16-Design

- Readings
  - OOAD Using the UML
    - Copyright 1994-1998 Rational Software, all rights reserved
    - [Partly] posted

Rational Unified Process

- The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of a software-intensive system
- A software development process defines Who is doing What, When and How in building a software product
- The Rational Unified Process has four phases: Inception, Elaboration, Construction and Transition
- Each phase ends at a major milestone and contains one or more iterations
- An iteration is a distinct sequence of activities with an established plan and evaluation criteria, resulting in an executable release
In RUP: Major Workflows Produce Models

- Business Modeling produces Business Model supported by Test.
- Requirements produce Use-Case Model realized by Test.
- Analysis & Design produce Design Model implemented by Implementation.
- Implementation produces Implementation Model verified by Test.

The RUP Iterative Model

- Process Workflows:
  - Inception
  - Elaboration
  - Construction
- Supporting Workflows:
  - Configuration Mgmt
  - Management Environment

Workflows group activities logically.

In an iteration, you walk through all workflows.
Requirements Workflow

- System Analyst
  - Develop Vision
  - Manage Dependencies
- Use-Case Specifier
  - Elicit Stakeholder Needs
  - Capture a Common Vocabulary
- User-Interface Designer
  - Find Actors and Use Cases
  - Structure the Use-Case Model
- Architect
  - Prioritize Use Cases
- Reviewer
  - Review Requirements
- Review the Design
- Design Reviewer

Analysis & Design Workflow

- Architect
  - Architectural Analysis
  - Review the Architecture
- Designer
  - Use-Case Analysis
  - Review the Use-Case Design
- Database Designer
  - Database Design

©Rick Adrion 2005 (except where noted)
Implementation Workflow

1. Architect
   - Structure the Implementation Model

2. System Integrator
   - Plan System Integration
   - Integrate System

3. Implementer
   - Plan Subsystem Integration
   - Implement Classes
   - Fix a Defect
   - Perform Unit Test
   - Integrate Subsystem

4. Code Reviewer
   - Review Code

Test Workflow

1. Test Designer
   - Plan Test
   - Design Test
   - Implement Test
   - Evaluate Test

2. Integration Tester
   - Execute Integration Test

3. System Tester
   - Execute System Test

4. Performance Tester
   - Execute Performance Test

5. Designer
   - Design Test Classes and Packages

6. Implementer
   - Implement Test Components and Subsystems
A Minimal Iterative Process

Getting Started: (do this once)
1. Capture the major functional and non-functional requirements for the system.
   - Express the functional requirements as use cases, scenarios, or stories.
   - Capture non-functional requirements in a standard paragraph-style document.
2. Identify the classes which are part of the domain being modeled.
3. Define the responsibilities and relationships for each class in the domain.
4. Construct the domain class diagram.
   - This diagram and the responsibility definitions lay a foundation for a common vocabulary in the project.
5. Capture use case and class definitions in an OO CASE tool (e.g., Rose) only when they have stabilized.
A Minimal Iterative Process

Getting Started: (do this once)

6. Identify the major risk factors and prioritize the most architecturally significant use cases and scenarios.
   • It is absolutely imperative that the highest risk items and the most architecturally significant functionality be addressed in the early iterations. You must not pick the “low hanging fruit” and leave the risks for later.

7. Partition the use cases/scenarios across the planned iterations.

8. Develop an Iteration plan describing each “mini-project” to be completed in each iteration.
   • Describe the goals of each iteration, plus the staffing, the schedule, the risks, inputs and deliverables.
   • Keep the iterations focused and limited (2-3 weeks per iteration). In each iteration, conduct all of the software activities in the process: requirements, analysis, design, implementation and test.

For each iteration: (repeat until done)

1. Merge the functional flow in the use cases/scenarios with the classes in the domain class diagram
   • Produce sequence (and collaboration) diagrams at the analysis level.

2. Test and challenge the sequence diagrams on paper, or whiteboard
   • Discover additional operations and data to be assigned to classes
   • Validate the business process captured in the flow of the sequence diagram

3. Develop statechart diagrams for classes with “significant” state
   • Statechart events, actions, and most activities will become operations on the corresponding class

4. Enhance sequence diagrams and statechart diagrams with design level content
   • Identify and add to the class diagram and sequence diagrams any required support or design classes (e.g. collection classes, GUI and other technology classes, etc.)

5. Challenge the sequence diagrams on paper/whiteboard, discovering additional operations and data assigned to classes.
Views & Workflows

- Architectural Analysis
- Use Case Analysis
- Subsystem Design
- Deployment Design
- Database Design

Use-Case View
- Use Case
- Class
- Use-Case Design
- Class Design
- Use-Case Analysis

Worker Responsibilities

- Architect
- Designer
- Database Designer
- Software Architecture Document
- Design Model
- Data Model
- Use Case Realization
- Package/Subsystem
- Class
- Architecture Reviewer
- Design Reviewer
So where do we start?

