Lecture 8

Software Reuse

Don't reinvent the wheel,
Do something smart

Last lecture and tutorial …

Aspect-orientation

- We explained the concept of aspect orientation: separation of crosscutting concerns
- In programming, aspects modularizes scattered joinpoints in the code
- It is not only programming, you can separate concerns scattered in design, requirements specifications, goals as long as crosscutting happens to them

Today …

On Software Reuse

1. Software reuse principles
   1. Why reuse?
   2. Elements of software reuse
   3. Classic examples of software reuse
2. Software reuse in new practice
   1. Component-based software reuse
      1. Web service-oriented architecture (SOA)
      2. WSDL, Semantics Web and BPEL
   2. Quality-based software reuse
      1. Non-functional requirements and quality attributes
      2. Advices can be implemented through aspect orientation
      3. Q7, a language for the quality-based aspect oriented reuse
3. Summary

1. Software reuse principles

- Hardware reuse
  use the same tool more than once, producing the same product more than once, etc.
  Hammer a nail
  Hammer a nail again
  Hammer a nail again and again

- Software reuse: ***don't reinvent the wheel***
  use the same knowledge more than once
  Hammer a nail
  Hammer a nut
  Hit an object with a force, *Newton's discovery …*

Create new software by reusing pieces of existing software rather than creating new software from scratch.

1.1 Why Reuse?

- *Save the cost, Reduce the effort*
  Software costs huge when it was created, but costs almost nothing to copy or redistribute
  One should focus on more creative tasks
- *Reduce bugs*
  Use proven legacy software rather than write it completely from scratch

The goal of software reuse is to reduce the cost of software production by replacing creation with recycling.

1.2 What hampers software reuse?

Common problems make the reuse difficult

- Identify units of reusable knowledge
- Store the reusable knowledge into a “knowledge base”
- Search the reusable knowledge for your target
- Modify the reusable knowledge to fit your new situations
- Combine the reusable knowledge with your project

R. Prieto-Diaz. *Status Report: Software Reusability*. IEEE Software. 10(3): 61-66, 1993.

1.2 What hampers software reuse?
Improve Software Reusability

Build for reuse

- Identify units of reusable knowledge
- Store the reusable knowledge into a “knowledge base”

Build with reuse

- Search the reusable knowledge
- Modify the reusable knowledge to fit new situations
- Combine the reusable knowledge with your project

1.3 Five dimensions of good SR

Build for reuse

- **Abstraction**: Identify units of reusable knowledge and concisely represent them in abstract form
- **Classification**: Store the reusable knowledge into a “knowledge base” that is indexed and classified

Build with reuse

- **Selection**: Query the reusable knowledge into parameterized form (e.g. function with formal parameters)
- **Specialization:** Modify the reusable knowledge to fit new situations (e.g. function with actual parameters)
- **Integration:** Combine the reusable knowledge with your project (e.g. invocation, weaving, etc.)

[Krueger92] Software Reuse, ACM Survey. 1992

1.3 Five dimensions of successful SR

Classic software reuse examples

- 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) through 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

1.3 Classic software reuse example 1

High-level programming languages

- Imagine the difficulty (complexity) in writing matrix multiplication in machine code, or assembly. In APL, all you need is one line!
- The level of abstraction is important!
  C < Fortran < C++/Java < Python < SQL (4GL)
- The efficiency is another issue, but we have *compilers*, HLPL increase the productivity of programming by 10x!
- Even better, the compiler-generators can reduce the efforts of writing a new compiler
- Programming libraries support still higher level of abstraction

1.3 Classic software reuse example 2

Transformation systems

- Even better, the ***compiler-generators*** can reduce the effort of writing a new compiler
- In transforming systems, the ***semantics*** of the artifacts are defined through transformations and refinements
- Once a transformation is defined, it can be applied to many semantics mappings
- This is still an active SE area in *domain-specific languages*, *generative programming*
- A new trend is ***document-driven programming***, i.e. consider programs as data to be processed by other programs.
  For example, XSLT is XML transformation, while itself is also an XML document (to be processed by XSLT).
  You can write a localizing stylesheet to convert English markup into Chinese, while the stylesheet itself can be transformed as well…

2. New practice of software reuse

Where is the next 10x productivity breakthrough …

Let's take a tour on component-based and quality-based software reuse.
We must keep the following SR criteria in mind:

