55:182/22C:182
Software Engineering Languages and Tools

Spring, 2005

Instructor Information

• Instructor:
  – Professor Jon Kuhl, Elec. And Computer Eng.
    • office: 4016A SC
    • e-mail: kuhl@engineering.uiowa.edu
    • phone: 335-5958
    • Office hours: 1:00-2:00 p.m. M.W.F. (Other times by chance or appointment.)

Course Material

• Required Texts:

• Additional Supplemental Texts:
  – See list on class web page.

• Additional Readings:
  – Various web-based material will be assigned. URLs will be posted on the class web page.

Class Web Site

• http://www.engineering.uiowa.edu/~kuhl/CompSoft

• Information on class web page
  – office hours, e-mail addresses etc.
  – announcements
  – lecture notes–copies of slides used in class.
  – assignments and solutions
  – supplementary material
  – grades

• Students should check the web site for new postings on at least a daily basis.
Required Background for This Course

- General familiarity with Object-Oriented
  - Notion of classes, objects, methods, attributes
  - Abstract vs. concrete classes
  - Inheritance and class hierarchies
  - Polymorphism
  - OO Analysis and Design
- Familiarity with an OO programming language, preferably C++ or Java.
- Some familiarity with software engineering processes.
- A sense of adventure.

A Brief Pop Quiz

- What is wrong with the following statement that attempts to describe a communication channel (port) comprised of 1024 sub-channels (virtual circuits):
  - “The port class has 1024 virtual-circuit classes.”

Pop Quiz--Continued

- Answer--The statement should say something like:
  - “A port object is comprised of 1024 virtual circuit objects.”

Administrative Trivia

- Exams:
  - Determination of need for exams will be based upon success of homework and project activities.
  - Homework/projects:
    - Two homework assignments
    - A small group project (mini-project)
    - A more substantial group project later in the semester
Collaboration Policy

- Students are encouraged to study together.
- Students are NOT to collaborate in the solution of homework, except in the context of group project teams.
- Detected homework collaboration penalties:
  - First offense: All parties equally share a single grade.
  - Second offense: Penalty up to and including expulsion from the course, depending on severity of the offense.

Course Overview

- Objectives:
  - To motivate the notions of object-oriented and component-based software and their growing importance to software engineering.
  - To understand the relationship between component-based design (CBD) and object-oriented design (OOD).
  - To understand the theoretical underpinnings of OOD and CBD.
  - To appreciate the role of modeling tools and patterns in OOD and CBD.
  - To study important CBD frameworks such as CORBA, COM/DCOM, and JavaBeans.

Small Group Project (Mini-project)

- Conducted prior to midterm.
- Objective: to insure that all students have a solid foundation in Object Oriented Design and sound software development practices.
- Will involve iterative development of a nontrivial application in Java.
- Students will work in teams of two.

Large Group Project

- Conducted during the last 4-5 weeks of the semester.
- Involves the development of a nontrivial system using sound OOD and CBD practices.
- Students will work in teams of 4-5.
What is Component-based Development?

- The assembly of new software systems (applications) from standard, reusable building blocks (components) and frameworks.
- Analogous to the development of digital systems from standard integrated circuit chips or standard design modules.
- Component software represents the maturation (industrialization) of software development practices.

The Need for CBD

- Commercial software development has been going on for about 50 years.
- For that entire time span, the demand for new software applications has outstripped the state-of-the-art for software development practices.
- As a result, the software industry has been in a perpetual state of crisis.

The Need for CBD--Continued

- Some traditional software problems
  - Large software systems are extremely complex
    - Large, monolithic systems are difficult to design and build and even harder to maintain.
    - Reliability and robustness is a major problem.
  - It is difficult to reuse assets--e.g. design knowledge, software--across multiple projects.
  - Tailoring of software products for different user requirements is difficult.

Software Maintenance Costs

- Changes in user requirements 41.8%
- Changes in data formats 17.6%
- Emergency Fixes 12.4%
- Routine Fixes 9%
- Documentation 6.2%
- Efficiency improvements 5.5%
- Other 3.4%

How Can Components Help?

- Components can provide robust, reusable building blocks for software applications.
- Components can come from independent sources and can be catalogued for future use.
- Component frameworks can provide standard architectural approaches for complex applications—e.g. distributed client-server systems.
- Components can provide a natural basis for upgrade and customization of software.

Component Software—A Warning

- CBD does not represent the elusive “silver bullet” for software engineering.
  - Even with component technologies, large software systems will remain extraordinarily complex and difficult to design, implement, and maintain.
  - CBD does not eliminate the need for sound software engineering practices and processes.
  - It should be noted that many of the claims currently being made for component technologies were earlier made for object-oriented technologies.

