Software Reuse and Component-Based Software Engineering

Minsoo Ryu

College of Information and Communications
Hanyang University

msryu@hanyang.ac.kr
Contents

- Software Reuse
- Components
- CBSE (Component-Based Software Engineering)
- Domain Engineering
- CBD (Component-Based Development)
Software Reuse

- In most engineering disciplines, systems are designed by composing existing components that have been used in other systems.

- Software engineering has been more focused on original development but it is now recognised that to achieve better software, more quickly and at lower cost, we need to adopt a design process that is based on systematic \textit{reuse}. 
Software Reuse

- Application system reuse
  - The whole of an application system may be reused either by incorporating it without change into other systems (COTS reuse) or by developing application families.

- Component reuse
  - Components of an application from sub-systems to single objects may be reused.

- Function reuse
  - Software components that implement a single well-defined function may be reused.
Benefits of Reuse

☒ Increased reliability
  ▪ Components exercised in working systems

☒ Reduced process risk
  ▪ Less uncertainty in development costs

☒ Effective use of specialists
  ▪ Reuse components instead of people

☒ Standards compliance
  ▪ Embed standards in reusable components

☒ Accelerated development
  ▪ Avoid original development and hence speed-up production
Reuse Problems

- Increased maintenance costs
  - Source code is not available
- Lack of tool support
  - CASE tools do not support development with reuse
- Not-invented-here syndrome
  - Some engineers prefer to rewrite
- Maintaining a component library
- Finding and adapting reusable components
  - Software components have to be discovered in a library, understood and, adapted to work in a new environment
Contents

- Software Reuse
- Components
- CBSE (Component-Based Software Engineering)
- Domain Engineering
- CBD (Component-Based Development)
Definitions of Components

☐ Councill and Heinmann:
  ▪ A software 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.

☐ Szyperski:
  ▪ 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.
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

<table>
<thead>
<tr>
<th>Standardised</th>
<th>Component standardisation means that a component that is used in a CBSE process has to conform to some standardised component model. This model may define component interfaces, component meta-data, documentation, composition and deployment.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Independent</td>
<td>A component should be independent Š 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 ŠrequiresŠ interface specification.</td>
</tr>
<tr>
<td>Composable</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Component Characteristics 2

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Deployable</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Documented</strong></td>
<td>Components have to be fully documented so that potential users of the component can decide whether or not they meet their needs. The syntax and, ideally, the semantics of all component interfaces have to be specified.</td>
</tr>
</tbody>
</table>
Component Interfaces

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

- **Requires interface**
  - Defines the services that specifies what services must be made available for the component to execute as specified.
Component Interfaces

Requirements interface
- Defines the services from the component's environment that it uses

Provides interface
- Defines the services that are provided by the component to other components
A Data Collector Component

Requires interface

- sensorManagement
- sensorData

Provides interface

- addSensor
- removeSensor
- startSensor
- stopSensor
- testSensor
- initialise
- report
- listAll
Components vs. Objects

- Objects are not primarily concerned about reuse

- Components aim at reuse
  - Components are deployable entities
  - Components do not define types
  - Component implementations are opaque
  - Components are language-independent
  - Components are standardized
Contents

- Software Reuse
- Components
- CBSE (Component-Based Software Engineering)
- Domain Engineering
- CBD (Component-Based Development)
Component-Based Software Engineering

- Reuse is an idea both old and new
  - Programmers have reused ideas, abstractions, and processes since the earliest days of computing, but the early approach to reuse was ad hoc

- CBSE is a process that emphasizes the design and construction of computer-based systems using reusable software components
Component-Based Software Engineering

- Clements describes CBSE in the following way
  - CBSE is changing the way large software systems are developed. CBSE embodies the “buy, don’t build” philosophy espoused by Fred Brooks and others. In the same way that early subroutines liberated the programmer from thinking about details, CBSE shifts the emphasis from programming software to composing software systems. Implementation has given way to integration as the focus. As its foundation is the assumption that there is sufficient commonality in many large software systems to justify developing reusable components to exploit and satisfy that commonality.
Component-Based Software Engineering

A number of questions arise about CBSE

- Is it possible to construct complex systems by assembling them from a catalog of reusable software components?
- Can this be accomplished in a cost- and time-effective manner?
- Can appropriate incentives be established to encourage software engineers to reuse rather than reinvent?
- Is management willing to incur the added expense associated with creating reusable software components?
- Can the library of components necessary to accomplish reuse be created in a way that makes it accessible to those who need it?
- Can components that do exist be found by those who need them?
Engineering of Component-Based Systems

- On the surface, CBSE seems quite similar to conventional or object-oriented software engineering.
  - But, once an architecture design is established, the team examines requirements to determine what subset is directly amenable to composition, rather than moving into more detailed design tasks.

