Bridging the Introductory Course Sequence (221/222/321) and 560 – A Working Document

Draft: Thursday, November 2, 2006
Author: Paul Sivilotti
In consultation with: Wayne Heym, Rajiv Ramnath

New Course Proposal
CSE 4??:01 Software Development in Java
or Programming in the Small in Java
or The Java Language and Tools
and
CSE 4??.02 similar course for C++

Strengths of current 221-321/459/560 course structuring:

1. *Foundations-first approach in intro sequence.* Students learn about specification, reasoning about correctness, abstraction, design-by-contract, aliasing, etc. They learn principles that they should be able to apply throughout their academic and professional careers.
2. *Keystone role of 560*. The team-based large implementation project is a critical component of our curriculum. This course serves as a prerequisite for many (most?) advanced courses and has been an important part of our ABET accreditation. Students gain significant "programming maturity", as well as important experience with writing documentation, testing, designing, and working as a team.
3. *Language neutrality of 560*. Making language choice part of the design space for 560 teams underlines the fact that it is a blank-slate implementation (of a significant project). Discussion of "programming in the large" techniques (and even tools) is done primarily in a language-independent manner, emphasizing fundamentals.
4. *Pluralism of 459's*. Learning a breadth of languages enriches students' understanding of a variety of concepts. At the same time, specifics of language syntax have a shorter period of relevance than fundamentals do. It makes sense, then, to partition out these specifics to a part of a curriculum that is easily modifiable. This is planning for change in the Parnas sense.

Problems/frustrations:

1. *Tools in 560*. In 560, we want group projects to begin almost immediately: The bigger the lab (and the longer the students have to work on it) the better. But students have not yet learned many tools that are useful for "programming in the large" (CVS, Make, testing harnesses, debugger) until (midway through) 560. As a result, this information comes too late to be helpful in the early lab(s).

2. *Student appreciation for RESOLVE foundations*. In 560 (and beyond?), students working in languages other than RESOLVE seem to have difficulty mapping RESOLVE concepts into their group's implementation language. For example, they don't see the relevance of defining a mathematical model, or of distinguishing between concrete and abstract state.
3. *Complexity of Java*. It is hard to do Java justice in a 1-credit S/U class. In 459.23 (Programming in Java), it is practically the end of the quarter before students can do anything interesting in Java. In order to get to the Collections Framework, one must first cover interfaces, inheritance, generics, and exceptions as well as the OO basics (objects, classes, encapsulation, syntax).

Proposal: Design a new class that serves as a "bridge" class, sitting between the intro course sequence and 560. The educational objectives of this class would be:

1. Introduce basic language syntax and semantics, giving students the basic skills needed for them to write their own programs.
2. Reinforce solid design principles from intro sequence by illustrating their manifestation (or approximate manifestation) in a mainstream programming language (in particular, Java).
3. Relate these design principles to a set of "best practices" for Java programming
4. Describe limitations of Java (eg behavioral subtyping, aliasing, shallow/deep copying)
5. Expose students to language-based software development tools, in particular:
    - JUnit, for testing
    - Javadoc, for documentation
    - Ant, for makefiles
    - Eclipse, as an IDE, including CVS and thedebugger
6. Implement several moderately sized Java programs

Course Weighting: Ideally, this would be 3 credit hours, with approximately 1 credit hour dedicated to each of:

1. Language mechanics (classes, objects, encapsulation, inheritance, interfaces, exceptions, generics, libraries)
2. Tools (Eclipse, CVS, debugger, JUnit, Javadoc, Ant)
3. Best practices (immutable objects, coding to the interface, getters/setters for abstract state, behavioral subtyping, implementing equals/clone, singleton objects)

Placement in curriculum:

Concurrent with 321, ie:
Prereq: 222
Postreq: 560

Issues to be addressed:

1. Where will credits come from?
2. Centrality of 560 means changes here must be done with caution

3. Is it too ambitious to hope that these students will recognize the RESOLVE concepts as they are applied in a new language?

Discussion points:

1. Decreasing 560 weighting from 5 to 4 credit hours.
    - In favor: Tool material would move out of the course (1 lecture on Make, 1 lecture on CVS, generic testing & documentation discussion would likely remain). Other systems software topics could be removed: CFG & regular expressions, compilers, lex & yacc, macro preprocessor algorithm
    - Against: This decrease reduces what we can expect from the students in terms of implementation project. With what we are trying to accomplish in 560 from an ABET perspective, 5 credit hours is what we would expect the weighting to be.
2. Sliding schedule for 560. Could there be more frequent meetings of 560 at the beginning of the quarter? For example, for a 5-credit 560, could there be 5 meetings / week in the first 3 weeks, then 4 meetings / week in the next 4 weeks, then 3 meetings / week in the last 3 weeks?

Specific Java constructs/Best Practices informed by RESOLVE

1. *Interfaces vs classes.* The abstract model and behavioral specifications correspond to an interface. Classes implement an interface like a Kernel implements a type.
2. *Encapsulation and modularity*. Private fields represent concrete representation. Abstraction relations and representation invariants must be considered when designing the private elements of a class.
3. *Dangers of aliasing*. Pass-by-reference-value means that all method calls create aliasing with accompanying problems (eg repeated arguments).
4. *Defensive copying*. Aliasing means that components should make deep copies of arguments passed in as constructors and that the component expects to have ownership
5. *Accessor methods.* Getter and setter methods in Java are usually designed (or generated even) from private fields. But these methods should really come from the abstract state, not the concrete representation.
6. *Use of Exceptions*. What does it mean for behavior to be "exceptional"? This should be related to observability. Real-world concurrency means that some preconditions can not be guaranteed by the caller.
7. *Exception Chaining*. A component may need to create a new exception type (and chain to the underlying cause) to isolate its clients from implementations more than 1 layer below.
8. *Inheritance*. Subclasses should be behavioral subtypes. This involves understanding invariants and method specifications.
9. *Implementing to the interface*. The right interface to choose from an inheritance hierarchy (eg SortedSet, Set, Collection, Iterable) depends on what functionality the client needs.

Topics in new course (Java version)

Key: L – Language
BP – Best Practices
T – Tools

Lectures	Key	Topic
1	L	Overview: compilation, primitive types
2	L	Objects and classes: instantiation, encapsulation, packages
1	L	Generics
2	L	Inheritance
1	L	Interfaces
1	L	Collections
1	L	Exceptions, Assertions
1	L	Logging, IO
1	L	Concurrency
1	L	Swing
1	L	Network Programming
1	L	Reflection
1	L	Garbage Collection
1	L	Nested classes and interfaces
1	L	Annotations
1	BP	Implementing equals (and hashCode)
1	BP	Implementing cloning and copy constructors
1	BP	Immutable objects (value objects)
1	BP	Factories
1	BP	Singleton objects
1	BP	Checked vs unchecked exceptions
1	T	Eclipse
1	T	CVS
2	T	JUnit
2	T	Javadoc
1	T	Debugger

Total: 30