What is a “Software Component”

- A formal definition from 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.
- A more intuitive definition:
  - An independently deliverable unit of software that encapsulates its design and implementation and offers interfaces to the outside, by which it may be composed with other components to form a larger whole. (From: Objects, Components, and Frameworks with UML—The Catalysis Approach, by D. D’Souza and A. Wills, Addison Wesley, 1998.)

Some Important Aspects of the Preceding Definitions

- Composition
  - Components are building blocks
  - Components package deployable units of functionality (service).
  - Components can be “assembled” to form larger systems.
- Independence
  - Components should be independently deployable
    - self-contained
    - fully encapsulates its constituent behavior
Important Aspects of the Preceding Definitions--Continued

• Contractually Specified Interface(s)
  – Components can be accessed only via their interfaces
  – Interface nominally consists of a set of named operations (methods).
  – The semantics (behavior) of each operation must be precisely specified.
  – Interface can be specified as a “contract” between the component and a client.

• Encapsulation
  – All details of internal structure are hidden
  – Component is accessible only through its interface
  – Component is self-contained.
  – White-box vs. black-box visibility
    • White-box--User has access to source code of component.
    • Black-box--Component source code is not available.

• “…explicit context dependencies only…”
  – all requirements regarding the environment in which the component will function must be explicitly specified.
    • operating system
    • data representation
    • hardware
    • ...
  – Component frameworks such as CORBA, COM, and JavaBeans, provide a contextual environment--i.e. a component world.

So, How Does a Component Differ From a Class (or Object)?

• Granularity
  – Components are intended to be “independently deployable--hence they are fairly “coarse grained”.
  – Class hierarchies tend to package small, highly cohesive units of functionality.
• Focus on interfaces and composition rather than inheritance and class hierarchy.
• Focus on language and location transparency.
CBD Versus OOD-Some Additional Observations

• OOD does not necessarily lead to well-structured systems.
• Class hierarchies do not necessarily provide a good basis for software reuse.
• CBD is based upon an architectural perspective--i.e. conceiving a system in terms of its physical components and their interconnections.

A Little More About Composition

• Components may be “plugged together” in different ways:
  – Components can be integrated into larger applications by developing external software that utilizes and coordinates their services.
  – Generic components can be customized by “plugging in” subcomponents that provide various versions of the component’s functionality.

Composition--An Example
from D’Souza and Willis, Catalysis

Composition Example--Continued
Components--Another Example
Consider a simple text editor:

<table>
<thead>
<tr>
<th>Editor</th>
</tr>
</thead>
<tbody>
<tr>
<td>spellCheck( )</td>
</tr>
<tr>
<td>layout( )</td>
</tr>
<tr>
<td>addElement(…)</td>
</tr>
<tr>
<td>delElement(…)</td>
</tr>
</tbody>
</table>

Component Example--Continued
A possible component-based design of the editor:

<table>
<thead>
<tr>
<th>Editor</th>
</tr>
</thead>
<tbody>
<tr>
<td>spellCheck( )</td>
</tr>
<tr>
<td>layout( )</td>
</tr>
<tr>
<td>addElement(…)</td>
</tr>
<tr>
<td>delElement(…)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Editor Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>spellCheck( )</td>
</tr>
<tr>
<td>layout( )</td>
</tr>
<tr>
<td>maxsize</td>
</tr>
<tr>
<td>resize</td>
</tr>
<tr>
<td>children</td>
</tr>
</tbody>
</table>

Component Example Continued
The Spell Checker interface of the Editor Core:

<table>
<thead>
<tr>
<th>Spell checker</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC Interface</td>
</tr>
<tr>
<td>seq* Word</td>
</tr>
<tr>
<td>next word</td>
</tr>
<tr>
<td>replace word</td>
</tr>
</tbody>
</table>

Component Example Continued
The notion of pluggability (substitutability):

<table>
<thead>
<tr>
<th>Editor Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>spellCheck( )</td>
</tr>
<tr>
<td>SC Interface</td>
</tr>
<tr>
<td>seq* Word</td>
</tr>
<tr>
<td>next word</td>
</tr>
<tr>
<td>replace word</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E-mail Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acme Spell Checker</td>
</tr>
<tr>
<td>SpellzWell Spell Checker</td>
</tr>
</tbody>
</table>
What We Have Seen So Far:

- Motivation and rationale for CBD
- Basic definition of a component
- Basic relationship of CBD to OOD
- The need for precise specification of interfaces.

Interfaces and Specification

- Components connect and interact solely via their interfaces.
- An interface provides a set of named operations (methods) that can be invoked by clients.
- Some terminology:
  - Provider (supplier): The entity (component) that implements the operations of an interface.
  - Client (user): An entity that uses (invokes) the services provided by an interface.

