Finding and Reusing Programmers' Work

Peter Norvig

Sun Microsystems Laboratories

What to Reuse

- Anything a programmer or designer produces is a legitimate object of reuse
- Think in terms of reusing your programmers' work and time, in whatever form it takes. Don't limit yourself to reusing classes only.
- Consider reuse of:
    - Classes (data types and application frames)
    - Functions (protocols)
    - Control flow abstractions
    - Syntactic abstractions (macros, entire languages)
    - Data and data formats
    - Tools for documentation, testing, version control, etc.

Levels of Reuse

- Decide what level of reuse you want to support
- Choice can affect all stages of software development

Level	Reuse of	Motto
Instructional	Ideas	Read & Learn
Informal	Source code	Copy & Edit
Verbatim	Object code	Link & Load
Extensible	Protocols, Classes	Use & Extend
Stratified	Languages, Approach	Mix & Match

- Note that each level implies all the levels before it
- Later levels imply more sophistication on part of programmer/designer
  - This is inevitable
  - It is the major reason why reuse is difficult to achieve
  - Reuse at earlier levels may be ineffective

Design for Reuse

- The spirit of reuse:
  - *Don't do it twice if you can do it once!*
  - Think in terms of a lifetime, not just one program
- Design for the level of reuse you want - Have a policy
  - So that the reusers will know what to expect
- Reuse requires an understanding of implementation vs. specification
  - This is a good thing for other reasons as well
- Split each design into "reusable" and "custom" parts
  - Requires awareness of the range of possible uses
  - Even "custom" parts may end up being reused
- See a design as an assembly of appropriate components
  - Most of the components should have been written already

Reuse of Classes

- **Common data structures**
    - ***Sets, Sequences, Stacks, Tables, Trees, Queues, Graphs, ...***
- **Common programming tools**
    - ***Conditions, Dates, Filenames, Data set/Statistics...***
    - ***Windows, Menus, Dialog box, List editor, Lines, Polygons, ...***
- **Common programs**
    - ***Text editor, Spreadsheet, Whatever is important to you ...***
- **Important lessons:**
    - *No global variables!* **With reuse, there is no such thing as "global," only classes, subclasses and instances**
    - *No such thing as top level!* **Design every program so that it can be incorporated into another program. Make sure that multiple instances and multiple specializations can work together (Thread safe).**

Reuse of Functions or Protocols

- *Functions still matter!* **Decomposition into protocols is probably more important than decomposition into classes (consider IDL).**
    - **"Computer Science is actually a study of things that happen."**

      - Robin Milner
- **Design your programs to decompose into reusable tools**
- **If your language supports it, consider basing the design on generic function protocols rather than on classes**
- **Some useful techniques are not available in class-obsessed languages:**
    - **Multi-methods allow dispatch on more than one argument**

      Example: `draw(object,stream)` depends on both
    - **Method combination allows customization of a method for a subclass without having to duplicate code in the superclass**

      Example: audited transactions with `before` and `after` methods

Rule of English Translation

- English (or any other natural language) is highly reusable
  - Proof: We can talk to a stranger without having to invent each word and phrase we use
- Code that can be quickly read as English is easy to understand and reuse
- Code that requires the writer's annotations may not be
- Algorithm for easy to understand, reusable code:
  - Write a version of the code
  - Translate code into English
  - Translate English back into code
  - Repeat until both pieces of code are the same

Example of Functional Decomposition

- **Problem: `least_common_superclass` of a set of instances**
- **Nonreusable decomposition:**
  - **`loop, loop, loop`**

    (Also violates rule of English translation)
- **Locally reusable decomposition:**
  - **`superclasses, common_superclasses, more_specific_class`**
- **Globally reusable decomposition:**
  - **`least_upper_bound, select_best`**

Reuse of Control Flow Abstractions

- **Most algorithms can be characterized as:**
  - ***Searching, Sorting, Filtering, Mapping, Combining, Counting***
- **Code that uses loops to implement these is nonreusable**
- **Code that uses control flow abstractions is reusable**
  - **Build a library for each of these (either functions or macros)**
  - **Requires iterators or closures or powerful macros**
- **Invent abstractions for common programming situations**
  - `try` **+** `throw` **=** `unwind_protect`
  - **Non-determinism with** `choose` **and** `fail`

Reuse of Syntactic Abstractions

- "Expert engineers stratify complex designs ... The parts constructed at each level are used as primitives at the next level. Each level of a stratified design can be thought of as a specialized language with a variety of primitives and means of combination appropriate to that level of detail."

