CS242

Designing Object-Oriented Software with Patterns and Frameworks

Course Overview

Douglas C. Schmidt (314-935-4215)
http://www.cs.wustl.edu/∼schmidt/
schmidt@cs.wustl.edu

Patterns of Learning

Good design and programming is not learned by generalities, but by seeing how significant programs can be made clean, easy to read, easy to maintain and modify, human-engineered, efficient, and reliable, by the application of good design and programming practices. Careful study and imitation of good designs and programs significantly improves development skills.
– Kernighan and Plauger.

When people begin to learn to play chess they first learn all the rules and physical requirements of the game.[1] They learn the names of all the pieces, the way that pieces move move and capture, and the geometry and orientation of the board.

At this point, people can play chess, although they will probably not a very good players. But as they progress, they will learn the *principles* of the game. They learn the value of protecting their pieces, and the relative value of those pieces. They learn the strategic value of the center squares and the power of a threat. They learn how the king can oppose the enemy king and how even one passed pawn can win the game.

At this point, people can play a good game of chess. They know how to reason through the game and can recognize "stupid" mistakes.

However, to become a master of chess, one must study the games of other masters. Buried in those games are patterns that must be understood, memorized, and applied repeatedly until they become second nature. There are thousands upon thousands of these patterns. Opening patterns are so numerous that there are books dedicated to their variations. Midgame patterns and ending patterns are also prevalent, and the master must be familiar with them all.

So it is with software. First one learns the rules. The algorithms, data structures and languages of software. At this point, one can write programs, albeit not very good ones. Later, one learns the principles of software design. Structured programming, modular programming, object oriented programming. One learns the the importance of cohesion and coupling, of information hiding and dependency management.

But to truly master software design, one must study the designs of other masters. Deep within those designs are patterns that can be used in other designs. Those patterns must be understood, memorized, and applied repeatedly until they become second nature.

[1] This chess analogy is courtesy of Robert Martin (rmartin@oma.com).

Contents

CS 242 focuses on the practical aspects of software design and implementation. The concepts addressed in the course cover a broad set of topics important to developing and maintaining quality software:

- *Overview of the software life cycle*
- *Reuse of design patterns and software architectures*
- *Developing, documenting, testing and applying reusable class libraries and object-oriented frameworks*

Abstraction based on design patterns and object-oriented techniques (such as separating interfaces from implementations) will be the central concepts and principles throughout the course. These concepts and principles will enable you to construct reusable, extensible, efficient, and maintainable software.

Design patterns will be taught so that you will have good role models for structuring their own designs, as well as to clearly articulate the tradeoffs of alternative methods for designing systems. Object-oriented techniques will be taught so that you will learn by example how to build highly decentralized software architectures that decouple inter-dependencies between components. The course will cover object-oriented techniques that address:

- *Decentralized software architecture*
- *Design patterns*
- *Component reuse*
- *Class interface definition*
- *Module decomposition*
- *Hierarchical classification*
- *Extensible object-oriented application frameworks*

Several in-depth case studies will be used in class to illustrate the OO design and programming process.

Course Work

We will construct software components and applications using popular programming tools available in the UNIX environment (such as C++, make, emacs, dbx, RCS, gprof, and CORBA). You are expected to be familiar with an object-oriented programming language. Implementation will be done using C++.

All course material will be available on-line using the WWW at the following URL:

http://www.cs.wustl.edu/~schmidt/cs242/

- **Programming Projects**

  There will be a number of programming projects that will build upon one another to illustrate the value of good design and implementation techniques on the software lifecycle. Online descriptions of these assignments will be available via the WWW at the following URLs (all of which are prefixed by `http://www.cs.wustl.edu/~schmidt/cs242`).

An important goal of the assignments is to teach you about design patterns by comparing and contrasting different design alternatives. Algorithms and data structures taught in CS241 will be used to reinforce the implementation of these projects.

Projects will involve substantial programming in C++ and UNIX and will be done individually by each student. Students are assumed to be competent in C++ and familiar with basic UNIX operating system features such as electronic mail, WWW browsers, and USENET newsgroups. Students not familiar with these tools will have to learn them outside of class. I will teach a short seminar on C++ early in the semester during my office hours.

To encourage you to write well-designed software, I reserve the right to change the assignment specifications at any point before the due date. Expect this to happen several times during the semester. If you have written your program in a modular fashion the changes will be straightforward to implement.

The programs will be graded using the following criteria:

40%	execution correctness
30%	program structure (*e.g.*, modularization, information hiding, etc.)
10%	insightful programming (*e.g.*, developing reusable class components, etc.)
10%	Consistent style (*e.g.*, capitalization, indenting, etc.)
10%	appropriate commenting

There will be a mandatory **5** point deduction (out of a possible **100** points) for each day that your program is late after the deadline. Moreover, I will not accept programs that are turned in later than two calender days after the due date.

- **Quizzes**

  There will be a short graded quiz at the end of class each Thursday, starting on January 18th. The quizzes will be based on material presented in class. Therefore, it is essential that you attend class in order to prepare for the quizzes and exams. There will be no “makeup” quizzes unless you ask permission from me before the quiz.

- **Exams**

  There will be one in-class exam: a final. The final will be worth 25% of your grade. Material tested on the final exam will be comprehensive.

The relative weighting of each portion of the course is presented below:

25%	Exams
40%	Programming projects
35%	Quizzes

Note that I reserve the right to change the weights during the course of the semester.

Textbooks

Required Course Textbooks:

- *Design Patterns: Elements of Reusable Object-Oriented Software*, Gamma et at., Addison-Wesley, Reading, MA, 1994
- *The C++ Primer* (Second Edition) by Stanley Lippman

Recommended Reading:

- *Object-Oriented Design with Applications* (Second Edition) by Grady Booch
- *Object-Oriented Software Construction* by Bertrand Meyer
- *The C++ Programming Language* (Second Edition) by Bjarne Stroustrup
- *The Annotated C++ Reference Manual* by Stroustrup and Ellis
- *Effective C++* by Scott Meyers

Detailed Course Outline

Topic 1: *Object-Oriented Design and Programming*

1. Overview of OOD and OOP
2. Overview of OO modeling techniques and notations
3. Comparison of OO with other design and programming paradigms

Topic 2: *C++ programming*

1. Review of C++ features (*e.g.,* classes and dynamic allocation)
2. C++ object-oriented features (*e.g.,* inheritance and dynamic binding)
3. C++ generic programming features (*e.g.,* templates)

Topic 3: *Design Patterns*

1. Overview and motivation
2. Tactical patterns (*e.g.,* Adapter, Wrapper, Factory Method, Bridge, Strategy, etc.)
3. Strategic patterns (*e.g.,* EG Reactor, Active Object, Acceptor, Connector, etc.)

Topic 4: *OO Frameworks*

1. Overview
2. Comparison of OO frameworks and class libraries
3. Implementing and documenting OO frameworks with design patterns

Topic 4: *Case Studies Developing OO Applications using Patterns, Frameworks, and C++*

1. Arrays and Stacks
2. Balanced delimiter checking
3. Thread-specific storage manager
4. System sort utility
5. Sort verification
6. Operator precedence parsing and binary trees