Objectives

• Describe EAMF Application and Information Perspectives and Related Activities
Roadmap

EAMF Perspectives
- Information Perspective
- Application Perspective
- Conceptual Technology Architecture (CTA)
- Architectural Style
- Reference Architecture/Element
- ABASs
- Architectural Mechanisms
- Patterns
- Framework
- Architectural Pattern
- Design Pattern
- Idiom

Information Perspective

Table used to summarize the artifacts captured and stored in the information perspective

Viewpoints for information perspective
Application Perspective: Conceptual Technology Architecture (CTA)

- The CTA is the counterpart of problem specific part of CBA diagram in the application perspective
- CTA diagram visualizes the high level computer processes, data stores, network assets without specific classifications
- The actors’ interaction with the technology would be depicted and explained
- CTA will not show specific software or hardware equipment or logical layering of any software components if this information is not known upfront

Table used to summarize the artifacts captured and stored in the application perspective
Application Perspective: Viewpoints

• Technology viewpoints are described differently by different resources
• EAMF combines the viewpoints of the industry think-tanks and defines a combined set as the following

<table>
<thead>
<tr>
<th>Microsoft’s Enterprise Solution Patterns’ Viewpoints</th>
<th>Gartner’s Expanded Architecture Framework’s Viewpoints</th>
<th>PDA EAF Viewpoints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>Data</td>
<td>Data</td>
</tr>
<tr>
<td>Application</td>
<td>Application</td>
<td>Application</td>
</tr>
<tr>
<td>Integration</td>
<td>Integration</td>
<td>Integration</td>
</tr>
<tr>
<td>Point of Access</td>
<td>Point of Access</td>
<td>Point of Access</td>
</tr>
<tr>
<td>Deployment</td>
<td>n/a</td>
<td>Moved to Deployment Mapping</td>
</tr>
<tr>
<td>Infrastructure</td>
<td>n/a</td>
<td>Moved to Infrastructure Mapping</td>
</tr>
</tbody>
</table>

Application Perspective: Viewpoints (cont.)

• Data
  • Data viewpoint shows patterns, pattern clusters and artifacts that relate to access and storage of persistent data

• Application
  • Application viewpoint shows patterns, pattern clusters and artifacts that relate how application logic processes data internally

• Integration
  • Integration viewpoint shows patterns, pattern clusters and artifacts that relate how a solution accesses external systems

• Point Of Access
  • Point of Access viewpoint shows patterns, pattern clusters and artifacts that relate how external systems and users access the solution
Use of UML Artifacts to Construct EAMF Views

<table>
<thead>
<tr>
<th>Diagram Name</th>
<th>Type</th>
<th>EAMF Disciplines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case</td>
<td>Dynamic</td>
<td>Analysis</td>
</tr>
<tr>
<td>Class</td>
<td>Static</td>
<td>Analysis</td>
</tr>
<tr>
<td>Activity</td>
<td>Dynamic</td>
<td>Analysis</td>
</tr>
<tr>
<td>State Machine</td>
<td>Dynamic</td>
<td>Analysis</td>
</tr>
<tr>
<td>Object</td>
<td>Static</td>
<td>Analysis</td>
</tr>
<tr>
<td>Package</td>
<td>Static</td>
<td>Analysis</td>
</tr>
<tr>
<td>Composite Structure</td>
<td>Static</td>
<td>Analysis</td>
</tr>
<tr>
<td>Deployment</td>
<td>Static</td>
<td>Analysis</td>
</tr>
<tr>
<td>Interaction Overview</td>
<td>Dynamic</td>
<td>Design</td>
</tr>
<tr>
<td>Sequence</td>
<td>Dynamic</td>
<td>Design</td>
</tr>
<tr>
<td>Communication</td>
<td>Dynamic</td>
<td>Design</td>
</tr>
<tr>
<td>Timing</td>
<td>Dynamic</td>
<td>Design</td>
</tr>
<tr>
<td>Component</td>
<td>Static</td>
<td>Design</td>
</tr>
</tbody>
</table>

Roadmap

Using the EAMF Application Perspective
• Sample CTA Diagram
• Identifying/Confirming RA
• Application Model Catalog Relationships
• Working with Application Model Catalogs
• Sample Logical Architecture Analysis Diagram
• Identification of Applicable Patterns
• Sample Guidelines
• Sample Design Considerations
• Sample Design Model
• Identification of Reference Implementation
• Product Mapping
Identifying/Confirming RA

Sample CTA Diagram
Working with Application Model Catalogs

- The application model catalog contains a list of the reference architectures
- For each reference architecture in the application model catalog there is a:
  - Document which explains the reference architecture in detail. The document may contain many links to more detailed explanations, graphics and tables
  - Model matrix which identifies the related patterns for that reference architecture
  - Capability matrix which lists the functional capabilities of the given reference architecture
  - A capability matrix for a reference architecture (Application Server) looks like the following:

```
<table>
<thead>
<tr>
<th>Capability 1</th>
<th>Capability 2</th>
<th>Capability 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature A</td>
<td>Feature B</td>
<td>Feature C</td>
</tr>
<tr>
<td>Feature D</td>
<td>Feature E</td>
<td>Feature F</td>
</tr>
</tbody>
</table>
```

Sample Logical Architecture Analysis Diagram

Identification of Applicable Patterns

- NF policies identified in the CR-Matrices are used to determine appropriate styles and patterns for the solution
- Not all non-functional capabilities lead to software design patterns
- Some non-functional capabilities such as reliability, availability, recoverability and dependability (and possibly others) require hardware deployment patterns along with organizational behavior changes towards quality
- There is no (bullet-proof) defined process that would help an architect to identify appropriate patterns
  - A pattern can be applicable to a certain problem but it may not be appropriate
  - The identification of the applicable and appropriate pattern requires immense working pattern knowledge which is not only knowing and understanding what patterns are but also recognizing when and when not to use certain patterns
- NFR Framework Based Approach is suggested
Sample Guidelines