  - Harold Abelson and Gerald Sussman

- Consider the range of extensions and modifications for your program
- Implement a language that covers the range well
- Write your program in this new language
- When the specification changes, the changes to the program will be easy

  (Assuming you designed the right language)

- Problem: Not all programmers are good language designers

  (But not all of them need to be)

Naming Conventions

- **Most reuse starts with the name of the thing to be reused**
- **Therefore, names must be chosen wisely**
  - **Minimize abbreviations (Rule of English Translation)**

    Most words have many possible abbreviations, but only one spelling

    Possible exceptions: `char, demo, intro`
  - **Avoid ambiguous names**

    (Use `previous` or `final` instead of `last`)
  - **Have a shared glossary of terms from which to build names**

Reading is Prior to Reusing

- **"I've learned a lot from looking at other people's programs."**

  - Jonathan Sachs (Lotus 1-2-3)

- **"You need to study other people's work. Their approaches to problem solving and the tools they use give you a fresh way to look at your own work."**

  - Gary Kildall (CP/M)

- **"The best way to prepare [to be a programmer] is to write programs, and to study the great programs that other people have written. In my case, I went to the garbage cans at the Computer Science Center and I fished out listings of their operating system."**

  - Bill Gates (Altair Basic)

Trade-offs in Reuse

- *Plus:*
    - Development costs are amortized
    - Resulting code is shorter, simpler, more modular, easier to understand, modify and extend
    - Improvements to one program can retroactively improve another
- *Minus:*
    - Increased abstraction may be hard to understand for some programmers
      Requires greater mathematical sophistication
    - Resulting code may be hard to optimize for some compilers
      Requires inter-procedural analysis - Inlining is not enough
    - Questions of version control and source code management

Languages for Reuse

- Pick an implementation language that supports the reuse you want to do
- Languages support different kinds of reuse to different degrees:

Language	Class	Function Protocol	Control Flow	Syntactic Abstract.	Popularity
C++	Y	Y/N	N	N	High
CLOS	Y	Y	Y	Y	Medium
Eiffel	Y	Y	Y/N	N	Low
Modula-3	Y	Y	N	N	Low
Sather	Y	Y	Y	N	Low
Smalltalk	Y	Y/N	Y	Y	Medium

- Features that are missing in some languages:
  - Multiple inheritance, multi-methods, method combination, closures, powerful macros

Finding

- **Finding is easy once the spirit of reuse is instilled**

  If you build it, they will find and use it!

- **Classes can be browsed through the hierarchy**
- **Non-hierarchical objects are harder to browse**
- **Indexing and retrieval technology works well**
  - **Vector model is far better than boolean**

    The more keywords the better

  - **Choosing good names is very important**

    Languages with packages or modules make this easier

  - **Index documentation and other text as well as code**
  - **Glossary/thesaurus can be helpful**

Publishing, Seeking, Finding and Paying

- *Publishing:* Making the existence of a data base known
  - Mosaic/WWW/WAIS/Gopher works for public domain information
  - Mosaic would be a good choice for enterprise-wide data base
  - Establish in-house incentives for publishing code
- *Seeking:* Locate plausible data bases where information might be
  - Just beginning to be an issue -- standards are emerging
  - There is a need for indexing, retrieval and visualization
- *Finding:* Locate information within a single data base
  - Many commercial indexing and retrieval systems available
  - Many graphical browsers available
- *Paying:* Exchanging some currency for information
  - The missing link in the information superhighway/supermarket
  - Even if the cost is free, want to keep statistics on what is useful

Conclusions

- **Reuse cannot be tacked on. It permeates all design and coding.**
  - **Efforts to reuse as an afterthought have failed.**
- **Choose the level of reuse you want:**
  - **Instructional, Informal, Verbatim, Extensible, Stratified**
- **Choose what you want to reuse:**
  - **Classes, Function Protocols, Control Flow Abstractions, Languages**
- **Make sure your programmers think in terms of reuse**
  - **Never do something twice if you can do it once**
  - **Analyze your own code and discover common abstractions**
  - **Combine existing components to build new code**
- **Support Finding:**
  - **Browsers, Naming Conventions, Indexing and Retrieval tools**
  - **Infrastructure for publishing, seeking, finding and paying**