Introduction to Unified Modelling Language (UML)
(part 3- Dependency relationship, Component diagram)

Material covered in this lecture is based on various chapters from UML 2 and the Unified Process- 2nd Edition Practical Object Oriented Analysis & Design

CS 446/646 ECE452
May 9th, 2011

IMPORTANT NOTICE TO STUDENTS
These slides are NOT to be used as a replacement for student notes. These slides are sometimes vague and incomplete on purpose to spark a class discussion.
Dependency Relationship

What is a Dependency Relationship?

- a relationship between (UML) model elements, whereby change to one element impacts the other element(s)

Example

- object passing via method calls
- locally scoped variables
- class data fields?
  - Hmm ..... not really why?
Dependency Relationship

Common Types

- «use», «call», «parameter», «instantiate»

Notation
Realization Relationship

Class style notation

«interface» Borrowable
borrow() return() isOverdue()

Book CD

Lollipop style notation

Borrowable

Book CD
Realization Relationship

```
interface

«interface» Borrowable
borrow()
return()
isOverdue()

realization relationship

Book
CD

Borrowable

Book
CD
```
Required Connector

Library depends on Borrowable items

Library

«interface» Borrowable

Library

Borrowable

Library

Borrowable

Library requires Borrowable
Require/Provide Example

Observations

- Library is composed of
  - Book objects
  - CD objects
- Library requires `Borrowable` interface
- Book class provides `Borrowable` interface
- CD class provides `Borrowable` interface
Design Example

Inheritance based solution

Q1: are BorrowableItem & NonBorrowableItem the right super classes?
Q2: how can we accommodate another feature: such as XMLExportable for Book & Journal.

But journal is not borrowable

WATERLOO
CHERITON SCHOOL OF
COMPUTER SCIENCE
Design Example
Components

What is a Component?

- “a component represents a **modular** part of a system that **encapsulates its contents** and whose **manifestation is replaceable** within its environment”
  

- interfaces are **key** to component based development
  - WHY?

- Notation
Components

Properties

• can have attributes & operations
• can participate in association & generalization relationships
• can represent
  – an entity that can be instantiated at run-time
  – subsystems
Components

Observations (ask the class)
Sequence Diagrams

Intention

- show *interactions* between *life-lines* as *time-ordered sequence* of events
UC001: use case: AddCourse

**Brief Description:** Add details of a new course

**Primary actors:** Registrar

**Pre-Conditions:**
1. The registrar has logged on to the system

**Main Flow:**
1. The registrar selects “add course”
2. The registrar enters the name of the course
3. System creates a new course

**Post-Conditions:**
1. A new course has been added to the system

**Alternate flows:**
CourseAlreadyExists
The registrar selects "add course"

addCourse("cs446")

The system creates new course

```plaintext
sd AddCourse

:Registrar

:RegistrationManager

<<create>>

cs446:Course
```
Example

The registrar selects "add course"

The system creates new course

notes can describe the flow

return message

activation

object is created at this point

synchronous message

lifeline

object creation message

sd AddCourse

:Registerar

:RegistrationManager

addClassCourse("cs446")

«create»

cs446:Course

WATERLOO
CHERITON SCHOOL OF
COMPUTER SCIENCE
Sequence Diagram

Observations

- *time* running from top to bottom
- *lifelines* running from left to right
- abstraction of use case realization
- **activations** indicate when a lifeline has *focus of control*
  - *self delegation* → *nested activation*
  - maybe omitted if it complicates the diagram
- focus of control shifts with message call
  - → leads to nested focus of control
Sequence Diagram Destruction

```
sd DeleteCourse

:Registrar

:RegistrationManager

self delegation

findCourse("cs446")

nested activation

delecourse("cs446")

async. «destroy»
```

```
cs446:Course
```
State & Constraints

State

- state invariants on the lifeline
- useful for analysis

Order System Example

- State: \textit{unpaid} $\rightarrow$ \textit{paid} $\rightarrow$ \textit{delivered}
- Conditions:
  - order must be paid in full by a single payment
  - items can only be delivered after the payment has been made
  - items are delivered within 29 days
State & Constraints

cd ProcessOrder

:Customer

raiseorder() -> :OrderManager

acceptPayment() -> "create" :Order

{B-A <= 29 days}

A

:DeliveryManager

deliver()

B

delivered

delivered

acceptPayment() -> paid

acceptPayment() -> unpaid

del"create" :Order

acceptPayment() -> paid

acceptPayment() -> unpaid
State & Constraints

cd ProcessOrder

:Customer

raiseorder()

label

A

acceptPayment()

constraint

{B-A <= 29 days}

B

«create» :Order

unpaid

acceptPayment()

paid

deliver()

delivered

deliver()

:OrderManager

:DeliveryManager

state invariant
Conditional Execution

- opt creates a single branch
  - do this if condition is true
- alt creates multiple branches
  - do this if condition 1 is true
  - do this if condition 2 is true
  - do this otherwise
## Conditional Execution Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>opt</td>
<td>if..then</td>
</tr>
<tr>
<td>alt</td>
<td>if ..elseif ..else</td>
</tr>
<tr>
<td>loop</td>
<td><strong>loop min,max[condition]</strong>: loop min times, and then loop max times while condition is true</td>
</tr>
<tr>
<td>ref</td>
<td>the fragment refers to another interaction</td>
</tr>
<tr>
<td>par</td>
<td>parallel execution</td>
</tr>
<tr>
<td>critical</td>
<td>atomic (without interruption) execution</td>
</tr>
<tr>
<td>neg</td>
<td>invalid interactions (things that must not happen)</td>
</tr>
<tr>
<td>assert</td>
<td>only valid behaviour at that point in the interaction</td>
</tr>
</tbody>
</table>