SOFTWARE ENGINEERING

LECTURE-15

Component-Level Design

TOPICS COVERED

- Introduction
- The software component
- Designing class-based components
- Designing conventional components

INTRODUCTION

BACKGROUND

- Component-level design occurs after the first iteration of the architectural design
- It strives to create a <u>design model</u> from the analysis and architectural models
  - The translation can open the door to <u>subtle errors</u> that are difficult to find and correct later
  - "Effective programmers <u>should not waste their time</u> debugging – they should not introduce bugs to start with." Edsgar Dijkstra
- A component-level design can be represented using some <u>intermediate representation</u> (e.g. graphical, tabular, or text-based) that can be translated into source code
- The design of data structures, interfaces, and algorithms should conform to well-established <u>guidelines</u> to help us avoid the introduction of errors

THE SOFTWARE COMPONENT

DEFINED

- A software component is a modular building block for computer software
  - It is a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces
- A component communicates and collaborates with
  - Other components
  - Entities outside the boundaries of the system
- Three different views of a component
  - An object-oriented view
  - A conventional view
  - A process-related view

OBJECT-ORIENTED VIEW

- A component is viewed as a set of one or more collaborating classes
- Each problem domain (i.e., analysis) class and infrastructure (i.e., design) class is elaborated to identify all attributes and operations that apply to its implementation
  - This also involves defining the interfaces that enable classes to communicate and collaborate
- This elaboration activity is applied to every component defined as part of the architectural design
- Once this is completed, the following steps are performed
  1) Provide further elaboration of each attribute, operation, and interface
  2) Specify the data structure appropriate for each attribute
  3) Design the algorithmic detail required to implement the processing logic associated with each operation
  4) Design the mechanisms required to implement the interface to include the messaging that occurs between objects

CONVENTIONAL VIEW

- A component is viewed as a <u>functional</u> element (i.e., a module) of a program that incorporates
    - The <u>processing logic</u>
    - The internal <u>data structures</u> that are required to implement the processing logic
    - An <u>interface</u> that enables the component to be invoked and data to be passed to it
- A component serves one of the following roles
    - A <u>control</u> component that coordinates the invocation of all other problem domain components
    - A <u>problem domain</u> component that implements a complete or partial function that is required by the customer
    - An <u>infrastructure</u> component that is responsible for functions that support the processing required in the problem domain

(More on next slide)

CONVENTIONAL VIEW (CONTINUED)

- Conventional software components are derived from the data flow diagrams (DFDs) in the analysis model
  - Each transform bubble (i.e., module) represented at the lowest levels of the DFD is mapped into a module hierarchy
  - Control components reside near the top
  - Problem domain components and infrastructure components migrate toward the bottom
  - Functional independence is strived for between the transforms
- Once this is completed, the following steps are performed for each transform
  1) Define the <u>interface</u> for the transform (the order, number and types of the parameters)
  2) Define the <u>data structures</u> used internally by the transform
  3) Design the <u>algorithm</u> used by the transform (using a stepwise refinement approach)

PROCESS-RELATED VIEW

- Emphasis is placed on building systems from existing components maintained in a library rather than creating each component from scratch
- As the software architecture is formulated, components are selected from the library and used to populate the architecture
- Because the components in the library have been created with reuse in mind, each contains the following:
  - A complete description of their interface
  - The functions they perform
  - The communication and collaboration they require

DESIGNING CLASS-BASED COMPONENTS

COMPONENT-LEVEL DESIGN PRINCIPLES

- **Open-closed principle**
  - A module or component should be open for extension but closed for modification
  - The designer should specify the component in a way that allows it to be extended without the need to make internal code or design modifications to the existing parts of the component
- **Liskov substitution principle**
  - Subclasses should be substitutable for their base classes
  - A component that uses a base class should continue to function properly if a subclass of the base class is passed to the component instead
- **Dependency inversion principle**
  - Depend on abstractions (i.e., interfaces); do not depend on concretions
  - The more a component depends on other concrete components (rather than on the interfaces) the more difficult it will be to extend
- **Interface segregation principle**
  - Many client-specific interfaces are better than one general purpose interface
  - For a server class, specialized interfaces should be created to serve major categories of clients
  - Only those operations that are relevant to a particular category of clients should be specified in the interface

