Component-Based Design

What is a Component?
Defining Reuse
Obstacles to reuse
Exercise Questions

Lecture 1, Thursday 31 August 2006

After this lecture …

- You should be able to explain what a component is, and how the idea of reuse and components have evolved over time

Software components: an old dream

Doug McIlroy (1968): *Mass produced software components:*

”We undoubtedly produce software by backward techniques. [...] Components, dignified as a hardware field, is unknown as a legitimate branch of software.”

What is the purpose of software components?

The main purpose is software **re-use**:

- Don’t build a software system from scratch, build it from existing components.
- Re-use this project’s components in the next project.
- Or sell components to others who need them.
- There may arise an industry – or community – of component suppliers, with catalogs (mail-order style) of ready-made components.
- This vision has been realized in numeric computing (netlib, Numerical Recipes, Colt) or operating systems (POSIX) but not in general.

What is a software component?

A decade ago, component-based development became fashionable.

At the ECOOP conference in 1996, a software component was characterized as:

- a unit of composition with
- contractually specified interfaces and
- explicit context dependencies only.

Historical support for re-use

- Source code re-use: copy-paste-edit.
  Very bad for maintenance; duplication of bugs
- Function libraries (POSIX).
  Limits re-use to functions, not data structures.
- Object-oriented programming.
  Supports re-use of both functions and data structures. But inheritance does not support cross-organizational re-use of base classes well.
- Software components.
  But how specify the interfaces?
- Component standards (CORBA, COM).
  Too complicated, or limited to one platform.
- Enterprise Java Beans.
  Several notions of component – and the focus of this course.
- Microsoft .Net.
  Several notions of component – and we'll touch upon it.

Components in the Java world

	Component	Lectures in this course	Use
J2SE	Applet	(none)	Client-side user interfaces and small applications
	JavaBeans	(none)	Client-side visual user-interface design support
J2EE	Servlets, JSP, Taglibs, JSF	3, 4, 5	Server-side support for client-side user interfaces
	Enterprise Java Beans	6, 7, 8	Container-integrated services
	Application client components	(none)	

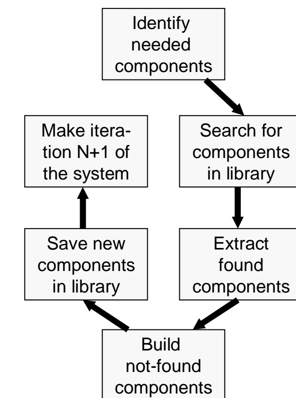
How do components arise?

- Components may be created by encapsulating existing code providing well specified functionality.
- Such components may be written in any programming language, for example Fortran or Cobol.
- New components are typically created in
  1. Microsoft® .NET
  2. Microsoft Component Object Model™ (COM) components
  3. Java™ 2 Platform Enterprise Edition (J2EE)
  4. JavaBeans™
  5. Borland Delphi
- The most widely used are .NET/COM and J2EE/Java-based components as there are vast numbers of programmers who program in each of these environments.

Component Size

- Components may be small with a limited functionality or large and complex with extensive functionalities – They have different granularity
- A fine-grained component example: Calculate the sum of 3 numbers.
- A coarse-grained component will provide a large piece of functionality. Example: An eCommerce component that provide 80% or 90% of the functionality of a full eCommerce system including registration, shopping cart, catalog services, invoicing, emailing, etc.
- The eCommerce component is in fact a framework of components that are integrated together to provide a coarse-grained solution.

Component Based Development



Properties of a good Component

- Well-specified service
- Encapsulation (hiding internal structure)
- Well-defined interface
- Unique Identification
- Known Quality

+ Organizational requirements

Well specified service

- The component must provide a well specified service with clearly defined boundaries
- If the service of a component is not sufficiently general, simple source code reuse may be a more appropriate alternative
- It is also important that the functional abilities are in demand, as the effort spent on preparing the component for reuse is wasted if they are not
- There is a difficult balance between predicting the required functionality and deliver it when needed, and providing the functionality that is actually in demand. Striking the balance requires experience, taste and luck.

Encapsulation

- An important property of a component is that internal structure, algorithms and data are hidden from the user.
- Important requirement that the component can be used in other projects without any knowledge of the internal working of the component.
- If this is not the case, maintenance of the component will suffer and/or become very expensive.

