CSC 408F/CSC2105F Lecture Notes

Reading Assignment

van Vliet Chapter 17

Software Reuse Concepts

- Create new software by **reusing** pieces of existing software rather than creating new software from scratch.
- Reuse can be done at different knowledge levels
  - Reuse requirement specifications
  - Reuse application architecture
  - Reuse design
  - Reuse source code (*components*)
  - Reuse run time libraries
  - Reuse documentation
  - Reuse tests
  - Reuse process elements
- The goal of software reuse is to reduce the cost of software production (and improve productivity) by replacing creation with recycling.

Software Reuse is a Critical Issue

- *Major* economic and productivity gains from reusing rather than reinventing
- To remain economically viable, all modern software producers must increase percentage of software derived from reusable components
- Example of successful reuse
  - High-level programming languages (e.g., Java, SQL)
  - Library of generic (parameterized) components (e.g. Math library)
  - Parser-generators and application generators (e.g. YACC, JavaCC, ANTLR, automake, Eclipse)
  - Menu/table driven mechanism for specifying parameters (e.g. GUI widgets)
  - Application frameworks (e.g. Smalltalk, Motif, Swing/SWT)
  - Aspects: Pointcuts and advices (e.g. AspectJ etc.)

- Internationalization/Localization (i18n/ l10n)
  (e.g. tag transformations)
- Document generations
  (e.g. Javadoc/XDoclet, DocBook, LaTeX, CSS, RSS, XSLT)
- Components-off-the-shelf (COTS) using middleware
  (e.g., OLE/ActiveX, CORBA, Web Services)
- Plugin-ins, Skins, Themes, Macros, Extensions
  (e.g. Eclipse, Word, WinAmp, Mozilla Firefox etc.)
- Domain engineering and application generation
  (e.g. SAP)
- Domain-specific languages (DSL) and transformation systems
  (e.g. Draco, TXL)
- 4-G languages
  (e.g. SQL, Wizards, templates, MIL/ADL, etc.)

Over 90% of source code in new applications is reuse code

Dimensions of Software Reuse

substance components, concepts, procedures (i.e. process elements), advices
What type of thing is being reused?

scope horizontal or vertical
horizontal – generic components usable in many domains
vertical – components apply to a specific application domain
mixed – aspects apply to a similar application domain

approach planned, systematic or *ad hoc*, opportunistic
planned & systematic – build reusable components, then develop software
ad hoc, opportunistic – reuse a component if it happens to be suitable for an new project (e.g. a math library)

Taxonomy of Software Reuse

`-- C. Krueger. "Software Reuse". ACM Computer Survey, 1992.`

classifications A library with index

abstraction Make the substance understandable

selection Search library for reusable substances

specialization Adapt the selected substance to new context

integration Compose the specialized substance with other substances

technique compositional or generative
compositional – build software through the composition or reusable components.
generative – build software through reuse of knowledge and software *generating* tools

usage black-box (as is) or white-box, modified
black box – reuse component as is (e.g. **C**omercial **O**ff **T**he **S**helf software)
white box – modify the component to customize it for the new application

product source code, object, design, text

Substance Libraries

- *Concept* – establish a standard library of reusable substances and do reuse by selecting substances from this library.
- This concept is easy to implement for domains that
  - **Good Classification, Easy Selection**: Have a well-developed field with a standardized terminology.
  - **Good Abstraction, Easy Adaptation**: E.g. Components have small simple interfaces and a standardized data format.
  - **Easy Integration**: E.g. Blackbox (library linking or web service binding), White-box (aspect weaving)
- Examples: Scientific (Math) function library, string handling package, network interface library
- Needs good classifications (index): keywords, taxonomy, facets
- Needs common understanding: ontology

Library Issues

Searching (Selection) Need effective ways to find a component in a large database of possible components. May need to provide a variety of search criteria[a]

- Programming Language? [how?]
- Capacity or Speed ? [why?]
- Overall quality? How well tested? [how much?]
- Reused by others? [where?]
- Author? [who?]
- Algorithm or data structure used. [what?]
- . . .

Understanding Need documentation that provides a precise and complete description of what the component does.

[a] See van Vliet Figure 17.2 for a typical taxonomy of reusable components

Adaption (Specialization) If the component doesn't do *exactly* what we need done, how can we adapt it?

Want to avoid/minimize reprogramming since that defeats the purpose of reuse.

Composition (Integration) What mechanisms exist for producing the *glue code* that ties the components together?

Indexing (Classification) The method chosen to index the library will have a strong influence on how effectively the library can be used.

