COP 3330: Object-Oriented Programming
Summer 2011

Introduction to Object-Oriented Programming

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cop3330/sum2011

Department of Electrical Engineering and Computer Science
Computer Science Division
University of Central Florida

Programming

- A *program* is a set of instructions to perform a given task. Typically the task solves some problem or part of a larger problem.
- In your CS classes, so far, you've learned the fundamental programming concepts, such as loops, assignments, conditional statements, and so on. Hopefully, you learned one or more ways to implement each of those constructs.
- You also learned that computer science is not just programming, but rather that programming is just a tool of a computer scientist.

Programming

- What you might not have learned yet, are the advantages and disadvantages of the different ways to write a program to solve the same problem.
- For example, at a low level, there are many different ways to implement a structure such as a loop, e.g., you could use a for loop, a while loop, or recursion. At a higher level, there are many different ways to perform a task, e.g., a merge sort versus a selection sort. At even higher levels there are many different ways to divide a program into modules such as classes and methods.
- As an example of the latter case, consider the problem a writing a module that is supposed to maintain a collection of people and the dogs they own. Should the collection be a hash table or an array or some other kind of collection? Should the collection contain Person objects or maybe Person-Dog pairs of objects? Should each dog have an instance variable that points to its owner? Should owners have an instance variable that refers to a collection of dogs they own? If so, what kind of collection should it be? What other data should the Person object store? If the people are US citizens, do you need to store their social security number? If a person's only dog dies, should that person be removed from the collection or left in as a person who owns no dogs?

Programming

- At this point you might reasonably ask whether the answers to these questions really matter. If two designs, one with few classes and methods and one with many, both solve a problem, does it matter which one you use?
- Similarly, if all versions of a loop correctly perform a computation, does it matter which one you use? If two algorithms both work correctly, does it matter that one is faster, especially given the increasing speeds of processors?
- The answer is: YES, it really does matter!

Programming

- Before we write a computer program to solve a problem, we should organize its solution. (*problem solving*)
- Normally computer scientists are good at problem solving, but we should apply certain methods to solve problems (especially when we solve large problems) elegantly.
- Good problem solving steps make life easier when we write a computer program to solve a given problem. We will talk about top-down approach (divide and conquer) when we organize solutions for problems.
- We will also talk about object-orient software development techniques.

Programming

- Why should we worry about writing elegant "error-free" code? Consider some of the following situations which have occurred over the years:

1. In 1962, the Mariner I spacecraft lifted off for its voyage to Venus but was destroyed by the people running the mission because, due to a bug in the ground-based computer system, they incorrectly thought the booster rocket had malfunctioned (it hadn't).
2. Between 1985 and 1988 there were six cases of patients being given massive overdoses of radiation from a Therac25 radiation therapy system. Part of the blame was due to a error in the control software for the system.
3. In 1993, a bug in the SunSoft operating system I/O library delayed a corporate $20 million sale (costing the purchasing company an extra $5 million). The problem was eventually traced to a statement that read `x == 2` instead of `x = 2` in a C program.
4. On January 16, 2006, a software upgrade to improve security for ATMs (in the UK) had a flaw that allowed anyone to withdraw any amount of cash they liked using any password they wanted. About £850,000 (about $1.5 million at the time) was withdrawn before analysts caught the problem. Since the "customers" were not identifiable in any way, none of this money was ever recovered.

Programming

- These problems are just the tip of the iceberg. According to a National Institute of Standards report in 2008, software bugs cost the US economy $58 billion annually!
- The report further states that software developers spend 80% of their development time finding and fixing bugs.
- One might argue that all of these problems merely indicate a failure on the part of the programmer to write error-free code.
- While, in a sense, that argument is true, but avoiding such failures is not that simple. When a program contains thousands or millions of lines of code, it is inevitable that there will be bugs in it.
- The real issue is how to minimize the number of bugs that occur when the code is written in the first place, how to maximize the detection and removal of the bugs that do make it into the code, and how to minimize the number of new bugs that are accidently introduced whenever the code is modified.