Use Case Analysis Overview

Use-Case Model

Supporting Documents
- Architecture Document
- Glossary
- Supplemental Specs

Use-Case Realization

Analysis Classes

Analysis Model

Design Model
Use Case Analysis Steps

- Supplement the Descriptions of the Use Case
- For each use case realization
  - Find Classes from Use-Case Behavior
  - Distribute Use-Case Behavior to Classes
- For each resulting analysis class
  - Describe Responsibilities
  - Describe Attributes and Associations
  - Qualify Analysis Mechanisms
- Unify Analysis Classes

What is an Analysis Class?

- Early conceptual model
- Functional requirements
- Model problem domain
- Likely to change
  - Boundary
  - Information used
  - Control logic
The Roles

Collaboration Diagram

- **Boundary Class**: Model interaction between the system and its environment
- **Control Class**: Coordinate the use case behavior
- **Entity Class**: Store and manage information in the system

Customer

Example: Entity & Control Classes

- Course (from University Artifacts)
- CourseOffering (from University Artifacts)
- Grade (from University Artifacts)
- Student (from University Artifacts)
- Professor (from University Artifacts)
- Schedule (from University Artifacts)
- RegistrationController (from Registration)
- CloseRegistrationController (from Registration)
- MaintainStudentController (from Registration)
- MaintainProfessorController (from Registration)
- SelectCoursesToTeachController (from Registration)
- ReportCardController (from Student Evaluation)
- SubmitGradesController (from Student Evaluation)
Describe Responsibilities

- What are responsibilities?
- How do we find them?

Class Responsibilities from a Collaboration Diagram

Register for Courses use case

1: // select maintain schedule()
2: // open schedule form()
3: // select 4 primary and 2 alternate offerings()
4: // get course offerings() // open()
5: // select 4 primary and 2 alternate offerings()
6: // add courses to schedule()
7: // create with offerings() // create with offerings()

Non-functional requirements
- Class should have multiple responsibilities

First cut at class operations
- Actions that object can perform
- Knowledge object maintains
- Non-functional requirements
Class Responsibilities from a Sequence Diagram

Register for Courses use case

What are Roles?

- The “face” that a class plays in the association
Example: Finding Relationships

