accessible book with lots of code examples).

Patterns in Java Vol.1 by Mark Grand, published by John Wiley & Sons, 1998, 0-471-25839-3

Design Patterns Java Workbook, by Steven John Metsker, published by Addison Wesley, 2002, 0-201-74397-3

Applied Java Patterns, by Stephen Stelting and Olav Maassen, published by PH/PTR, 2002, 0-13-093538-7

A System of Patterns - Pattern-Oriented Software, by Frank Buschman et al., Wiley 1996, ISBN 0-471-95869-7.

Design Patterns, Elements of Reusable Object-Oriented, by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Addison Wesley 1995, ISBN 0-201-63361-2. (This is the

original book on programming design patterns and is the best selling computer science text book of all time. It uses the C++ and Smalltalk languages for examples.)

Pattern and Software Architecture

These are more specialised books looking at patterns and software architecture.

Pattern Hatching — Design Patterns Applied, by John Vlissides, published by Addison Wesley, ISBN 0-201-43293-5 (This is a C++ book, but is worth looking at.)

Pattern Languages of Program Design, ed. by James O. Coplien and Douglas C. Schmidt, published by Addison Wesley, ISBN 0-201-60734-4

Pattern Languages of Program Design 2, ed. by John Vlissides, Jim Coplien and Norman Kerth, published by Addison Wesley, ISBN 0-201-89527-7

Pattern Languages of Program Design 3, ed. by Robert Martin, Dirk Riehle & Frank Buschmann, published by Addison Wesley, ISBN 0-201-31011-2

Pattern Languages of Program Design 4, ed. by Neil Harrison, Brian Foote and Hans Rohnert, published by Addison Wesley, 2000, 0-201-43304-4

Patterns of Software — Tales from the Software Community, by Richard P. Gabriel, published by Oxford University Press, ISBN 0-19-510269-X

Designing Hard Software - The Essential Tasks, by Douglas Bennett, PTR/Manning, 1997, ISBN 0-13-304619-2 (Hard to read but worth it.)

Software Architecture, by Mary Shaw & David Garlan, published by Prentice Hall PTR, ISBN 0-13-182957-2

Software Architecture in Practice, by Len Bass, Paul Clements and Rick Kazman, published by Addison Wesley, 1998, 0-201-19930-0 (Another book worth looking at.)

Design & Use of Software Architectures, by Jan Bosch, published by Addison Wesley, 2000, 0-201-67494-7

Applied Software Architecture, by Christine Hofmeister, Robert Nord and Dilip Soni, published by Addison Wesley, 2000, 0-201-32571-3

Building Architecture Books

These books look at how building are designed and constructed. They have greatly influenced those working with design patterns in the software development community.

The Timeless Way of Building, by Christopher Alexander, Oxford University Press, ISBN 0-19-502402-8 (This book is the source of all patterns inspiration — read this book.)

A Pattern Language. Towns, Buildings, Construction, by Christopher Alexander et al., published by Oxford University Press, ISBN 0-19-501919-9

A Place of My Own - The Education of an Amateur Builder, by Michael Pollan, published by Bloomsbury, ISBN 0-7475-3513-2

How Buildings Learn - What happens after they're built, by Stewart Brand, published by Viking, ISBN 0-75380-0500 (This book is really, really worth the time to read.)

Looking Around - A Journey Through Architecture, by Witold Rybczynski, published by Penguin, ISBN 0-14-016889-3

In addition to books there are many resources available on web sites. Use a search engine like Google to locate them (make sure you learn how to do effective searches). Some web sites worth looking at are:

- java.sun.com and www.java.com
- www.apache.org
- ant.apache.org
- testng.org
- www.junit.org

- www.extremeprogramming.org

4. Software Design - The Wider Context

It is easy, and tempting, to write small programs simply by sitting down in front of a computer and typing in the code as you think of it *without a proper testing strategy* (although perhaps not quite so easy to get that code to do what you actually want!). A well known object-oriented guru calls this 'programming by wandering about'.

