CPS 108, Spring 2001

- Software Design and Implementation
  - Object oriented programming and design
  - good design helps do away with late night Teer-fests, but some late nights are inevitable
  - your toolkit must include mastery of language/programming and design

- What's in the course?
  - C++ and Java, team projects, mastery exams
  - team projects can be more and less than the sum of their parts
  - high-level abstractions, low-level details
  - patterns, heuristics, and idioms

Program Design and Implementation

- Language independent principles of design and programming
  - design heuristics
    - coupling, cohesion, small functions, small interfaces ...
  - design patterns
    - factories, adapter, MVC aka observer/observable, ...

- Language specific
  - idioms
    - smart pointers, vectors/arrays, overloaded operators ...
  - idiosyncracies, idiosycies
    - must define virtual destructor, stream zoo in Java, ...

Administrivia

- check website and news regularly
  - duke.cs.cps108

- Grading (see web pages)
  - group projects: small, medium, large
  - mastery programs (solo or semi-solo endeavors)
  - readings and summaries
  - tests

- Evaluating team projects, role of TA, UTA, consultants
  - face-to-face evaluation, early feedback

- Compiling, tools, environments, Linux, Windows
  - g++ 2.95, Java 2 aka 1.2, JRE, ...

C++ idioms/general concepts

- Genericity
  - Templates, STL, containers, algorithms

- Copy/Assignment/Memory
  - Deep copy model, memory management "required"

- Low-level structures
  - C-style arrays and strings compared to STL, Tapestry

- const
  - Good for clients, bad for designers/coders?

- From C to C++ to Java
  - function pointers, function objects, inheritance
Java idioms/general concepts

- Primitive and other types
  - Object superclass, static vs dynamic allocation
- References, Shallow copy
  - Equal vs ==, clone
- Applets and Applications
  - Do we want MSWord over the web?
- Package concept for modularity
  - Nothing similar in C++ (what about C++ friendship?)
- API/Library
  - Enormous, GUI, Graphics, Network, XML, ...

From C++ to Java

- Java history: Oak, toaster-ovens, internet language, panacea
  - Not really a standard language like C++
  - Arguably proprietary (and arguably not)
  - Precursor to C#?
- What it is
  - O-O language, not a hybrid like C++
  - Compiled to byte-code, executed on JVM
  - Byte-code is “highly-portable”, write once run “anywhere”
    simple, object-oriented, portable, interpreted, robust, secure,
    architecture-neutral, distributed, dynamic, multithreaded,
    high performance

Classes: Review/Overview

- A class encapsulates state and behavior
  - Behavior first when designing a class
  - Java interfaces enforce behavior first, abstract base classes are similar in C++ (but can have state)
- State is private/protected; some behavior is public
  - Private/protected helper functions
  - A class is called an object factory, creates lots of instances
- Classes communicate and collaborate
  - Parameters: send and receive
  - Containment: has a reference to
  - Inheritance: is-a

C++ and Java class construction

- C++ uses .h and .cpp, Java uses .java
  - Documentation different (javadoc vs. ???)
- Default, overloaded, copy constructor
  - tvector, string, Date
  - Default constructor needed in C++, where?
  - Copy constructor needed to avoid shallow copy
  - In C++ destructors needed to free resources/self, Java?
  - Clone makes copy in Java (rare), share is default
- Private, protected, public, (package)
  - Private default in C++, package default in Java
  - Per method declaration in Java, class sections in C++
Design Criteria

Good design comes from experience, experience comes from bad design

Fred Brooks (or Henry Petroski)

- Design with goals:
  - ease of use
  - portability
  - ease of re-use
  - efficiency
  - first to market
  - ????

How to code

- Coding/Implementation goals:
  - Make it run
  - Make it right
  - Make it fast
  - Make it small
  - spiral design (or RAD or ‘waterfall or ...)

  ➤ what’s the design methodology?

XP and Refactoring

(See books by Kent Beck (XP) and Martin Fowler (refactoring))

- eXtreme Programming (XP) is a lightweight design process
  - Communication: unit tests, pair programming, estimation
  - Simplicity: what is the simplest approach that works?
  - Feedback: system and clients; programs and stories
  - Courage: throw code away, dare to be great/different

- Refactoring
  - Change internal structure without changing observable behavior
  - Don’t worry (too much) about upfront design
  - Simplicity over flexibility (see XP)

Design Heuristics: class/program/function

(see text by Arthur Riel)

- Coupling
  - classes/modules are independent of each other
  - goal: minimal, loose coupling
  - do classes collaborate and/or communicate?

- Cohesion
  - classes/modules capture one abstraction/model
  - keep things as simple as possible, but no simpler
  - goal: strong cohesion (avoid kitchen sink)

- The open/closed principle
  - classes/programs: open to extensibility, closed to modification