Well defined Interface

- The interface to the component must be clearly defined and well described.
- This includes both how the component interfaces to the system which it is part of, and how the end user applies the services offered by the component.
- The encapsulation requirement also implies that the data structures and methods of manipulating the data must be clearly defined, as well as any infrastructure requirements, e.g. a requirement for access to a specific database tool.

Unique Identification

- The component must be provided with a precise identification and appropriated labels. The identification must make it possible to distinguish different versions of a component without any doubt. Without an appropriate identification of a component maintenance of systems built from that component will become very difficult.
- If specific conditions for the usage of the component apply, this information must also be provided as a label to the component.

Known Quality

- The quality of the component must be well documented
- Expected that a component has a higher quality than code produced for single use
- Example: Component comes with a log of proposed, accepted and rejected change requests AND documentation of the quality found during testing of the component.

Organizational Requirements

- If components are being developed or maintained in-house, a responsibility for each component must be designated, and a support function set up.
- There is also an urgent need for configuration management (e.g. which component is used where in which version and which context)
- And a need for quality assurance of the components, including support for efficient regression testing

The Market for Components

- There are several places on the web where components can be purchased, or are distributed as open source.
- There is a component business, but it suffers from some basic problems.
- For example there is no system for classifying components (like books in a library)
- Thus a component with specific attributes can be difficult to find.

  + There are no commonly accepted rules for certifying or evaluating components.

Why Reuse?

Three obvious reasons:

- Time, because it is much faster to take something existing than to build something new
- Cost, because a component that has been built does not resources to be built again
- Quality, because one can assume that the quality of a reusable component increases over time, as it is being used in several places and thus maintained by many people

Source Code Reuse

- Developers have always reused their own source code
- Could be reason why software is unreliable, big, slow and difficult to maintain

The dark side of Code Reuse

- Misunderstandings may lead to wrong interfaces causing difficult debugging.
- Difficult to trace bugs inside reused code
- Reusing others code often results in pulling in code it depends on
- Some of the reused code often duplicates other code with almost similar functionality, resulting in bigger software

Code Libraries

- The first attempts to organize reuse consisted of defining a set of libraries with specific functionality, and then reusing these in various applications.
- This requires an architecture where the functionality of the application and the support functionality within the application is distributed among reusable libraries and/or assigned to the application itself.
- This type of reuse was developed in many companies, perhaps because it is a natural extension of the way a procedural language is constructed.

Object Oriented Reuse

- Object Oriented programming was marketed as a way of achieving software reuse
- OO became a hot topic during the 1990's, even though the concept is from 1967 (Simula)
- OO emphasizes encapsulation => Important for both Components and Reuse
- Expectation that OO would cause reuse to become the state of practice in companies - This has not been the case
- OO has inspired many people to think about reuse – mainly in the form of "Patterns". See for example books on Analysis Patterns (Fowler, 1997) and Design Patterns (Gamma et al., 1995)

Reuse requires Management

- Reuse is typically a long-term investment, which is expected to pay off as a component is being reused in several products
- A project with focus on delivering the product on time may not have the resources required to make the necessary adaptations for reuse, unless Management allow for it
- Management needs to request reuse, provide the necessary resources and follow up on the results
- There may also be cultural problems when an organization is adopting reuse
- For example the power over the product features change from product development to the part of the organization that maintains the reusable components

”Soft” obstacles to Reuse

- Blocked creativity
- Low cost of rework in software (compared to hardware)
- Changing developer roles
- Power shift in organization

Obstacle 1: Blocked Creativity

- For a component to be effectively reusable it has to be stable over time and changes must be strictly controlled. Component reuse => Inflexibility.
- Product development is a creative venture, and human beings are by nature creative. They want to learn and improve and change
- Reuse does not necessarily block creativity. But creativity must be directed at the larger picture, not at the properties of the single, reusable component

Obstacle 2: Little cost of rework

Hardware development

- Cost of rework in production is usually quite high => focus on quality
- Good idea to reuse an existing component of known quality instead of developing a new one of unknown quality

Software development

- A software production run is usually (but not always) relatively cheap, and redoing it is usually not so much a matter of cost as – perhaps – of time

Result

- Higher focus on reuse of hardware than of software
- Exception: Embedded software - because changes in the software may entail large hardware costs

Obstacle 3: Changing Roles

- In a project the software architect typically have had control over the functionality
- But with components and reuse a part of the functionality is controlled outside the product
- It is not as simple to effectuate a change in a reusable component as it is in a component that is developed solely by the project itself
- Similarly, a number of interfaces and standards will be imposed on the project in order to facilitate reuse of components developed outside the project, thus limiting the project's freedom of choice