COMPONENT PACKAGING PRINCIPLES

- Release reuse equivalency principle
  - The granularity of reuse is the granularity of release
  - Group the reusable classes into packages that can be managed, upgraded, and controlled as newer versions are created
- Common closure principle
  - Classes that change together belong together
  - Classes should be packaged cohesively; they should address the same functional or behavioral area on the assumption that if one class experiences a change then they all will experience a change
- Common reuse principle
  - Classes that aren't reused together should not be grouped together
  - Classes that are grouped together may go through unnecessary integration and testing when they have experienced no changes but when other classes in the package have been upgraded

COMPONENT-LEVEL DESIGN GUIDELINES

- Components
  - Establish naming conventions for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model
  - Obtain architectural component names from the problem domain and ensure that they have meaning to all stakeholders who view the architectural model (e.g., Calculator)
  - Use infrastructure component names that reflect their implementation-specific meaning (e.g., Stack)
- Dependencies and inheritance in UML
  - Model any dependencies from left to right and inheritance from top (base class) to bottom (derived classes)
  - Consider modeling any component dependencies as interfaces rather than representing them as a direct component-to-component dependency

COHESION

- Cohesion is the “single-mindedness’ of a component
- It implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself
- The objective is to keep cohesion as high as possible
- The kinds of cohesion can be ranked in order from highest (best) to lowest (worst)
  - Functional
    - A module performs one and only one computation and then returns a result
  - Layer
    - A higher layer component accesses the services of a lower layer component
  - Communicational
    - All operations that access the same data are defined within one class

(More on next slide)

COHESION (CONTINUED)

- Kinds of cohesion (continued)
  - Sequential
    - Components or operations are grouped in a manner that allows the first to provide input to the next and so on in order to implement a sequence of operations
  - Procedural
    - Components or operations are grouped in a manner that allows one to be invoked immediately after the preceding one was invoked, even when no data passed between them
  - Temporal
    - Operations are grouped to perform a specific behavior or establish a certain state such as program start-up or when an error is detected
  - Utility
    - Components, classes, or operations are grouped within the same category because of similar general functions but are otherwise unrelated to each other

COUPLING

- As the amount of communication and collaboration increases between operations and classes, the complexity of the computer-based system also increases
- As complexity rises, the difficulty of implementing, testing, and maintaining software also increases
- Coupling is a qualitative measure of the degree to which operations and classes are connected to one another
- The objective is to keep coupling as low as possible

(More on next slide)

COUPLING (CONTINUED)

- The kinds of coupling can be ranked in order from lowest (best) to highest (worst)
  - Data coupling
    - Operation A() passes one or more atomic data operands to operation B(); the less the number of operands, the lower the level of coupling
  - Stamp coupling
    - A whole data structure or class instantiation is passed as a parameter to an operation
  - Control coupling
    - Operation A() invokes operation B() and passes a control flag to B that directs logical flow within B()
    - Consequently, a change in B() can require a change to be made to the meaning of the control flag passed by A(), otherwise an error may result
  - Common coupling
    - A number of components all make use of a global variable, which can lead to uncontrolled error propagation and unforeseen side effects
  - Content coupling
    - One component secretly modifies data that is stored internally in another component

(More on next slide)

COUPLING (CONTINUED)

- Other kinds of coupling (unranked)
  - Subroutine call coupling
    - When one operation is invoked it invokes another operation within side of it
  - Type use coupling
    - Component A uses a data type defined in component B, such as for an instance variable or a local variable declaration
    - If/when the type definition changes, every component that declares a variable of that data type must also change
  - Inclusion or import coupling
    - Component A imports or includes the contents of component B
  - External coupling
    - A component communicates or collaborates with infrastructure components that are entities external to the software (e.g., operating system functions, database functions, networking functions)