- At this stage, the following questions must be considered:
  - Are commercial off-the-shelf (COTS) components available to implement the requirement?
  - Are internally developed reusable components available to implement the requirement?
  - Are the interfaces for available components within the architecture of the system to be built?
Contents

- Software Reuse
- Components
- CBSE (Component-Based Software Engineering)
- Domain Engineering
- CBD (Component-Based Development)
The CBSE Process

- The process of CBSE emphasizes two parallel tracks
  - Domain engineering and component-based development

- Domain engineering
  - Identify, construct, catalog, and disseminate a set of software components that have applicability to existing and future software in a particular application domain

- CBD (component-based development)
  - Develops a system using reusable components
The CBSE Process

Domain Engineering
- Domain Analysis
- Domain Design
- Domain Implementation
- Domain model
- Generic Architecture
- Repository Reusable Artifacts/Components

Application Engineering
- User Requirements
- System Analysis
- System Spec
- Specification & Design
- Analysis & Design Models
- Construction
- Application Software

Real-Time Computing and Communications Lab., Hanyang University
http://rtcc.hanyang.ac.kr
The activities of domain engineering include identifying a domain, capturing the commonalities and differences within a domain (domain analysis), constructing an adaptable design (domain design), and defining the mechanisms for translating requirements into systems created from reusable components (domain implementation).
Domain Analysis

Domain analysis is "the process of identifying, collecting, organizing, and representing the relevant information in a domain, based upon the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within a domain"
Domain Design

- Domain design is the process of developing a design model from the products of domain analysis and the knowledge gained from the study of software requirement/design reuse and generic architectures.

- Nilson et al. states that "The key to design reuse is to develop and document an architecture that is generic enough for most systems in the domain and that supports the reuse of code components in the domain."
Structural Modeling and Structure Points

- Structural modeling is a pattern-based domain engineering approach that works under the assumption that every application domain has repeating patterns.

- The structural model is an architectural style that can and should be reused across applications within the domain.
  - (e.g.) Aircraft avionics systems differ greatly in specifics, but all modern software in this domain has the same structural model.
A “structure point” is a construct within a structural model

- A structure point is an abstraction that should have a limited number of instances. Restating this in object-oriented jargon, the size of the class hierarchy should be small.
- The rules that govern the use of the structure point should be easily understood. In addition, the interface to the structure point should be relatively simple.
- The structure point should implement information hiding by hiding all complexity contained within the structure point itself. This reduces the perceived complexity of the overall system.
Examples of Structure Points (Alarm Systems)

- An interface that enables the user to interact with the system
- A bounds-setting mechanism that allows the user to establish bounds on the parameters to be measured
- A sensor management mechanism that communicates with all monitoring sensors
- A response mechanism that reacts to the input provided by the sensor management system
- A control mechanism that enables the user to control the manner in which monitoring is carried out
Generic Structure Points

- Application front-end
  - The GUI including all menus, panels, and input and command editing facilities

- Database
  - The repository for all objects relevant to the application domain

- Computational engine
  - The numerical and nonnumerical models that manipulate data

- Reporting facility
  - The function that produces output of all kinds

- Application editor
  - The mechanism for customizing the application to the needs of specific users
Domain Implementation

- Domain implementation is the process of identifying and creating reusable components based on the domain model and generic architecture
  - Create reusable assets which are catalogued into a component library for use by application engineers
  - These reusable components are the principal outputs of this phase of domain engineering
  - Creation, management, and maintenance of a repository of reusable assets is also an important part of domain implementation
Identifying Reusable Components

☐ Is component functionality required on future implementations?
☐ How common is the component's function within the domain?
☐ Is there duplication of the component's function within the domain?
☐ Is the component hardware-dependent?
☐ Does the hardware remain unchanged between implementations?
☐ Can the hardware specifics be removed to another component?
☐ Is the design optimized enough for the next implementation?
☐ Can we parameterize a non-reusable component so that it becomes reusable?
☐ Is the component reusable in many implementations with only minor changes?
☐ Is reuse through modification feasible?
☐ Can a non-reusable component be decomposed to yield reusable components?
☐ How valid is component decomposition for reuse?
Contents

- Software Reuse
- Components
- CBSE (Component-Based Software Engineering)
- Domain Engineering
- CBD (Component-Based Development)
Once an architecture has been established, it must be populated by components.

