The Coding Phase

- Goal: Produce executable code in a coding language
- Gets down to very specific details:
  - Procedures/algorithms
  - Data structures
  - The interactions between them
  - THE DEVIL IS IN THE DETAILS
- Coding is usually 10-15% of the effort on a software development project: We will spend little time on it in this course
- Coding should follow closely the specifications resulting from the final phases of design
  - Modular structure of the code
  - Object specifications (data modules) (Data abstractions)

What makes a programming language “good”?
If it meets the needs of its stakeholders

Coding

- Goal: Create code that can be executed on a computer
- Developer writes source code
- Object code emitted from a compiler
  - So, is code really just another model?
- Executable results from loading object code with libraries, utilities, etc.
- Important to keep all of these straight
- Some designed to support specific design methodologies
- Some are special-purpose, well adapted to application domains

What makes a programming language “good”?
If it meets the needs of its stakeholders
A “good” language is one that meets the needs of its stakeholders

- Different kinds of projects
  - Quality is super-important
  - Rapid deployment is key
  - Evolvability is paramount
  - Emphasis on user interface
  - Etc.
- Suggest languages with strengths like
  - Readability
  - Expressive power
  - Low level (close to the machine)
  - Dynamism and late binding
  - Etc.

On Languages

- Bad code can be written in any language
  - But some languages encourage bad practices
- Good code can be written in any language
  - But some languages encourage it/make it easier
  - And discourage bad practices
- Most modern languages try to encourage good practices
  - Like those we have been advocating (in discussing design)
  - Modularity
  - Information hiding
  - Data abstraction
  - Incorporation of design and requirements specification into code
  - Support for testing and analysis

Information Hiding in Implementation

- Implementation units should hide internal details as specified by a Modular design
  - Superior procedure semantics support this better
- Implementation units should communicate through well-defined interfaces (not global variables).
  - Some languages make global data easier than others
  - Some languages make it hard to inspect internals of Modules.
  - Others make it easier
  - Different decisions are harder or easier to hide
    - Algorithm
    - Data representation
    - Lower-level modules
    - Policy

Data Abstractions

- User’s (client’s)-eye view of the data types to be used
- Essentially the same as Parnas notion of a “data module” —and the notion of an “object”
- Cluster of “accessing primitives” / “methods” whose purpose is to provide the only mechanisms for manipulating data of a given type
- Problem: How to specify the semantics of these types —without specifying their implementation
- Being rigorous help separate (even slightly) different notions of an ADT from each other

Assertion Languages

- Assert statements to define assertions
  - Assertions defined by programmer
  - Locations identified by programmer
  - Reactions to violations defined by programmer
- Different assertion language semantics
  - Usually Boolean logic
  - Sometimes private data space

Tool Suites

- Better tools make languages more useful
- Better editors
- Better diagnostics
- Better testing aids
- More powerful libraries
- Etc.
Cobol

- COMMON Business Oriented Language
- (Maybe) the first programming language
- Focus on business data processing
- Very wordy
  - Making it very modern?
- Tight discipline on loops
  - Making it modern?
- Focus on resources needed
  - Making it modern?
- New code is rarely written in this language
- But hundreds of millions of lines of "legacy code" have key roles in our national infrastructure

Fortran

- FORMula TRANslation language
  - Intended to support numerical/scientific computation
- Developed at about the same time as Cobol
  - But for scientific computation
- Updated with periodic new versions
  - As recently as 1995
- Latest versions look a lot like Algol
- Relatively little Fortran code is written now
- Much of our scientific infrastructure is written in Fortran

Algol

- European language answer to Fortran
- Focus on formal semantics and definitions
- Use of recursion
- Emphasis on disciplined use of types
- Intended to be the basis for reasoning

Ada

- Early language that supported information hiding
  - Use of External and Internal part dichotomy
    - Strict encapsulation
- Support for data abstraction
  - Packages
    - Very wordy
    - Support for disciplined concurrency
    - No type hierarchy
    - Very static language

C

- Gets you down "close to the machine"
- Little restriction on use of pointers
- Little restriction (help with) dynamic storage allocation
- Little support for encapsulation
- A great deal of code is currently written in C
  - Much of it is badly done and dangerous
  - Abuse of pointers is particularly common and egregiously dangerous

C++

- Adds support for objects to C
- Thus, support for objects (encapsulation)
- Type hierarchies
- Still little discipline over pointers, storage allocation
- A great deal of code is currently being written in C++
  - And much of it is poorly done and dangerous
Java
• Early language with special attention paid to dynamism and the web
• Designed to facilitate distributed applications
  – Host readily on various machines (across the web)
• Support by lots of tools
• Highly dynamic language
  – Various sorts of late binding
• But more discipline than C (e.g. over use of pointers)

Lisp
• Very dynamic language
• Very little compilation done
  – Mostly interpretation
  – Create code “on the fly” and interpret it
• Virtually no concept of types
  – Types with Lisp extensions
• Primitive flow of control structures
• Very hard to encapsulate
  BUT it is an excellent vehicle for rapid prototyping
  – Ideal for some stakeholders/stakes

Ruby/Python/Perl
• Highly dynamic
• Interpreted, not compiled
• Weak type checking/enforcement
• Sometimes used as a scripting language, sometimes as general-purpose programming language
• Object-orientedness varies
• Extensive libraries
• Supporting frameworks
  – RoR for web applications
• Ideal for stakeholders with high stakes based upon rapid availability

Prolog
• Rule-based language
  – No real procedural flow of control
  – Emphasis on reaction
• Language for trying to capture human knowledge
• Data is subordinated
• Structuring, modularization are difficult

Patterns
• Higher level implementation constructs
• Idioms (Rich and Waters, ~1985)
• The “Gang of Four” book
  – Inspiration from “real” architects (C. Alexander)
• Idioms in common use
• Suggest ways that humans think/human esthetics
• Transcend specific languages
• Some finding more direct support in newer languages