However, as soon as your program gets beyond a trivial size, this won't work. There will be so much detail you need to remember about the code that you find it will slowly degenerate into a useless mess — a process that can be described as *software rot*. Many programming projects founder this way; we all do it and we all regret it.

The way to avoid these sorts of problems lies, not surprisingly, in the effective use of planning, design and testing. You need to give your software order and structure, properly defining the abstractions, interfaces, classes, components, packages and all the other features needed to control the complexity of it all. But how is this achieved? This course looks at some of the foundations of good programming practice but you must be aware of the wider context. 2009 will cover some of this in more detail but it is really up to you to make use of the reading list and understand what this is all about.

4.1 What are Patterns and Software Architecture?

"A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." — Christopher Alexander

Patterns capture, document and communicate design problems and their solutions. Patterns are very valuable for those learning about software development as they distil the collective experience of skilled software developers in a way that can be presented to and used by learners. This includes proven experience, robust solutions to many design problems and the "craft knowledge" of how to go about building systems.

The use of patterns is becoming diverse, ranging from low level *idioms*, through *design patterns* and on to complete system *architectures*. Moreover, patterns are not restricted to software implementation but can also be used to describe the processes via which many work related tasks are structured and achieved. Notably this is happening in the area of financial systems, where banks and trading houses in particular have picked up the ideas very quickly.

And what about software architecture?

"The architecture of a software system defines that system in terms of computational components and interactions between those components", Mary Shaw and David Garlen, 1996. They also describe the need for software architecture as follows:

"As the size and complexity of software systems increase, the design and specification of overall system structure become more significant issues than the choice of algorithms and data structures of computation. Structural issues include the organisation of a system as a composition of components; global control structures; the protocols for communication, synchronisation, and data access; the assignment of functionality to design elements; the composition of design elements; physical distribution; scaling and performance; dimensions of evolution; and selection among design alternatives."

Until recently architecture has been defined largely in terms of modules and interfaces, classes and objects. Now, architectures are now being described using patterns. Software architecture is important as the architecture of your program must be correct in order to make progress. Get your architecture wrong and you're sunk, the only sensible way to proceed is to throw away what you have built and start again.

4.2 Design is Hard Work!

There is no denying that design is hard work. It is a mixture of knowledge, experience, innovation, luck, experimentation, making mistakes, insight, science, engineering and more.

Software design is made more difficult by the relative lack of experience and established engineering practice — we have only been designing software for a few decades, in a context where hardware and development tools are constantly changing. However, things are slowly moving forward. Experience is being accumulated, understanding is improving; test-driven development, object-oriented design and patterns are at the forefront of the process. You need to study patterns, think hard about design and know how to test.

When you write programs, such as when doing the programming work for this course, don't simply focus on the programming language but also work on the design, and make testing part of the design process. Is your design good? How do you know? How can it be made better? What are the alternatives? Can the code implementing the design be tested properly? What would an expert do? Keep asking yourself these sorts of questions. The course lectures will go a little way to providing you with answers (or, at least, ways of asking the right questions) but you have to fill in the rest by doing lots of background reading as well as doing the programming.

All this talk of architecture, structure and design may bring to mind ideas of buildings and building architecture. And so they should! Software patterns and architecture draw heavily from ideas on building architecture and design, an activity that has been going on for, at least, five thousand years. There is a lot experience to be exploited and knowledge to be gained at by looking at how buildings are designed and constructed.

Go and read about building architecture and what architects do; some useful books are recommended in the reading list. Also consider design in other disciplines such as electrical or chemical engineering. How do people go about designing digital electronic systems or chemical plants? What can software developers learn?

4.3 Engineering...

A final thought. You will have heard the term "software engineering" and you may wish to become a "software engineer". Is the activity of designing and constructing software actually engineering?

Well, here is an answer: No it isn't. As yet we simply don't know how to do true engineering of software. Amazing if you consider how important software is in the modern world.

Do you agree or disagree? By the end of the second year you should be able to form your own opinion based on your knowledge and understanding of the subject area.

In the longer term we do want software development to have strong engineering qualities, possibly becoming a full engineering discipline. Find out what engineering is and how software engineering might be achieved.