Programming

- Note that this minimization/maximization process is not a one time thing. Software continually changes due to patches introduced to fix bugs or due to enhancements added to the software and any change can introduce more bugs.
- While thorough testing is important to remove bugs, it is just as important to design and write the software so that as few bugs as possible are introduced in the first place and so that it is easy to modify the code later without introducing new bugs.
- Unfortunately, there are many forces at work against software developers and the development of bug-free code. Doing the job right takes time and money in the short term and the benefits do not appear until later. At the same time, software projects are under more and more pressure to be completed quickly and put into production before the window of opportunity for sales closes. As a result, the initial software design is often inadequately specified and once in production, the pressure to quickly fix the bugs and enhance the software works against major redesigns of the software.

Programming

- As a result, the software system tends to become a “big ball of mud”, that is a haphazardly structured, spaghetti-code jungle. The degradation over time makes finding and fixing bugs and adding enhancements harder and harder, costing more time and money and resulting in more pressure to put off large-scale redesign, and so a vicious cycle is created.
- Other forces too, push software toward the same balls of mud. These forces include the lack of skill, knowledge, and experience of the software developers regarding how to write high quality software.
  - If the developers have no experience with designing software systems of any size or complexity or if the developers are writing a business application but have no knowledge of that particular business domain and its needs and requirements, then it is easy for the software to become muddy. Even if the developers understand a system completely, the mud in the system will not go away if the developers don’t have the tools (skill or knowledge) to clean it up.

Programming

- What can be done to fight the tendency of software systems to turn into mud balls?
- Addressing the pressures of cost and time are beyond the scope of this course. Our focus concerns the skills, knowledge, and experience that developers need in order to do high quality work.
- In this course, you’ll learn some of the things a software developer should know in order to design and implement high quality software systems and be able to redesign existing software systems in a way that fight the forces of mud.

Software Engineering

- To create high quality software, you first need to know what it means for software to have high quality.
- Unfortunately, such quality is not easily measured.
- The field of software engineering was created with the purpose of understanding and devising ways of measuring the quality and reliability of software. One of its tools has been the application of engineering principles to software development.
- Software engineering traditionally divides the software process into stages, including specification and analysis, design, implementation, and maintenance.

Software Engineering

1. *Specification and Analysis*
   - Determine the precise behavior the final software system is to have. Exhaustively determine what the system should do in all possible cases. This stage also determines what the user interface will be. This stage is performed in close cooperation with the clients of the system and some users of the system. You must understand the problem completely and determine what is required for its solution.
2. *Design*
   - Determine the components of the system, what each component is responsible for doing, and how the components will interact. For example, this stage includes determining the data structures that will be used and the kind of data that will be stored in those structures.
   - Develop a list of steps (algorithm) to solve the problem
   - Refine steps of this algorithm. (Divide and Conquer)
   - Verify that the algorithm solves the problem, i.e. the algorithm is correct

Software Engineering

3. *Implementation*
   - Programmers develop the code of the software system components using an appropriate (or several appropriate) programming language.
   - You have to know a specific programming language (Java).
   - Extensive testing is part of this stage.
4. *Maintenance*
   - Once the product has shipped or been placed into service, programmers repair defects, update or enhance the software to extend its usefulness.
   - Testing is also crucial in this stage.
   - Use different test cases (not one) including critical test cases.

Criteria For Elegant Software

1. Usability
   – Is it easy for the client to use?
2. Completeness
   – Does it satisfy all of the client's needs?
3. Robustness
   – Will it deal with unusual situations gracefully and avoid crashing?
4. Efficiency
   – Will it perform the necessary computations in a reasonable amount of time and using a reasonable amount of memory and other resources?

Criteria For Elegant Software

5. Scalability
   - Will it still perform correctly and efficiently when the problems grow in size by several orders of magnitude?
6. Readability
   - Is it easy for another programmer to read and understand the design and code?
7. Reusability
   - Can it be reused in another completely different setting?
8. Simplicity
   - Is the design and/or implementation unnecessarily complex?

Criteria For Elegant Software

9. Maintainability
   - Can defects be found and fixed easily without adding new defects?

10. Extensibility
    - Can it be easily enhanced or restricted by adding new features or removing old features without breaking the code?

- The first four properties mostly relate to the functional requirements of the software, i.e., does it do what the requirements documents say it is supposed to do?
- The last six properties are the ones that we will be most concerned with in this course. They address software style and how easily the software can be changed.