Indexing can be based on keywords extraction (e.g. a KWIC index, facets) or on a preconceived index hierarchy (e.g. a library book index)

Small well-focused reuse libraries have proven to be more successful than very large libraries.

Granularity Large components provide more reuse functionality.

Small components are easier to reuse. [architectural reuse $>$ component reuse $>$ code scavenging]

Evaluating Components for Suitability

Component Quality

- Rating based on ISO 9126
- Reuse history (used successfully by others)
- Comments and feedback from other reusers

Administrative Information

- Author
- Modification history
- Cost (if any)
- Support status

Documentation

- Detailed internal documentation
- Adaptability features
- Algorithms and data structures used
- Original design documentation

Interface Information

- Description of the programming interface to the component.
- Function names, parameter descriptions, exceptions.

Testing Information

- Description of how to test the component.
- Library of test cases for regression testing.

Software Components Templates

- Create library of *generic, parameterized* software components
  Each component is a *template* that can be expanded with different parameters to fit a given reuse. The `template` mechanism in C++ is an example of how this might be implemented.
- An extension of techniques for building portable software
  But, requires even more care in parameterization and adding hooks to allow customization and adaptation
- Initial creation of component templates is more expensive than creating single use components. Initial effort must be justified by gains from reuse

Component Template Example

```
module $STACKNAME=Stack$
    export( $PUSHNAME=Push$, $POPNAME=Pop$, $TOPNAME=Top$,
            $EMPTYNAME=Empty$, opaque $STACKTYPENAME=StackType$,
            $CREATENAME=Create$, $DESTROYNAME=Destroy$ )
    type  StackRange : $STACKLOW=0$ .. $STACKHIGH=100$
    $assert( STACKLOW <= STACKHIGH )$
    $assert( STACKLOW > minInt & STACKHIGH < maxInt )$
    type  StackIndex : STACKLOW - 1 .. STACKHIGH + 1
    type  STACKTYPENAME :
          record
             stack: array  StackRange of $STACKELEMENT=int$
             index: StackIndex
          end record

    procedure $PUSHNAME$( var stk : $STACKTYPENAME$ ,
                val : $STACKELEMENT$ )
         $PREPUSH=$
         ...   /* Push happens here */
         $POSTPUSH=$
    end  $PUSHNAME$
    ...
end $STACKNAME$
```


Other Reuse Strategies

Design Reuse Build partial programs that incorporate generic processing models. Complete the program to make it specific to the needs of a given project.

Architecture Reuse Reuse well-proven architectural solutions in a given application domain. Example: compilers
Not as widely used since for many areas, there isn't a dominant and accepted architecture.

Transformation Systems Describe the software (e.g. it's logical structure) in some high level notation. Use an application generator to transform the description into a concrete program in some language.

Application Generators Build a software tool that generates programs for a particular application domain based on a small menu of parameters. Example: payroll processing

Stages of Software Reuse

No reuse All software developed from scratch.

Ad hoc opportunistic reuse driven by circumstances.

Structured reuse Existing components reused in an organized way for new software.

Institutionalized Software reuse is the primary method for building all software. Automated searching of large reuse libraries. Programmers work at the MIL level.

Non-Technical Reuse Issues

- Software developers prefer to invent rather than reuse.
  A successful reuse process has to motivate and encourage developers to reuse software.
- Management support is crucial for successful reuse.
  - Motivating developers to reuse
  - Accepting the cost/time required to build reuse components.
  - Establish a reuse support structure. Reuse librarian.
    Reuse library building, testing, documentation and indexing.
  - Recognize the *library problem* - some reusable components may never actually be reused. This is one of the costs of doing reuse.
- Need to perform domain analysis to identify potential areas that would benefit from reuse.
- Need to establish software architectural and interface standards to facilitate component reuse.

- Software reuse costs
  - Incremental cost of developing reusable components instead of non-reusable components. Can be 50% ... 100% more.
  - Direct and indirect costs of maintaining a reuse library.
  - Cost of adapting a component for use in a particular system.
- Need to justify the benefits from reuse to (non-technical) management
  - Each actual reuse reduces the cost of a new system.
  - Reuse (greatly) increases programmer productivity.
  - Reuse makes it easier to understand the software due to commonality effects.
- Reuse can help to reduce long term software maintenance costs
  - Track (log) where reuse components are actually used.
  - Any error discovered in a reuse component can be communicated to the maintainers of all systems that use the component.
  - One repair to the error can be propagated to all reusing systems.

