This chapter will present a complete picture of how to develop software systems with components and system integration.
Reuse: Past and Present

• reuse is both an old and a new idea. Programmers have reused ideas, abstractions, and processes since the earliest days of computing.

• the early approach to reuse was ad hoc.

• Today, complex, high quality computer-based systems must be built in very short time periods. This mitigates towards a more organized approach to reuse.
What is Component-Based Software Engineering (CBSE)

- CBSE is a process that emphasizes the design and construction of computer-based systems using reusable software components.
- CBSE is changing the way large software systems are developed. CBSE embodies the “buy, do not build” philosophy espoused by some software engineers.
- CBSE shifts the emphasis from programming software to composing software systems.
What is Component-Based Software Engineering (CBSE)

• Implementation has given way to integration as the focus.

• The foundation of CBSE is the assumption that there is sufficient commonality in many large software systems to justify developing reusable components to exploit and satisfy that commonality.
Architectural design & identification of composition candidate

• CSBE process begins with establishing requirements for the system using conventional requirements elicitation techniques.

• After architectural design established, rather than moving immediately into detailed design tasks, the team examines requirements to determine what subsets are directly amenable to composition, rather than construction.
Architectural design & identification of composition candidate

- The criteria are:
  - 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 compatible within the architecture of the system to be built?
Architectural design & identification of composition candidate

• The team attempts to modify or remove those system requirements that cannot be implemented with COTS or in-house components.

• If the requirement cannot be changed or deleted, conventional or object-oriented software engineering methods are applied to develop these new components that must be engineered to meet the requirement.
Component Engineering

For those requirements that are addressed with available components, the following software engineering activities must be done:

- Component qualification
- Component adaptation.
- Component composition.
- Component update.
Component qualification

- System requirements and architecture define the components that will be required.
- Reusable components are normally identified by the characteristics of their interfaces. That is, “the services that are provided, and the means by which consumers access these services”.
- However, the interface does not provide the degree to which the component will fit the architecture and requirements.
Component adaptation

- Software architecture represents design patterns that are composed of components connections, and coordination. In some cases, existing reusable components may be mismatched to the architecture’s design rules. These components must be adapted to meet the needs of the architecture or discarded and replaced by other more suitable components.

- For example, some components are configurable through parameterisation.
Component composition

• Architectural style again plays a key role in the way in which software components are integrated to form a working system.

• By identifying connection and coordination mechanisms (e.g., run-time properties of the design), the architecture dictates the component of the end-product.
Component update

• When systems are implemented with COTS components, update is complicated by the imposition of a third party. The organisation that developed the reusable component may be outside the immediate control of the software engineering organisation.
Classification of Components

• Software components can also be characterised based on their use in the CBSE process. In addition to COTS components, the CBSE process yields:

• Qualified components—assessed by software engineers to ensure that not only functionality, but also performance, reliability, usability, and other quality factors conform to the requirements of the system/product to be built.
Classification of Components

- Adapted components—adapted to modify (wrapping) unwanted or undesired characteristics.

- Assembled components—integrated into an architectural style and interconnected with an appropriate component infrastructure that allows the components to be coordinated and managed effectively.

- Updated components—replacing existing software as new versions of components become available.
The CBSE process

- The CBSE process includes to identify candidate components, to qualify each component’s interface, to adapt components to remove architectural mismatches, to assemble components into a selected architectural style, and to update components as requirements for the system change.

- The process model for component-based software engineering emphasizes parallel tracks in which domain engineering occurs concurrently with component-based development.
The CBSE process

Domain Engineering

Domain Analysis

Software Architecture Development

Reusable Component Development

Domain Model

Structural Model

Repository Reusable Artifacts/Component

Component-Based Development

Analysis

Architectural Design

Component Qualification

Component Adaptation

Component Composition

Component Engineering

Testing

Component Update

Application Software
Domain Engineering

• The intent of domain engineering is to identify, construct, catalogue, and disseminate a set of software components that have applicability to existing and future software in a particular application domain.

• The overall goal is to establish mechanisms that enable software engineers to share these components—to reuse them.
Domain Engineering

- Domain engineering includes three major activities:
  - analysis
  - construction, and
  - dissemination.
The domain analysis process

• An 8 step approach to the identification and categorisation of reusable components
  – select specific functions/objects
  – abstract functions/objects
  – define a taxonomy
  – identify common features
  – identify specific relationships
  – abstract the relationships
  – derive a functional model
  – define a domain language
A guide for identifying reusable components

- Is the component functionality required on future implementation?
- 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?
A guide for identifying reusable components

• Does the hardware remain unchanged between implementations?
• Can the hardware specifics be removed to another components?
• Is the design optimized enough for the next implementation?
• Can we parameterize a non-reusable components so that it becomes reusable?
A guide for identifying reusable components

- 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?
Domain Characteristics

- It is sometimes difficult to determine whether a potentially reusable component is applicable in a particular situation. A set of domain characteristics may be defined to make this determination.