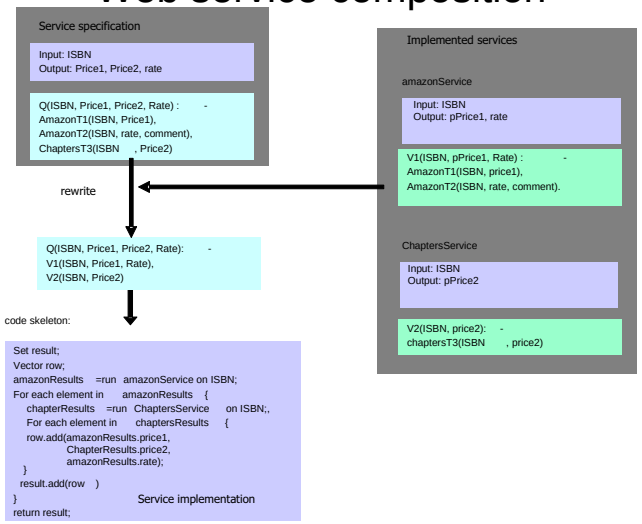
- **Abstraction**
- **Classification**
- **Selection**
- **Specialization**
- **Integration**

2.1 Component-based SR (COTS)

COTS = Component-off-the-shelf, shrink-wrap software

- Components are modules with high intra-component cohesion and low inter-component coupling (*modularizing*)
- Components hide implementation details and only expose abstract declarations (*information hiding*)
- Components can be specified through interface definitions, such as *MIL, IDL, ADL*, WSDL (*abstraction*)
- Components can be indexed in program libraries, such as Windows registries, Linux *RPMs*, sourceforge, UDDI *(classification)*
- Components communicate through standardized protocols, such as DCOM,CORBA/RPC,JavaRMI,SOAP *(selection)*
- Components can be tuned to perform specialized tasks, such as WS-policy (*specialization*)
- Components can be composed to perform complex tasks, using for example, *Shared libraries, WSFL/BPEL* (*integration*)

Web service composition



Consideration for SR

- Abstraction: Use *WSDL+Datalog+SQL* to formally describe the syntax + semantics + pragmatics of a web service interface (c.f. less abstract WSDL+OWL-S+BPEL approach)
- Classification: UDDI web service for the query, e.g. *xmethods*
- Selection: *query rewriting* to convert the composite web service into constituent ones
- Specialization: passing parameters through SOAP messages
- Integration: using the web services as user-defined functions in SQL (DB2)

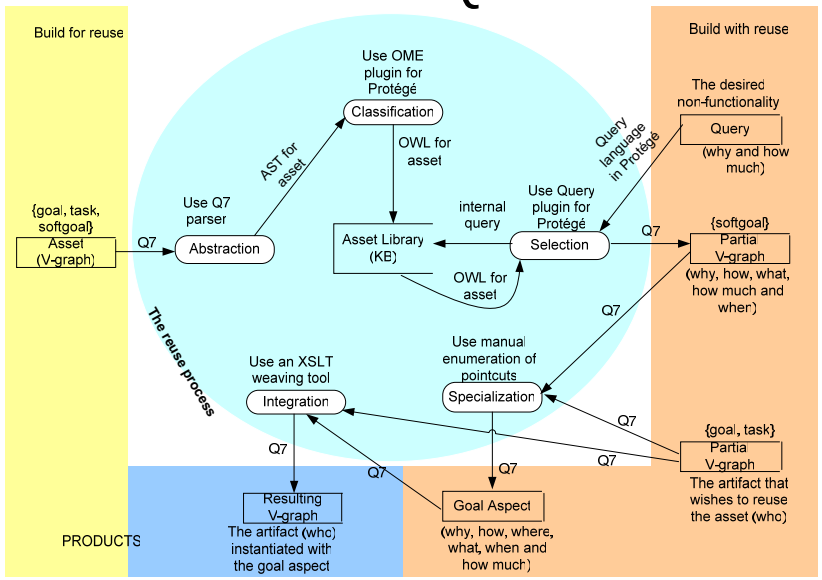
[WSC] J. Lu, Y. Yu, J. Mylopoulos. “A lightweight approach to web service sythesis”. WIRI, 2005.

2.2 Quality-based SR

- Most existing literature focuses SR on functionalities, as represented by component-based reuses
- Quality-based SR takes a new perspective on non-functionalities, as they are “tangled” with functionalities, one needs to separate them from the components to make it reusable assets
- Aspect-oriented SR aiming at just that!

[QBSR] J.C.S.P. Leite, Y. Yu, L. Liu, E.S.K. Yu, J. Mylopoulos. *Quality-based Software Reuse*. CAiSE. 2005.