Interface Specification

- The syntax (structure) and semantics (behavior) of each operation in an interface is given by a specification.
- This specification serves both the provider and the client.
  - Provider--specification defines the requirements for the behavior to be implemented within the provider component.
  - Client--specification defines the behavior that can be expected from interactions with the provider’s interface.

Aspects of a Specification

- Signature--specifies the syntactic nature of an operation.
  - name
  - parameters
  - types
- Functional (behavioral) specification--defines the computational behavior of the operation.
Aspects of a Specification--Continued

• Nonfunctional specification--defines other requirements:
  – performance requirements
  – storage requirements
  – other resource needs

Specification by Contract

• Contract--Explicit agreement that binds a provider and a client
  – States the rules that restrict use of a service (operation) by the client--preconditions
  – States the guaranteed result of legally invoking the operation--postconditions.
• The notion of contracts is due to Bertrand Meyer.

Contracts for Interface Specification

• Contracts can be stated in rigorous, mathematical language.
  – logical assertions
  – invariants
• Derived from abstract data type (ADT) concepts.
• Notion of contracts is widely used in software engineering methodologies.

Contracts--Pre- and Postconditions

• Consider an operation A
• A “correctness formula” for A can be expressed as a “Hoare triple”:
  – \{P\} A \{Q\}
  – Meaning: Any execution of A, beginning in a state where P holds, will terminate in a state where Q holds
  – P and Q are called assertions.
  • P is called the precondition for A.
  • Q is called the postcondition for A.
Correctness Formulae--Simple Examples and Observations

• A trivial correctness formula:
  – \{x \geq 9\} \ x := x + 5 \ \{x \geq 13\}
  – Why 13? Is this a typo? Why not 14?
  – A “stronger” postcondition: \{x \geq 14\}
  – A “weaker” precondition: \{x \geq 8\}

• A simple quiz:
  – Consider an advertisement for a job A, specified as \{P\} A \{Q\}. Would you rather have a job with a weak precondition or a strong one? What about the postcondition?

Weak versus Strong Conditions:

• Answer to the simple quiz:
  – Preconditions
    • Preconditions limit the conditions under which the employee must perform—i.e. the set of cases which the employee must handle.
    • Thus, a lazy employee would favor the strongest possible preconditions.
    • The strongest possible precondition is:
      – \{False\} A \{…\}

Preconditions and Postconditions--An Example

• Consider a stack class with operations:
  – int max( )—maximum allowed stack size.
  – int count( )—number of stack elements
  – bool empty( )—stack empty?
  – bool full( )—stack full?
  – item top( )—read top stack item.
  – void push( item i)—push item i onto stack.
  – void remove( )—remove top item from stack.
Example--Continued

- Preconditions:
  - before push(): stack must not be full
  - before remove(): stack must not be empty
  - before top(): stack must not be empty

- Postconditions:
  - after push(item i): stack must not be empty; top item on stack must be i; number of elements has been increased by one.
  - after remove(): stack must not be full; number of elements has been decreased by one.

Example--Continued

- Expressing pre-and post conditions as axioms:
  - In a postcondition, a variable x is referenced as follows:
    - x refers to postcondition value
    - x' refers to precondition value—i.e the old value of x
  - For values returned by observer methods, explicit assignments are used to capture pre and post state.
  - a mix of formal and informal expression is used.

Expressing Pre-and Postconditions

```java
interface Stack {
    int max();
    //pre true
    post result == maximum allowed stack size
    int count();
    //pre true
    //post 0 <= result <= this.max() and
    //result == number of items in stack
    boolean empty();
    //pre true
    //post result == (this.count() == 0)
    boolean full();
    //pre true
    //post result == this.max()
}
```

Expressing Pre and Postconditions--Continued

```java
item top();
// pre !this.empty()
// post result == top item on stack
void push(item i);
// [size: int ●
// pre size == this.count(); size < this.max(
// post !this.empty() and this.count() == (size + 1)
// and this.top() == i
// ]
void remove();
// [size:int ●
// pre size == this.count(); !empty()
// post !this.count() == this.max() and this.count == (size -1)
// ]
```
Interface Contracts--Another Example

- An interface for text manipulation. Operations:
  - int max( )--max. length of a text instance
  - int length( )--length of this text instance
  - char read (int pos)-- return char. at position pos
  - void write(int pos, char ch)--???
  - void delete (int pos): ???
  - void register (TextObserver x);
  - void unregister (textObserver x);