- MaintainScheduleForm does not make any sense outside of the context of a particular use session.
- Only one MaintainScheduleForm can be active at any one time, or none may be active.

```csharp
// select maintain schedule
<<boundary>>
MainForm

+ // open

+ // select 4 primary and 2 alternate offerings

<<boundary>>
MaintainScheduleForm

// get course offerings

<<control>>
CourseCatalogSystem

// add courses to schedule

// get course offerings

<<entity>>
Schedule

// create with offerings

5: // select 4 primary and 2 alternate offerings

1: // select maintain schedule

View of Participating Classes (VOPC) diagram.

What is a Use-Case Realization?

Use Case Model

Use Case

<<realizes>>

Use Case Realization

Use Case Realization Documentation

Design Model

Use Case Realization

Sequence Diagrams

Collaboration Diagrams

Class Diagrams
Alternatives

• RUP begins with Analysis classes we found by analyzing Collaboration & Sequence Diagrams (these derived from the Use Cases during Use Case Analysis), then is refined by defining Operations, States, Attributes, Associations and Generalizations (in Class Design)

• Some other approaches:
  • Noun Phrase Approach
  • Common Class Patterns
  • CRC (Class-Responsibility-Collaboration)

Noun Phrase Approach

• Examine the requirements and underline each noun
  • Each noun is a candidate class
• Divide list of candidate classes into
  • Relevant Classes
    • Part of the application domain; occur frequently in reqs
  • Irrelevant Classes
    • Outside of application domain
  • Fuzzy Classes
    • Unable to be declared relevant with confidence; require additional analysis
• Experience will eventually enable designers to avoid generating irrelevant classes
Find Classes from requirements

• Consider the following University Enrollment system specification
  • each university major has a number of required courses and a number of elective courses.

---

Classes, Relationships & Attributes

• A course can be part of any number of majors
• Each major specifies minimum total credits required
• Students may combine course offerings into programs of study suited to their individual needs and leading to the degree/major in which enrolled
Noun Phrase Approach

- may help in identifying domain objects
- not good at identifying objects that live in the application domain
- Thus, it can help at the beginning of analysis, but you will not return to it as you move into design
- Finding good objects during design means identifying abstractions that are part of your application domain and its execution machinery
- Objects that are part of your application domain will have a tenuous connection, at best, to real-world things
  - e.g. what’s the correspondence of a scrollbar to the real world

Common Class Patterns

- Derive classes from the generic classification theory of objects
  - Concept class
    - a notion shared by a large community
  - Events class
    - captures an event that demarks intervals within a system
  - Organization class
    - a collection or group within the domain
  - People class
    - roles people can play
  - Places class
    - a physical location relevant to the system
Common Class Patterns

- Rumbaugh proposed a different scheme
  - Physical Class (Airplane)
  - Business Class (Reservation)
  - Logical Class (FlightTimeTable)
  - Application Class (ReservationTransaction)
  - Computer Class (Index)
  - Behavioral Class (ReservationCancellation)
- These taxonomies are meant to help a designer think of classes, however it is difficult to be systematic
- Probably only useful during early analysis

CRC Cards

- CRC = Candidates, Responsibilities, Collaborators
- Meant primarily as a brainstorming tool for analysis and design
- In place of use case diagrams ⇒ use index cards
- In place of attributes and methods ⇒ record responsibilities

See Object Design by Wirfs-Brock and McKeen, © 2003
Index Cards

- On the unlined side of the index card
  - write an informal description of each candidate class’ purpose and role

- On the lined side of the index card
  - identify responsibilities and collaborators

<table>
<thead>
<tr>
<th>Document</th>
<th>←-candidate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose:</td>
<td>A Document acts as a container for graphics and text</td>
</tr>
<tr>
<td>Role: Container</td>
<td></td>
</tr>
<tr>
<td>Pattern: Composite</td>
<td></td>
</tr>
</tbody>
</table>

MVC using CRC cards

View
- Render the Model
- Transform coordinates

Controller
- Interprets user input
- Distributes control

Model
- Maintains problem related information
- Broadcasts change notification
Not Just Index Cards

- Post-It Notes can be used for even less “structure”;
- might be easier when brainstorming

Document
Purpose: A document
Represents a container
that holds text and/or
graphics that the user
can enter and visually
arrange on pages

Why index cards?

- Forces you to be concise and clear and focus on major responsibilities since you must fit everything onto one index card
- Inherent Advantages
  - cheap, portable, readily available, and familiar
  - gives people a "feel" for the design
  - can propose and test changes to the design rapidly (all you have to do is make new cards)
  - focus on responsibilities as opposed to "n:m attribute" design as promoted by OMT, Booch, etc
- affords Spatial Semantics…
  - close collaborators can be overlapped
  - vertical dimension can be assigned meanings
  - abstract classes and specializations can form piles …which provides benefits
  - Beck and Cunningham report that they have seen designers talk about a new card by pointing at where it will be placed
So Where Are We?

Architectural Design Overview

- Design and Implementation Mechanisms
- Design Classes and Subsystems
- Reuse Opportunities
- Refined Architectural Layers and Partitions
In analysis, we had one application with many different forms …

During design, some analysis classes may be split, joined, removed, etc.

In design, the one application becomes three applications, each with it’s own forms …
Classes & packages

• What is a class?
  • A description of a set of objects that share the same responsibilities, relationships, operations, attributes, and semantics.

<table>
<thead>
<tr>
<th>Class Name</th>
</tr>
</thead>
</table>

• What is a package?
  • A general purpose mechanism for organizing elements into groups
  • A model element which can contain other model elements

<table>
<thead>
<tr>
<th>Package Name</th>
</tr>
</thead>
</table>