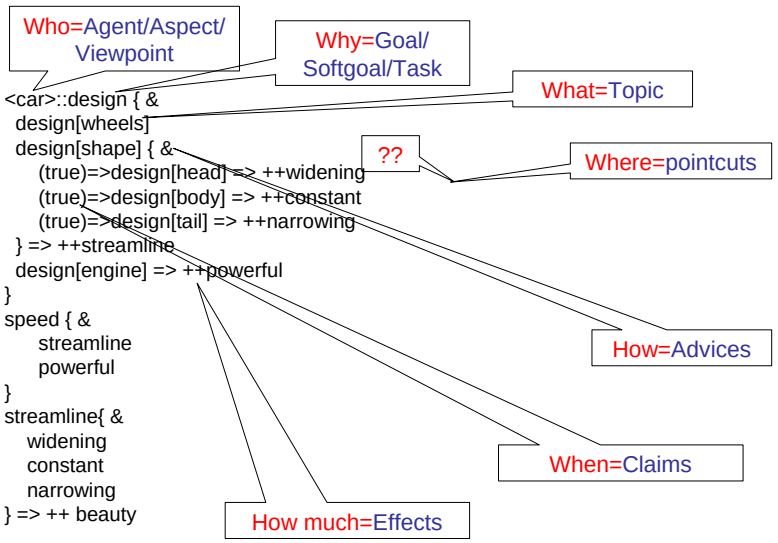
Towards QBSR



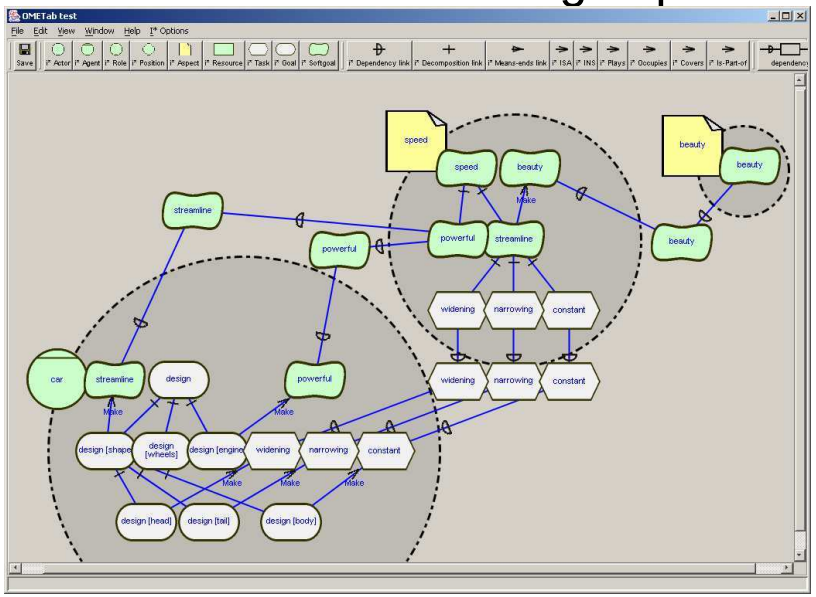
Abstraction: the Q7 language

- Q7 = 7 questions, 5W2H: (When, Who, Why, What, Where, How, How much)
- 5W2H is the core idea for the Quality Movements (adopted by the Japanese car industry)
- Q7 are useful to elicit and represent knowledge for quality attributes
- The idea of object-oriented (what), goal-oriented (why), agent oriented (who), aspect-oriented (where), testing-oriented (when), non-functional requirements framework (how much) all root deeply in the Q7 language

2.2.1 Q7 language for quality reuse



Classification: introducing aspects

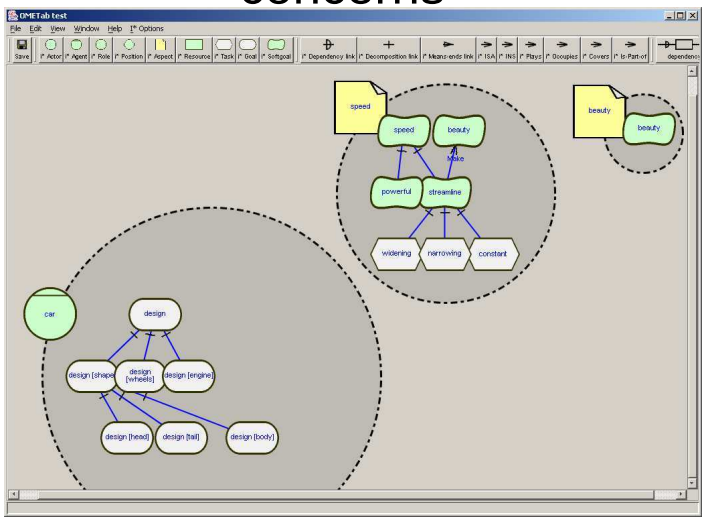


Where are the aspects?