The task flow of CBD has two parallel paths:

- When reusable components are available, they must be qualified and adapted.
- When new components are required, they must be engineered.
Component Qualification

- **Sources of reusable components**
  - In-house, existing applications, and third parties

- **Component qualification**
  - This ensures that a candidate component will perform the function required, will properly “fit” into the architectural style specified for the system, and will exhibit the quality characteristics (e.g., performance, reliability, usability) that are required for the application
Component Qualification

Factors to be considered during qualification

- application programming interface (API)
- development and integration tools required by the component
- run-time requirements including resource usage (e.g., memory or storage), timing or speed, and network protocol
- service requirements including operating system interfaces and support from other components
- security features including access controls and authentication protocol
- embedded design assumptions including the use of specific numerical or non-numerical algorithms
- exception handling
Component Adaptation

- Even after a component has been qualified, conflicts may occur in one or more areas of concern.
- To avoid these conflicts, an adaptation technique called “component wrapping” is often used.
  - White-box wrapping
    - When a software team has full access to the internal design and code for a component.
    - Code-level modifications to remove any conflicts.
  - Gray-box wrapping
    - When the component library provides a component extension language or API.
  - Black-box wrapping
    - Requires the introduction of pre- and post-processing at the component interface.
Component Incompatibility

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

- Operation incompatibility
  - The names of operations in the composed interfaces are different

- Operation incompleteness
  - The provides interface of one component is a subset of the requires interface of another
Adaptor for Data Collector
Component Composition

- An infrastructure must be established to bind components together.

- The infrastructure (usually a library of specialized components) provides a model for the coordination of components and specific services that enable components to coordinate with one another and perform common tasks.
Component Composition

Architectural ingredients for composition include:

- **Data exchange model**
  - Mechanisms that enable users and components to interact and transfer data should be defined for all reusable components

- **Automation**
  - A variety of tools, macros, and scripts should be implemented to facilitate interaction between reusable components

- **Structured storage**
  - It should be possible to freely navigate to locate, create, or edit data contents

- **Underlying object model**
  - Components written in different languages that reside on different platforms can be interoperable
Types of Composition

- Sequential composition (a)
  - The composed components are executed in sequence
  - This involves composing the provides interfaces of each component

- Hierarchical composition (b)
  - One component calls on the services of another
  - The provides interface of one component is composed with the requires interface of another

- Additive composition (c)
  - The interfaces of two components are put together to create a new component
Types of Composition

(a)  

(b)  

(c)
Component Engineering

- Even though the CBSE encourages the use of existing components, there are times when new software components must be developed and integrated with existing COTS and in-house components.

- Because these new components become members of the in-house library of reusable components, they should be engineered for reuse.
Nothing is magical about creating software components that can be reused

Design concepts such as abstraction, hiding, functional independence, and refinement all contribute to the creation of software components that are reusable
Analysis and Design for Reuse

Binder suggests a number of key issues that form a basis for design for reuse (DFR)

- **Standard data**
  - The application domain should be investigated and standard global data structures should be identified

- **Standard interface protocols**
  - Three levels of interface protocol should be established
  - Intramodular interfaces, external technical interfaces, and the human/machine interfaces

- **Program templates**
  - The structure model can serve as a template for the architectural design of a new program
Economics of CBSE

☐ Quality
   ▪ In a study conducted at Hewlett Packard
     • The defect rate for reused code is 0.9 defects per KLOC
     • The defect rate for newly developed software is 4.1 defects per KLOC

☐ Productivity
   ▪ It appears that 30 to 50 percent reuse can result in productivity improvements in the 25—40 percent range

☐ Cost
   ▪ The cost to develop a reusable component is often greater than the cost to develop a component that is specific to one application
   ▪ Be sure that there will be a need for the reusable component in the future