- A domain characteristic is shared by all software within a domain. It defines generic attribute of all products that exist within the domain. E.g., generic characteristics might include: the importance of safety/reliability, programming language, concurrency in processing.
Component qualification
Definition

- Component qualification 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) required for the application.
Component qualification -- Factors to be considered

- The interface description provides useful information about the operation and use of a software component, but it does not provide all of the information required to determine if a proposed component can be reused effectively in a new application.

- Among the many factors considered during component qualification are:
  - Application programming interface (API)
Component qualification --
Factors to be considered

- 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
Component qualification

Factors to be considered

- 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--ideal situation

- In an ideal setting, domain engineering creates a library of components that can be easily integrated into application architecture. The implication of “easy integration” is:
  - that consistent methods of resource management have been implemented for all components in the library;
  - that common activities such as data management exist for all components, and
  - that interfaces within the architecture and with the external environment have been implemented in a consistent manner.
Component adaptation wrapping

• In reality, even after a component has been qualified for use in an application architecture, it may exhibit conflicts in one or more of the areas.

• To mitigate these conflicts, an adaptation technique called “component wrapping” is often used.
Component adaptation

wrapping

- When a software team has full access to the internal design and code for a component, white-box wrapping is applied. This wrapping examines the internal processing details and makes code-level modifications to remove any conflicts.

- Grey-box wrapping is applied when the component library provides a component extension language or API that enables conflicts to be removed or masked.
Component adaptation -- wrapping

- Black-box wrapping requires the introduction of pre- and post-processing at the component interface to remove or mask conflicts.

- The software team must determine whether the effort required to adequately wrap a component is justified or whether a custom component should be engineered instead.
Component composition

• The component composition task assembles qualified, adapted, and engineered components to populate the architecture established for an application.

• To accomplish this, an infrastructure (usually a library of specialized components) provides a model for coordination of components and specific services that enable components to coordinate with each other and perform common tasks.
Component composition

Among the many mechanisms for creating an effective infrastructure is a set of four “architectural ingredients” that should be present to achieve component composition:

- Data exchange model. Mechanisms that enable users and applications to interact and transfer data (e.g., drag and drop, cut and paste) should be defined for all reusable components. The mechanism should cover:
Component composition

- human-to-software data transfer
- component-to-component data transfer, and
- data transfer among system resources (e.g., dragging a file to a printer icon for output).

• Automation. A variety of tools, macros, and scripts should be implemented to facilitate interaction between reusable components.
Component composition

- Structured storage. Heterogeneous data contained in a compound document should be organized and accessed as a single data structure.

- Underlying object model. The object model ensures that components developed in different programming languages that reside on different platforms can be interoperable. I.e., objects must be capable of communicating across network.
Component composition

• Most popular existing industrial standards for component software include:
  - OMG/CORBA
  - Microsoft COM
  - Sun javaBean
Component engineering

- Component engineering focuses on how to build quality reusable software components.
- Though CBSE encourages the use of existing components, in many cases building new components becomes a must to enhance the component library.
- To create software components involves the integration of the following techniques.
Component engineering

- Abstraction
- Hiding
- Functional independence
- Refinement
- Structured programming
- Object-oriented methods
- Testing
- SQA and correctness verification
Analysis and design for reuse

- Automatic tools may be used to browse a repository in an attempt to match the requirement noted in the current specification with those described for existing reusable components.

- Characterization functions and keywords are used to help find potentially reusable components.
Analysis and design for reuse

• If specification matching yields components that fit the needs for the current application, the designer can extract these components from the repository and integrate them in the design of new system.

• If no suitable component can be found, component engineering must be applied to create them.
Analysis and design for reuse

• To create a reusable component, the following key issues need to be considered:
  - Standard data. The application domain should be investigated and standard global data structures (e.g., file structures or database) should be identified. All design components can be characterized to make use of these standard data structures.
Analysis and design for reuse

- Standard interface protocols. Three levels of interface protocol should be established: the nature of intra-modular interfaces, the design of external technical (non-human) interfaces, and the human-machine interface.