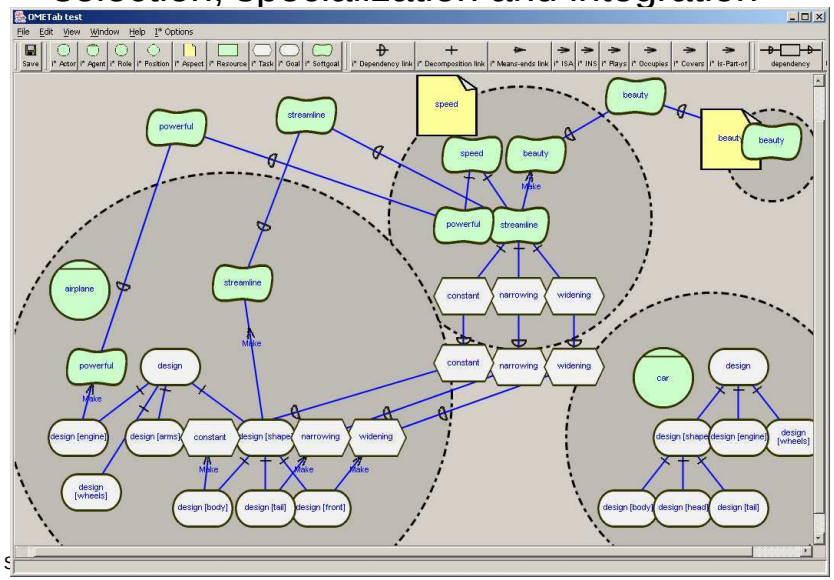
```
<car>::design { &
  design[wheels]
  design[shape] { &
      (true)=>design[head] => ++widening
      (true)=>design[body] => ++constant
      (true)=>design[tail] => ++narrowing
  } => ++streamline
  design[engine] => ++powerful
}
speed { &
      streamline
      powerful
}
streamline{ &
    widening
    constant
    narrowing
} => ++ beauty
```

```
<car>::design { &
  design[wheels]
  design[shape] { &
        (true)=> design[head]
        (true)=> design[body]
        (true)=> design[tail]
  }
  design[engine]
}
<speed>::speed { &
      streamline<=++*[shape]
      powerful<=++*[engine]
}
<beauty>:: beauty {&
      streamline<=++*[shape]
}
streamline { &
    widening <=++*[head]
    constant  <=++*[body]
    narrowing<=++*[tail]
}
```


Separation of crosscutting concerns



Build with reuse: selection, specialization and integration



2.2.3 Linking Q7 to your code

```
/* @purpose SendEmail */
void send_email () {
      /* @purpose ComposeEmail */
      Document d = compose_body();
      Address a = get_email_address();
      /* @purpose SendOut */
      send_out(a, d);
}
```

A JAVA PROGRAM

```
SendEmail { &
  /* void send_email () { */
        ComposeEmail
        /* Document d = compose_body();
        Address a = get_email_address(); */
        SendOut
       /* send_out(a, d); */
}
```

A Q7 “PROGRAM”

3. Your exercise

- Identify reusable parts from a legacy system
- If you would build for reuse, what would you do for the web service module? Imagine a scenario where your web service can be reused by some teams' client programs.
- If you would build with reuse, what would you do for the graph editor client module? Imagine a scenario where your client program can reuse some teams' web service modules.
- Use Q7 to categorize your non-functional requirements and reuse some of them through aspects

4. Summary

- Reuse and Reusability
- How to improve reusability
  build-for-reuse versus *build-with-reuse*
- Example of how to reuse through components
  web service-oriented software reuse
- Example of how to reuse through aspects
  quality-based software reuse

Further readings

[SR] Krueger. "Software Reuse". *ACM Survey*. 1992.

[SReusability] R. Prieto-Diaz. *Status Report: Software Reusability*. IEEE Software. 10(3): 61-66, 1993.

[NFR] L. Chung, E. Yu, J. Mylopoulos. *The non-functional requirements framework. 1999.*

[ReusableAspects] *S. Clarke, R. J. Walker. Composition patterns: An approach to Designing Reusable Aspects*. ICSE. 2001

[WSC] J. Lu, Y. Yu, J. Mylopoulos. "A lightweight approach to web service sythesis". WIRI, 2005.

[QBSR] J.C.S.P. Leite, Y. Yu, L. Liu, E.S.K. Yu, J. Mylopoulos. *Quality-based Software Reuse*. CAiSE. 2005.

What's next …

- A tutorial on componentization and Web service composition
- How to deploy web services on the Tomcat web server