Reusing Non-Functional Requirements

- Early Requirements are the most abstract substances in software lifecyle
- Goal model is suitable to express the early requirements
  - Functional requirements use hard goals
  - Non-Functional requirements use soft goals or quality attributes
- How to make them reusable?
  Classify? Abstract? Select? Specialize? Integrate?
- Example. Design speedcar to reuse the streamline requirements from animals.

How to classify requirements?

Through the magic of evolution, the fastest swimming animals, such as a bluefin tuna Thunnus thynnus, a porbeagle Lamna nasus, a Heaviside's dolphin Cephalorhynchus heavisidii and an ichthyosaur Stenopterygius quadriscissus, all share a common streamline shape.

R. Motani, "Scaling effects in caudal fin propulsion and the speed of ichthyosaurs". Nature 415(Jan. 17):309-312, 2002.

- Build for reuse

  In order to have speed, you can always follow the streamline aspect of a Tuna fish. Streamline is a kind of shape that typically has three parts at the surface. Concerning the diameters, the three parts include a widening head, a constant body and narrowing tail. The three parts are connected smoothly such that a streamlined fish not only swims faster, but also looks more beautiful.

- Classification

Who	What	Why	Where/Which	When	How	How much
Tuna fish	Shape	Speed	surface	always	∼	++speed, +complexity +beauty
Tuna fish	Shape	∼	head	always	∧	(+complexity)
Tuna fish	Shape	∼	body	always	O	
Tuna fish	Shape	∼	tail	always	∨	(+complexity)

- Abstraction:

```
Speed for Shape at Surface {
  Streamline for Surface at Head, Body, Tail {
    A for Head => +Complexity
  and    O for Body
  and    v for Tail => +Complexity
  }  => +Complexity, +Beauty
 }  => +Complexity, +Beauty
```


- Build with reuse

  In order to have speed for my car, I will mimic the streamline aspect of a Tuna fish. Streamline is a kind of shape that typically has three parts at the surface, which in my case, a surface to air instead to water. Concerning the diameters, the three parts include a widening front, a constant body and narrowing trunk. The three parts are connected smoothly such that a streamline car no only drive faster, but also looks more beautiful. The building of a streamline car will, however, require a more complex construction for the front and the trunk.

```
Speed for Shape at Surface {
  Streamline for Surface at Front, Body, Trunk {
    A for Front => +Complexity
  and    O for Body
  and    v for Trunk => +Complexity
  }  => +Complexity, +Beauty
 }  => +Complexity, +Beauty
```

[**Can you reuse streamline for software performance?**]

Programming in the LARGE

Module Interconnection Languages

- Module Interconnection Languages (MILs) are used to describe the relationship between *modules* in a software system[a]
- Programming with modules as the atomic unit of composition is called *Programming in the Large*.
- MILs help the software developer
  - Describe a system architecture at a higher level of abstraction.
  - Implement static type checking between different modules.
    i.e. make sure module interfaces are used correctly.
  - Describe the binding of modules together in one document.
  - Keep track of module versions as a system evolves.
- MILs are strongly related to the *System Models* that are used in Software Configuration Management.

[a] See van Vliet Figure 17.7 for an example

Module Interconnection Languages

- In a general model of a software system
  - The system is composed of some arbitrary collection of modules.
  - Some modules provide services (*servers*) and other modules (*clients*) use those services.
  - Most modules are both clients and servers
  - The connection between servers and clients is defined by a collection of *interfaces*.
  - An interface describes all of the *resources* that a server module make available to its clients.
  - A resource might be a data type, a constant, a variable, a function or a module.
- A MIL is used to describe all of the
  client $\rightarrow$ interface $\rightarrow$ server
  relationships in a software system.

- A module interconnection language must be rich enough to describe all of the following situations
  - One module might implement several interfaces.
  - One interface might be implemented by a cooperating collection of modules.
  - Different clients might be associated with different servers *for the same interface*
  - There might be arbitrary restrictions on which clients can use a given interface.
  - The relationship between clients, interfaces and servers is continually changing as a software system evolves.
- *Software Architecture Description Languages* evolved from MILs to provide more detailed and precise software system descriptions.
- Makefiles are a (poor) example of a Module Interconnection Language
  Module Interconnection Languages for real systems provide much more sophisticated tools for specifying the relationships between clients, interfaces and servers

Reuse through Web Services

- Web services are good for reuse
  - It has large granualrity
  - UDDI is a *library* of web services
  - WSDL is an *abstraction* for web service
  - WS-Policy is an *selection* for web service
  - SOAP is an *specialization* for a web service
  - WS-I and BPEL are for a web services *integration*
  - ...
- Wrapping up components into Web services
  - Define interfaces
  - Define protocols
  - Define data/control flow
  - ...