Lecture 9: Software Re-Use

Kenneth M. Anderson

Software Methods and Tools

CSCI 3308 - Fall Semester, 2003

Today's Lecture

- Software Reuse
  - Types of Reuse
  - Pros and Cons
- Introduction to Unix Libraries
- Brooks' Corner: Second System Effect
- But first…
  - Conceptual Integrity and System Architects at Bell Labs
    - Architects are not associated with development groups; they must "pitch" their designs to groups and get them "adopted"

Software Reuse

- Software Reuse involves
  - the use of some previously constructed software artifact
    - source code, library, component
    - requirements and design documents
      - e.g. design patterns
  - in a new context or development project

Types of Software Reuse

- In re-using code, we have several levels of granularity
  - single lines of code
  - functions/procedures
  - modules
  - components
  - packages
  - subsystems
  - entire programs

Other Types of Software Reuse

- Reuse can also include
  - Requirements Documents
  - Design Documents
  - Design Patterns
  - Software Architectures
    - a set of connected components at a high level of abstraction
- This type of reuse can be more powerful…why?

No Silver Bullet, revisited

- Producing requirements and design information is hard
  - We struggle with the essential difficulties of software when we create these documents
- Coding is relatively easy, in comparison
  - Typical projects spend 25% of their time coding, 75% on requirements, design, and debugging
    - In fact, Brooks estimates that most projects spend at least 50% of their time debugging!
- Reuse can help to address these problems
  - Reused reqs./design/code require less debugging

Pros of Software Reuse

- Efficiency
  - Reduces time spent designing or coding
- Standardization
  - Reuse of UI widgets in MacOS and Win32 leads to common “look-and-feel” between applications
- Debugging
  - Reused design/code is often tested design/code
- Profit!
  - Reuse can lead to a market for component software
    - real-world examples: ActiveX components, Hypercard stacks, Java packages, even software tools, such as xerces and xalan from xml.apache.org (they are often included in other software systems)

Cons of Software Reuse

- Mismatch
  - Reused Reqs. and/or Design may not completely match your situation
    - Requires time/effort to convert
  - Non-functional characteristics of code may not match your situation
    - Consider a database that can scale to 10,000s of items, but you need it to scale to 100,000s of items
- Expense
  - Some components may be too expensive for your project’s budget. For instance, SGML (a precursor to HTML and XML) tools sell for 5000 dollars a license!

One example of code reuse

- Generic functions

```
multiply (float A[I][J], float B[J][K])
    for i = 0 to I do
        for k = 0 to K do
            sum = 0
            for j = 0 to J do
                sum += A[i][j] * B[j][k]
            endfor
        endfor
    endfor
```

- In fact, math libraries are an example of a successfully reused software component

Reuse in a Unix Environment

- Two commonly reused software objects in Unix environments
  - source code
  - object code
- Source code is translated to object code via a compiler
- Object code consists of machine instructions with place holders

Object Code Example

```
MOV register4, (some memory location)
SUM register4, register3
CMP register3, #7
JMP (some program address)
```

- .o files contain object code
- Place holders are removed and replaced with specific values when object code is linked into a program executable

Libraries

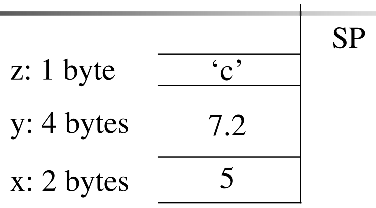
- Unix Libraries are one technique for reusing object code
- Libraries depend on marshalling
  - a uniform method of packing and unpacking parameters

Marshalling example

- When a program calls a function, its parameters are pushed onto the program stack

```
function foo(int x, float y, char z) {
   …
}
foo(5, 7.2, 'c')
```

Marshalling example, continued



When linking the object code, the linker substitutes (SP-1) for z, (SP-5) for y, and (SP-7) for x. This works as long as both the calling program and the called function follow the same rules. This is what marshalling specifies and it allows libraries to be reused on Unix systems.

Marshalling, continued

- In order to use a library, a developer needs
  - a header file (.h) that indicates the parameters of each function contained in the library
  - the object code of the library
- A compiler can then link the object code of the library into a developer's program using the rules of marshalling…
- …and a developer's program can then use the functions contained in that library

Next Lecture

- We will learn how to create libraries with the ar ("archive") command
- We will learn about command flags of the C and C++ compilers that allow libraries to be re-used in new programs

Brooks' Corner: The Second-System Effect

- An engineer is careful in designing a system the first time
  - He or she realizes that they are working in uncharted territory
  - Extraneous features get delayed until…
- The Second System!
  - Now, you've got some experience and you want to throw everything into the design!

Symptoms of Second-System Effect

- Functional Embellishment
  - to an unnecessary degree
- Optimizations to obsolete functionality
  - OS/360 linker had sophisticated program overlay functionality
  - The problem: the application architecture no longer depended on overlays!
    - Side-effect: linker is slower than it otherwise would have been; slower than the compilers whose usage it was meant to reduce!

How to avoid it?

- Must employ extra self-discipline
  - avoid functional ornamentation
  - be aware of changes in assumptions
  - strive for conceptual integrity
- How do manager's avoid it?
  - Insist on a senior architect with more than two systems under his or her belt
    - If this is not possible, then help your architect avoid the second system effect: e.g. question unnecessary features, review assumptions, etc.