Packages Vs. Subsystems

• Packages provide no behavior
• Packages are simply containers of things which provide behavior
• Packages help organize and control sets of classes that are needed in common, but which aren’t really subsystems
• Dependencies are on specific elements within the Package

• Subsystems provide behavior, packages do not
• Subsystems completely encapsulate their contents
• Dependencies are on the interface of the subsystem
• Subsystems are easily replaceable

Encapsulation is the key! But note for packages dependencies should be on public classes
Design Classes and Subsystems

- Identifying Design Classes
  - analysis class is simple and already represents a single logical abstraction -> design class
  - entity classes survive relatively intact into design.

- Identifying Subsystems
  - analysis class is complex, such that it appears to embody behaviors that cannot be the responsibility of a single class acting alone, or the responsibilities may need to be reused, the analysis class should be mapped to a subsystem
  - may take a few iterations to stabilize.

- Analysis classes which evolve into subsystems might include:
  - complex services and/or utilities
  - user interfaces and external system interfaces.

Layering

- Concentrate on encapsulating change
- Package dependencies are not transitive, thus one layer can shield another from change
- Upward dependencies should be resolved in design
  - e.g., call backs can be replaced with the “subscribes to” association whose source is a class (called the subscriber) and whose target is a class (called the publisher)
    - subscriber specifies a set of events and is notified when one of those events occurs in the target
Layering Guidelines

- **Visibility**
  - Dependencies only within current layer and below

- **Volvatility**
  - Upper layers affected by requirements changes
  - Lower layers affected by environment changes

- **Generality**
  - More abstract model elements in lower layers

- **Number of layers**
  - Small system: 3 layers
  - Complex system: 5-7 layers

**Goal is to reduce coupling and to ease maintenance effort**
Describe Concurrency

• Concurrency Requirements driven by:
  • degree to which the system must be distributed
  • degree to which the system is event-driven
  • computational intensity of key algorithms
  • degree of parallel execution supported by the environment

• Modeling Processes map on independent threads of control supported by environment
  • Processes - stand-alone, heavyweight flow of control that may be divided into individual threads
  • Threads - lightweight flow of control which run in the context of an enclosing process

• Mapping Processes onto the Implementation Environment
• Distributing Model Elements Among Processes

Example

Mapping Design Elements to Processes
Example

Class Diagram

Component Diagram

Mapping to Implementation

Describe Distribution Overview
Why Distribute?

- Reduce processor load
- Special processing requirements
- Scaling concerns
- Economic concerns

Distribution Patterns
- Client/Server
  - 3-tier
  - Fat-Client
- Web Application
- Distributed Client/Server
- Peer-to-peer

Distribution patterns

- Common services
  - Presentation Services
    - UI including, the visual appearance of output and how user input is handled
  - Business Services
    - Business rules and logic
  - Data Services
    - Data relationships, efficiency of storage, and data integrity
- Patterns
  - One-Tier
  - Two-Tier
    - Fat Client -- client has its presentation and business services; server has the data services
    - Thin Client -- client has the presentation services; server has the business and data services
  - Three-tier
    - client has presentation services; server has business services; separate (logical) server has data services.
  - Web-tier
    - client accesses a web server that at least handles presentation services; web server may have its own business and data services or it may utilize one or more servers that handle business and data services
Deployment Model Modeling Elements

- **Node**
  - Physical run-time computational resource
- **Processor**
  - Execute system software
- **Device**
  - Support devices
  - Typically controlled by a Processor
- **Connection**
  - Communication mechanisms
  - Physical medium
  - Software protocol

Deployment Diagrams
Process-to-Node Allocation

- Dial up access and behind campus firewall

- External Desktop PC
  - StudentApplication
  - <<Internet>>
  - <<Campus LAN>>

- Desktop PC
  - StudentApplication
  - ProfessorApplication
  - RegistrarApplication
  - <<Campus LAN>>
  - <<Legacy>> Billing System

- Registration Server
  - CourseCatalogSystemAccess
  - CourseRegistrationProcess
  - GradeSubmissionProcess
  - TeachingCoursesSelectionProcess
  - ProfessorMaintenanceProcess
  - StudentMaintenanceProcess
  - CloseRegistrationProcess
  - ReportCardProcess
  - FinanceSystemAccess
  - <<Campus LAN>>
  - <<Legacy>> Course Catalog

- <<Legacy>> Course Catalog
- <<Internet>>
- <<Campus LAN>>
- <<Campus LAN>>
- <<Campus LAN>>

Design Distribution Pattern: Proxy

- ProxyDistributedController
  - 0..*
  - currentUser
  - Naming (from java.rmi)

- RemoteDistributedController
  - 0..*
  - currentUser

- <<Interface>> SecureUser (from Secure Interfaces)

- <<controller>> RegistrationController (from Student Activities)
  - 1

- RemoteRegistrationController (from Student Activities)
  - 0..1

Client
Server
secure user instance is created on the client and passed to the server when the remote controller is created

©Rick Adrion 2005 (except where noted)
Design Distribution Pattern: Proxy

RemoteTimecard
Lookup("RemoteTimecardController")

SomeForm
- ProxyDistributedController
- Naming
- RemoteTimecardController

1: new(SecureUser)
2: lookup(String)
3: new()
4: setSession(SecureUser)
5: DoSomething
6: DoSomething

The connection between the proxy and remote controller is established when the proxy controller is created.
The current user context is passed to the server for later access checks.
All calls to the proxy controller are forwarded to the remote controller.

Affect on Process Model Associations

MainStudentForm
(from Student Interface)
MaintainScheduleForm
(from Student Interface)

StudentApplication

RegistrationController
(from Student Activities)

CourseCatalogSystemAccess
(from CourseCatalog)
CourseCache
(from Course)
CourseCache
(from University Artifacts)
CourseOffering
(from University Artifacts)

©Rick Adrion 2005 (except where noted)