Example--Continued

```java
interface TextModel {
    int max( );
    // pre true;
    // post result = = maximum length this text instance can have
    int length( );
    // pre true
    // post 0 <= result <= this.max( ) and result = = length of text
    char read(int pos);
    // pre 0 <= pos < this.length( )
    // post result = = character at position pos
    (continued on next slide)
```

Example--Continued

```java
void write(int pos, char ch);
// [ len: int; txt: array of char
// pre len = this.length( ); (∀i : 0 <= i < len: txt[i] = this.read(i)) :
// len < this.max( ) and 0 <= pos <= len
// post this.length() = = len + 1
// and (∀i: 0 < i < pos: this.read(i) = = txt[i])
// and this.read(pos) = ch
// and (∀i:pos < i < this.length( ): this.read(i) = = txt[i-1])
// ]
(continued on next slide)
```

Example--Continued

```java
void delete (int pos);
// [len: int; txt: array of char
// pre len = this.length( ); (∀i : 0 <= i < len: txt[i] = this.read(i)):
// 0 <= pos < len
// post this.length() = = len -1
// and (∀i: 0 <= i < pos: this.read(i) = = txt[i]
// and (∀i: pos <= i < this.length( ): this.read(i) = = txt[i+1]);
// ]
void register(TextObserver x);
void unregister(TextObserver x);
}
```
**Invariants**

- Invariants are assertions that describe global properties that must always hold for an object or component.
- e.g. consider a Bank-Account class which maintains a deposits_list, withdrawals_list, and balance.
- An appropriate invariant for this class:
  - deposits_list.total - withdrawals_list.total = balance

**Invariants--Continued**

- Who is responsible for preserving invariants?
- Since an invariant must hold both prior to and following all operations, potentially strengthens both the preconditions and postconditions of operations
- Modified correctness formula:
  - \{Inv \text{ and } \text{pre}\} \text{ A } \{Inv \text{ and } \text{post}\}

---

**Invariants--Continued**

- What are some possible invariants for our Stack example?
  - 0 <= count (count is non-negative)
  - count <= maximum allowed size (count is bounded)
  - empty == (count == 0) (empty if no elements)

---

**Contracts--Some Problems**

- The notion of contractual interfaces is straightforward for procedural libraries and simple class libraries.
- However, some structures used in component frameworks complicate matters
  - callback mechanisms
  - object registration/notification mechanisms
  - re-entrancy and recursion
  - concurrency
What is a Callback?

- A callback is a reference to a procedure that is passed to (registered with) some entity (provider) by a client, usually associated with some event recognized by that entity.
- Then, this procedure is called whenever the event occurs.
- Callback mechanisms are common in windowing systems--e.g. X-windows/Motif

A Diagram of a Callback

Why are Callbacks Problematic?

- In a callback structure, the provider actually calls the client.
  - This violates the notion that an interaction with a provider is always viewed as an atomic event by the client.
  - With callbacks, state of the provider at any arbitrary point when the callback occurs may be revealed to the client.
  - Contract must guarantee “validity” of observable state of the provider for as long as any callbacks are active.

Observers--An OO Form of Callback

- Consider the TextModel example used earlier to motivate pre- and postconditions.
- Note that the TextModel interface includes methods to register and unregister TextObserver objects.
- Registered observers will be notified whenever an insertion or deletion of text takes place.
The TextObserver Interface

interface TextObserver {
    void insertNotification (int pos);
    // pre character at position pos has just been inserted
    void deleteNotification (int pos)
    // pre character previously at position pos has been deleted
}

Any object implementing this interface can be registered with a TextModel.

An example of such an observer object, might be a View object, responsible for managing the display of the text.

An Example View Interface

interface TextView extends TextObserver {
    TextModel text ( );
    // pre true
    // post result != null
    int CaretPos ( );
    // pre true
    // post 0 <= result <= this.text( ).length( )
    void setCaret (int pos);
    // pre 0 <= pos <= this.text( ).length( )
    // post this.caretPos( ) = pos
    int posToXCoord (int pos);
    // pre 0 <= pos <= this.text( ).length( )
    // post result = x-coord. corresponding to text position pos.
    // post result = x-coord. corresponding to text position pos.
    void rubout (char ch);
    // [caret:int •
    // pre caret = this.CaretPos( ); caret > 0
    // equiv this.text( ).delete(caret -1)
    // post this.caretPos( ) = = caret -1
    // ]
    ... similar for type ( )
Potential Problem with this Example

- TextView is invoked to update the view after a text has been changed.
- TextView will need to carry out several steps to complete the view update.
- During this time, there may be inconsistencies between displayed state and the actual state of the text.
- Which version of state should TextView present to outside entities—i.e. via calls to its mapping methods.