CONDUCTING COMPONENT-LEVEL DESIGN

1) Identify all design classes that correspond to the <u>problem</u> domain as defined in the analysis model and architectural model
2) Identify all design classes that correspond to the <u>infrastructure</u> domain
    - These classes are usually not present in the analysis or architectural models
    - These classes include GUI components, operating system components, data management components, networking components, etc.
3) Elaborate all design classes that are not acquired as reusable components
    a) Specify message details (i.e., structure) when classes or components collaborate
    b) Identify appropriate interfaces (e.g., abstract classes) for each component
    c) Elaborate attributes and define data types and data structures required to implement them (usually in the planned implementation language)
    d) Describe processing flow within each operation in detail by means of pseudocode or UML activity diagrams

(More on next slide)

CONDUCTING COMPONENT-LEVEL DESIGN (CONTINUED)

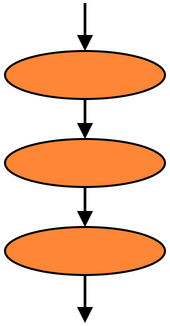
4) Describe persistent data sources (databases and files) and identify the classes required to manage them
5) Develop and elaborate behavioral representations for a class or component
   - This can be done by elaborating the UML state diagrams created for the analysis model and by examining all use cases that are relevant to the design class
6) Elaborate deployment diagrams to provide additional implementation detail
   - Illustrate the location of key packages or classes of components in a system by using class instances and designating specific hardware and operating system environments
7) Factor every component-level design representation and always consider alternatives
   - Experienced designers consider all (or most) of the alternative design solutions before settling on the final design model
   - The final decision can be made by using established design principles and guidelines

DESIGNING CONVENTIONAL COMPONENTS

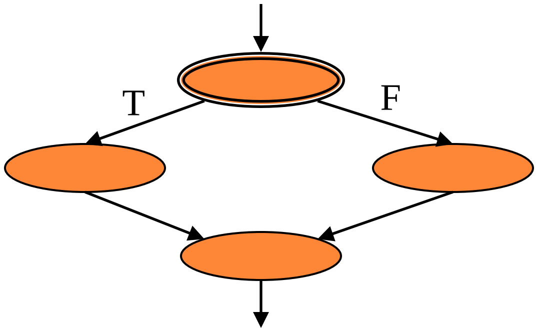
INTRODUCTION

- Conventional design constructs emphasize the maintainability of a functional/procedural program
  - Sequence, condition, and repetition
- Each construct has a predictable logical structure where control enters at the top and exits at the bottom, enabling a maintainer to easily follow the procedural flow
- Various notations depict the use of these constructs
  - Graphical design notation
    - Sequence, if-then-else, selection, repetition (see next slide)
  - Tabular design notation (see upcoming slide)
  - Program design language
    - Similar to a programming language; however, it uses narrative text embedded directly within the program statements

