Principles of Program Design

What should be taught in core programming curricula

Basic Principles

- Design the data first and let the structure of the data guide the structure of the code.
- Design for unit testing.
- Document all code with contracts.
- Avoid mutation.
- Avoid duplicating code [use patterns].
- Look for higher-order solutions (code as data).
- Simpler is better, if it meets requirements.
- In OO programs, maximize polymorphism.

Relationship to Extreme Programming

- XP only mandates
  - Design for unit testing
  - Simpler is better
- XP focuses on program evolution rather than program structure
- XP program evolution must be guided by intelligent design; otherwise the resulting code will be ugly

Design Principles To Supplement XP

- Data Directed Design
  - Taught in Comp 210
  - Key observation: most program data types have natural formulations as free term (word) algebras which are represented in OO languages using the composite pattern.

    Aside: free term algebras are simply context free grammars that define *trees* rather than *strings*.

Design Principles To Supplement XP

- Document code with contracts when feasible
  Why? So that the intended behavior of the code can be understood without reading the code.
  Reverse engineering contracts from code is extremely difficult and error-prone. Unit tests (examples) can help, but good contracts are equally important.
- Partial contracts are better than no contracts.

Design Principles To Supplement XP

- Avoid Mutation.
  - Mutating data breaks algebraic laws that we take for granted such as “substituting equals for equals”.
  - If a method uses mutation, it can potentially modify any accessible object. Hence, it is difficult to determine if calling such a method can break a given program invariant.
  - In the absence of mutation, objects can safely be shared (rather than copied). Example: DAG tree representations.
  - Objects subject to mutation cannot be used in contexts that depend on immutable values such as hash table keys.
  - In multi-threaded programs, access to immutable data does not need not be synchronized

When Mutation is Justified

- When execution recapitulates evolution. Many programs model processes that change over time (physical simulations, financial transactions, etc.). The data representing entities in the model changes as the modeled entities change. Examples:
  - Solving a time dependent partial differential equation
  - Banking software
  - Document editor

When Mutation Is Justified (cont.)

- When faster algorithms require mutation. Examples:
  - Depth-first searching of a DAG.
  - Memo-ization of a recursively defined function (dynamic programming from a functional perspective)
  - Avoiding recopying when attributing a tree, as in a compiler.

When Mutation is Justified (cont.)

- When the most accurate data model requires circularity.
  - Example: Josephus problem (see Knuth, *The Art of Computer Programming*, Volume 1: *Fundamental Algorithms*)

You cannot build circular structures without performing mutation.

When Mutation is Justified (cont.)

- When instrumenting a program. Examples:
  - Inserting print statements to log debugging output.
  - Maintaining event counters. (How many times is a particular constructor called?)

Note that instrumentation involves modeling the execution of a program, a process that unfolds in time. Hence, this justification is really a special case of “execution recapitulating evolution”.

When Mutation is Justified (cont.)

- When interacting with hardware (e.g., controlling devices via device registers)
- When interacting with agents external to the program. Some interactions can be modeled using lazy functional programming, *e.g.*, interaction involving console input and output. But this form of modeling does not address external persistent data structures like file systems.

Use Closures To Represent Dynamic Behavior