Obstacle 4: Power Shift

- When an organizational structure supporting reuse is established, part of the responsibility for product development is moved to this part of the organization from the part that used to have full control of product development
- This can be a small change, but at other times it can involve shifts in power that may complicate the process

”Hard” obstacles to Reuse

- Unstable Standards
- Unclear Functionality
- Unclear Quality
- Unclear Method

Obstacle 5: Unstable Standards

- There are many component object standards, e.g. COM, ActiveX, .Net, CORBA, Java Beans and Enterprise Java Beans. These have existed for some time, but they also change and develop extensively.
- These standards do not cover the same needs and have different advantages and disadvantages.
- There is no one globally accepted standard.
- This instability will not change, as it is driven by market competition.
- Different suppliers need to differentiate their product from the others to gain market share.

Obstacle 6: Unclear Functionality

- There is no standardized method for description and classification of the functionality of a software component.
- The same is true for the describing the quality of a software component (and e.g. maintainability and portability).
- It is therefore difficult to provide a commonly recognized description of the required functionality of a component. This complicates the search for a component that fulfils a specific need.
- By contrast, many attributes of hardware components can be described in standard terms (logic, power consumption, sensitivity to electronic noise, …)

Obstacle 7: Unclear Quality

- It is not easy to see what the quality of a component is, esp. if it is purchased. There is no declaration of quality in standardized terms.
- Neither is it easy to discern how the quality of the product one is developing is affected by the quality of a reused component.
- There is no standardized method for certifying the quality (nor the content) of a component. There is an obvious need, but the rules by which such a certification scheme would function are not easily defined

Obstacle 8: Unclear Method

- There are many books in the software literature on reuse of software components
- But **no** clear set of “best practice” guidelines. Thus many organizations have difficulty getting started with reuse

The next buzzword? Service Oriented Architecture

- Google:
  - component based development: 88 900 000 hits
  - service oriented architecture: 52 800 000 hits
- *Service oriented architectures introduce a model for distributed software components* (Francis Curcubera, IBM Research, March 2006)

Technologies used for SOA

- XML, XSD, XSLT (W3C): transmitting, describing and transforming data – the *interface* between services (alias components)
- SOAP (W3C): XML via HTTP
- WSDL (W3C): describing what a service will do and how to talk to it
- BPEL (OASIS): orchestration, describing business workflows

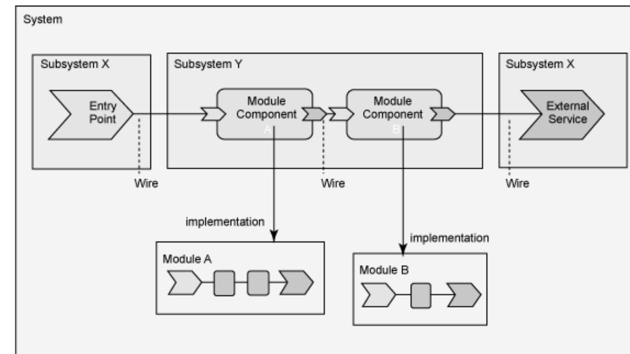
Some uses of SOA

- To package legacy (COBOL, PL/I, …) software so modern software can talk to it
- Example: Danske Bank’s systems
- To allow different vendors to deliver the parts of the same system
- Big challenges:
  - Well-defined interfaces (XML Schemas + semantics)
  - Efficiency (“client-side joins”, Yrk!)

SOA and J2EE?

- SOA with J2EE, C++, PHP, COBOL and more:
  - Service Component Architecture (SCA)
  - Service Data Objects (SDO)
- Open SOA Collaboration (OSOA) is IBM, BEA, Oracle, SAP, Sun and others – but not Microsoft
- Sources:
  - http://www-128.ibm.com/developerworks/library/specification/ws-sca/
  - osoa.org: SCA and SDO standards, EJB Integration, …
  - Apache Tuscany, early open SCA implementation

SCA modules and components



Literature

Main sources for this lecture:

- Olesen, R. (2003). *System Architecture and Component Reuse*. Datateknisk Forum, Report DF-18, November 2003.
- Clemens Szyperski: Component Software, Addison-Wesley 2002, pages 35-48. What is a component?
- Clemens Szyperski: Component Software, Addison-Wesley 2002, pages 302-323. About component concepts in J2EE.