Software Reuse & Component Based Software Engineering

Johan Östlund
johano@dsv.su.se

(much based on Ian Sommerville’s presentations for his book Software Engineering 8)
Software Reuse

- A big selling point of object-orientation was reuse

- As it turns out reusing objects or classes is difficult

- Classes/objects are too detailed and specific

- We either need to accept that we cannot reuse or we need to use something more coarse grained
Component definitions

A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction

(Grady Booch, 1987)
A software component is a static abstraction with plugs

(Nierstrasz and Tsichritzis, 1992 and 1995)
Component definitions

Megaprogramming is a technology for programming with large modules called megamodules that capture the functionality of services provided by large organizations like banks, airline reservation systems and city transportation systems ...

... Megamodules are internally homogenous independently maintained software systems

(Wiederhold et al., 1992)
Component definitions

By components we mean already implemented units that we use to enhance the programming language constructs. These are used during programming and correspond to the components in the building industry

(Jacobson, 1993)
Component definitions

A component is a piece of software small enough to create and maintain, big enough to deploy and support, and with standard interfaces for interoperability

(Harris, 1995)
Component definitions

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties

(Szyperski, 2002)

- Szyperski also requires a component to be:
  - a binary entity
  - language independent
  - stateless
Component definitions

[A component is] a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard

(Council and Heineman, 2001)
Component as a service provider

- The component is an independent, executable entity. It does not have to be compiled before it is used with other components.

- The services offered by a component are made available through an interface and all component interactions take place through that interface.
Component characteristics

- Standardized
  Component standardization means that a component used in a CBSE process has to conform to some standardized component model. This model may define component interfaces, component metadata, documentation, composition and deployment

- Independent
- Composable
- Deployable
- Documented
Component characteristics

❖ Standardized

❖ Independent

~ A component should be independent – it should be possible to compose and deploy it without having to use other specific components. In situations where the component needs externally provided services, these should be explicitly set out in a ‘requires’ interface specification

❖ Composable

❖ Deployable

❖ Documented
Component characteristics

❖ Standardized

❖ Independent

❖ Composable
   ~ For a component to be composable, all external interactions must take place through publicly defined interfaces. In addition, it must provide external access to information about itself such as its methods and attributes

❖ Deployable

❖ Documented
Component characteristics

- Standardized
- Independent
- Composable
- Deployable
  
  To be deployable, a component has to be self-contained and must be able to operate as a stand-alone entity on some component platform that implements the component model. This usually means that the component is a binary component that does not have to be compiled before it is deployed

- Documented
Component characteristics

- Standardized
- Independent
- Composable
- Deployable
- Documented

Components have to be fully documented so that potential users of the component can decide whether or not it meets their needs. The syntax and, ideally, the semantics of all component interfaces have to be specified.
Components vs. objects

- Object-orientation tries to model the world by breaking down the domain into nouns (classes/objects) and verbs (methods)

- CBSE instead says that software should be developed by gluing components together like in electronics (software IC)

- However, a component may well be implemented as a single class or object, but may also be implemented in a procedural way, functional way or as many classes/objects
Component interfaces

- Provides interface
  ~ Defines the services that are provided by the component to other components

- Requires interface
  ~ Defines the services that must be made available for the component to execute as specified
Component models

- A component model is a definition of standards for component implementation, documentation and deployment

- Examples of component models
  - EJB model (Enterprise Java Beans)
  - COM+ model and .NET model
  - CORBA

- The component model specifies how interfaces should be defined and the elements that should be included in an interface definition
Middleware

- Middleware is the link between different distributed systems
- This can be some kind of ORB or RPC or an application server
Middleware

- Usually components are implemented using some kind of containers that provide the component with services
- These containers are connections to the application server/surrounding environment
Middleware

- Component models are the basis for middleware that provides support for executing components.

- Component model implementations provide the components with services, including:
  - Transaction management
  - Resource management
  - Concurrency
  - Persistence
  - Security
Component development for reuse

- Components developed for a specific application usually have to be generalized to make them reusable.

- A component is most likely to be reusable if it is associated with a stable domain abstraction (business object).

- For example, in a hospital, stable domain abstractions are associated with the fundamental purpose - nurses, patients, treatments, etc.
Component development for reuse

- Components for reuse may be specially constructed by generalizing existing components

- Component reusability
  - Should reflect stable domain abstractions
  - Should hide state representation
  - Should be as independent as possible
  - Should publish exceptions through the component interface
The reusability / usability trade-off

- In order to maximize reusability the interfaces of a component should be as general as possible.

- The more general an interface is the more complex it is, and thus harder to use.

- So - Maximizing reuse minimizes use.
Changes for reusability

- Remove application-specific methods
- Change names to make them general
- Add methods to broaden coverage of functionality
- Make exception handling consistent - make sure exceptions are not handled internally
- Add a configuration interface for component adaptation
- Integrate required components to reduce dependencies
Legacy system components

- Existing legacy systems that fulfill a useful business function can be re-packaged as components for reuse.

- This involves writing a wrapper component that implements provides and requires interfaces then accesses the legacy system.

- Although costly, this can be much less expensive than rewriting the legacy system.
Reusable components

- The development cost of reusable components may be higher than the cost of specific equivalents

  - This extra reusability enhancement cost should be a cost of the organization rather than a project cost as other parts of the organization are likely to benefit from the component

  - Otherwise it is not sensible to make it general in the first place

- Generic components may be less space-efficient and may have longer execution times than their specific equivalents
The CBSE process

- When reusing components, it is essential to make trade-offs between ideal requirements and the services actually provided by available components

- This involves:
  - Developing outline requirements
  - Searching for components and then modify requirements according to available functionality
  - Searching again to find if there are better components that meet the revised requirements
Component identification issues

- Trust
  - You need to be able to trust the supplier of a component. At best, an untrusted component may not operate as advertised; at worst, it can breach your security.

- Requirements
  - Different groups of components will satisfy different requirements.

- Validation
  - The component specification may not be detailed enough to allow comprehensive tests to be developed -- interface syntax vs. interface semantics.
  - Components may have unwanted functionality. How can you test that this will not interfere with your application?
Component composition

- The process of assembling components to create a system
- Composition involves integrating components with each other and with the component infrastructure
- Normally you have to write ‘glue code’ to integrate components
Interface incompatibility

- Parameter incompatibility
  - where operations have the same name but are of different types

- Operation incompatibility
  - where the names of operations in the composed interfaces are different - different names same semantics

- Operation incompleteness
  - where the provides interface of one component is a subset of the requires interface of another

- Architectural mismatch
  - assumptions made by the implementor of the component; control model, data model etc.
Adaptor components

- Address the problem of component incompatibility by translating between the interfaces of the components that are composed

- Here one should probably take the opportunity to make the adaptor a general component as well

- However, some glue code might well do the trick
Composition trade-offs

- When composing components
  - there may be several compositions that meet the requirements, or
  - there may be different compositions that meet different subsets of a set of requirements
Component markets

- The advocates of components have long anticipated component market shelves

- However, this has never reached any significant scale

- Reasons for this include:
  ~ Trust - Who ensures that the component works? Who pays if it doesn’t?
  ~ Market size - the market for many components is not sufficiently large for suppliers to establish a viable long-term business