- Program templates. The structure model can serve as a template for the architectural design of a new program.
Describing reuse components

• Reusable component description: 3C model---concept, content and context.

• The concept of a component is a description of what the component does. The interface to the component is fully described and the semantics, represented within the context of pre- and post-conditions, is defined. The intent of the component should be included in the concept.
Describing reuse components

• The content describes how the concept is realized. In essence, the content is information hidden from casual users and needs be known only to those who intend to modify the component.

• The context places a component within its domain of applicability. By specifying conceptual, operational, and implementation features, the context enables a software engineer to find the appropriate component to meet application requirements.
Transfer the 3C model into a concrete scheme

- Library and information science methods
  - Enumerated classification
  - Faceted classification
  - Attribute-value classification
- Artificial intelligence methods
- Hypertext systems
Enumerated classification

• Components are described by a defined hierarchical structure in which classes and varying levels of subclasses are defined. Actual components are at the leaf level.

• E.g., an enumerated hierarchy for window operations.

• The hierarchical structure is easy to understand and to use, but it lacks of flexibility.
Enumerated classification

Window operations
  Display
  Open
    Menu-based
      OpenWindow
    System-based
      SysWindow
  Close
    Via pointer
      ...
  Resize
    via command
      setWindowSize, stdResize, shrinkWindow
    via drag
      pullWindow, stretchWindow
  Up/down shuffle
    ...
  Move
    ...
  Close
    ...

example
Faceted classification

• A domain area is analyzed and a set of basic descriptive features is identified. These features, named facets, are then prioritized by importance and connected to a component.

• A facet can describe
  - the function of a component
  - the data manipulated
  - the context in which components applied
Faceted classification

• The set of facets describing a component is called the facet descriptor.

• E.g.,
  
  \{function, object type, system type\}

• Keywords (values) are assigned to the set of facets for each component in a reuse repository. Software engineers use keywords to search for possible components. Automated tools can be used to facilitate the search.
Faceted classification

• The advantage of faceted classification is flexibility. The facets and linked value can be added, deleted or modified easily.
Attribute-value classification

- A set of attributes is defined for all components in domain area.
- Values are then assigned to these attributes in much the same way a faceted classification.
- Attribute value classification is similar to faceted classification with the following exception:
  - there is no limit on the number of attributes that can be used
  - attributes are not assigned priorities
  - the thesaurus functions are not used
The reuse environment

• A component database capable of storing software components and the classification information necessary to retrieve them.

• A library management system that provides access to the database.

• A software component retrieval system that enables a client application to retrieve components and services from the library server. E.g., object request broker.

• CBSE tools that support the integration of reused components into a new design or implementation.
Economics of CBSE

- CBSE should provide a software organisation with advantages in quality and timeliness.
- Impact on quality, productivity and cost
- Reuse Metrics
Quality

- Ideal setting, components for reuse would be verified and defect-free.
- In reality, formal verification is not carried out routinely. But, with reuse, defects are found and eliminated. Over time, the component becomes defect free.
- In a HP study, the defect rate for reused code is 0.9 defects per KLOC, while for newly developed software is 4.1 per KLOC.
Productivity

- Hard to estimate a figure
- Less time spent in creating plans, models, documents, code and data
- It appears that 30-50% reuse can result in 25-40% productivity improvement
Cost

• Net cost saving for reuse = cost if developed from scratch – cost associated with reuse

• The costs associated with reuse include:
  - Domain analysis and modelling
  - Domain architecture development
  - Increased documentation to facilitate reuse
  - Support and enhancement of reuse components
Cost

- Royalties and licenses for externally acquired components
- Creation or acquisition and operation of a reuse repository
- Training of personnel in design and construction for reuse
The benefit associated with reuse within a system $S$ can be expressed as a ration:

$$R_b (S) = \frac{C_{noreuse} - C_{reuse}}{C_{noreuse}}$$

where $C_{noreuse}$ is the cost of developing $S$ with no reuse, and $C_{reuse}$ is the cost of developing $S$ with reuse.

So, $0 \leq R_b (S) \leq 1$
Benefits of Reuse

- \( R_b(S) \) will be affected by the design of the system.
- It is important to make a part of an assessment of design alternatives.
- The benefit associated with reuse is closely aligned to the cost benefit of each individual reusable component.
Reuse measure for OO systems

\[ R_{lev} = \frac{OBJ_{reused}}{OBJ_{built}} \]

where \( OBJ_{reused} \) is the number of objects reused in a system, and \( OBJ_{built} \) is the number of objects built for a system.