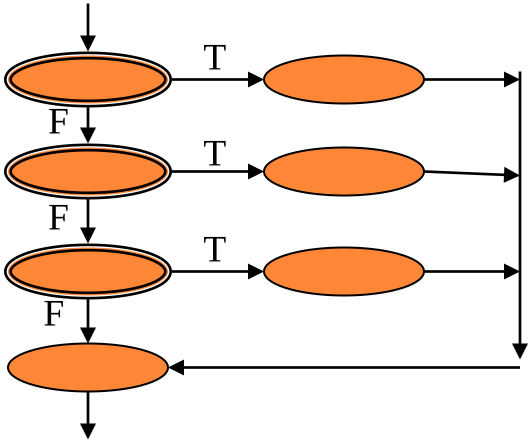
GRAPHICAL DESIGN NOTATION



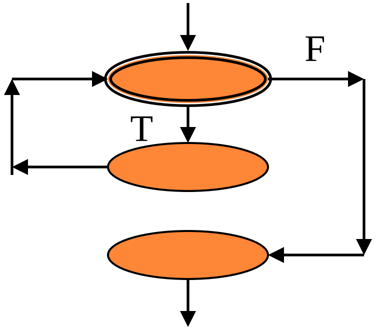
Sequence



If-then-else



Selection



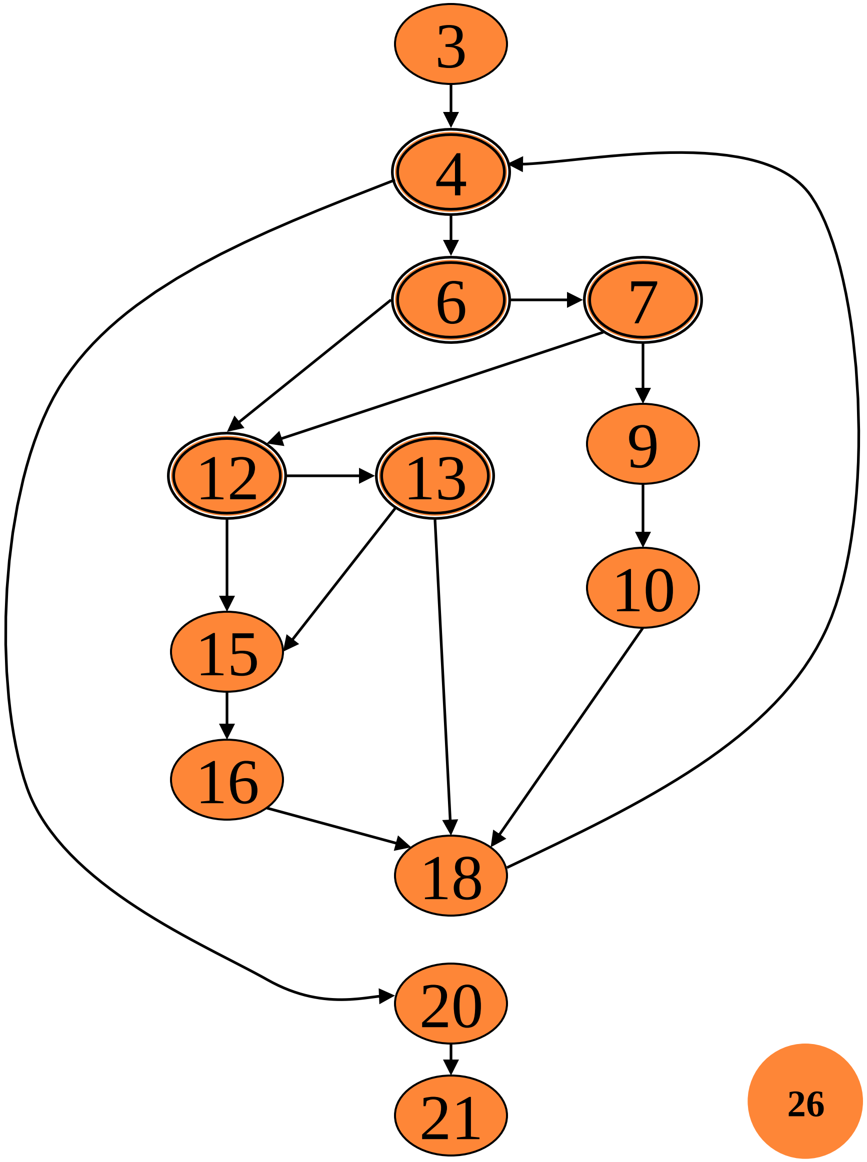
Repetition

GRAPHICAL EXAMPLE USED FOR ALGORITHM ANALYSIS

```
 1   int functionZ(int y)
 2   {
 3   int x = 0;

 4   while (x <= (y * y))
 5       {
 6       if ((x % 11 == 0) &&
 7            (x % y == 0))
 8          {
 9          printf(“%d”, x);
10          x++;
11          } // End if
12      else if ((x % 7 == 0) ||
13                 (x % y == 1))
14          {
15          printf(“%d”, y);
16          x = x + 2;
17          } // End else
18      printf(“\n”);
19      } // End while

20   printf("End of list\n");
21   return 0;
22   } // End functionZ
```



TABULAR DESIGN NOTATION

1) List all <u>actions</u> that can be associated with a specific procedure (or module)
2) List all <u>conditions</u> (or decisions made) during execution of the procedure
3) <u>Associate</u> specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions
4) Define <u>rules</u> by indicating what action(s) occurs for a set of conditions

(More on next slide)

TABULAR DESIGN NOTATION (CONTINUED)

Rules

Conditions	1	2	3	4
Condition A	T	T		F
Condition B		F	T	
Condition C	T			T
Actions				
Action X	✓		✓	
Action Y				✓
Action Z	✓	✓		✓

☺