**Policy Type**

**Motivation**
- ELR Product: Low Volume/Low Frequency - C2B Request/Response

**Measureable**
- Concrete (based on proven design)
- Measurable (only verifiable after implementation)

**Concrete**
- Business Driven
  - Required (resulting from analysis)
- Technology Driven
  - Desired (resulting from design considerations)

**Technology Driven**
- Desired (resulting from design considerations)

**Extensibility**
- Same generation process for different algorithms will be supported.

**Efficiency**
- Time: 8 sec per request per user

**Scalability**
- 20 concurrent users to 40 concurrent users without software modifications

**Readability**
- Follow company standards

**Availability**
- 6:00 am - 6:00 pm Business days

**Security**
- Only authenticated and authorized AFLAC employees

**Concurrency**
- Persistent State

**Transaction**
- Security

**Testability**
- Provide test harness, debugging and adjustable levels of logging capabilities.

**Naming and Directory**

---

**Sample Design Considerations**

- **Sample Steps: Generation of a new group number**
- **Steps are:**
  - Read the last used group number from the persistence store
  - Calculate the next group number based on an algorithm
  - Validate the calculated group number
    - Check that it does not include offensive words
    - Check that it has never been used before
  - If the newly calculated group number is not valid go to step 2. Otherwise, update the last used group number persistence store with the newly calculated group number
  - Return the newly calculated group number to the client

---

**Diagram:**

- Start
- Last Group Number = read Last Group Number from Data Store
- New Group Number = Calculate following the algorithm based on Last Group Number
- Offensive?
  - Yes: New Group Number = Unique?
  - No: Update Data Store with New Group Number
- Return New Group Number
- End
Sample Design Considerations:
Improvement on Efficiency.Time

- New Group Number: Offensive

- Offensive Words List = read Offensive Words List from data store

- Offensive Word = next word in the list

- Contains = new Group number contains offensive word

- More words in the list?:
  - No
  - Yes

- I/O operation
- CPU operation

- Repeated as many times as the number of words in the list.

The longer the list, the more time consuming is the process.

Sample Design Considerations:
Suggested Synchronous Layer
Sample Design Considerations:
Suggested Asynchronous Layer

Sample Design Considerations:
Standard Application Model Matrix
Sample Design Model

ud AIM System Partial Use Cases

AIM Team Member

New Account Setup

Team Member

AIM System Boundary

Setup New Account

Get New Group Number

<include>

Sample Design Model (cont.)

Get Group Number Activity

Get Group Number from the Cache

cache table

GroupNumber

get group number from the cache table

update read group number record as reserved with a timestamp

put read flag message into MoM

Group Number returned

Update Group Numbers Cache

get completeTo count

completeTo <= 0

stop update cache flow

load OffensiveWordsList (Lazy Load in Singleton)

insert completeTo new numbers into cache

eliminate used reserved numbers from cache

cache updated

No

Yes

ad get completeTo count

load cache update properties (Lazy Load in Singleton)

unreserved threshold

unreserved count

difference > 0

completeTo = full count - threshold + difference

No

Yes

ad insert completeTo new numbers into cache

read Last Used Group Number

get completeTo New Group Numbers

new group number

insert new group number into cache

new group number

update Last Used Group Number with Last Group Number in the List

new group numbers list cache

new group numbers list cache

insert new group number into cache

new group numbers list cache
Identification of Ref. Implementation

Product Mapping
Roadmap

EAMF-Compliant A&D Notations
- Architecture Analysis and Design Block Diagrams (Iconic Notations)
- BPMN Notations
- UML 2.0
- Reference Architectures Specific Notations
  - CWML
  - Service Oriented A&D Notations

Refined Application Architecture Blueprint
(e.g., virtual classroom environment)
Sample Logical Architecture Diagram
(e.g., virtual classroom environment)

Procedural Knowledge Road Map

<table>
<thead>
<tr>
<th>Type of knowledge/Levels</th>
<th>Conceptual model</th>
<th>Deployment methodology</th>
<th>Interoperability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Instance level</td>
<td>Process instance concept</td>
<td>BPMN+</td>
<td></td>
</tr>
<tr>
<td>Process definition level</td>
<td>Process definition concept WfMC meta-model+</td>
<td>XPDL+</td>
<td></td>
</tr>
</tbody>
</table>

- BPMN - Business Process Modeling Notation, BPMI
- XPDL - XML Process Definition Language, WfMC
- Wf-XML - Workflow Interoperability Binding, WfMC
Process Definition - An Example

Possible Process Execution
UML

• The UML emphasizes a graphical notation, called diagrams, for representing models
  • UML Models help specifying the artifacts and interactions of a software system
• UML provides little to no guidance on when and how to use UML diagrams
  • This is an area where EAMF helps as it describes the kinds of project artifacts, such as diagrams, needed during architectural disciplines and helps put them in the context of an overall project plan
• UML helps formalize architectural models by providing a precise syntax and semantics

Review of UML 1.x

• UML 1.x defines twelve types of diagrams, divided into three categories
• Four diagram types represent static application structure:
  • Class Diagram
  • Object Diagram
  • Component Diagram
  • Deployment Diagram
• Five represent different aspects of dynamic behavior
  • Use Case Diagram
  • Sequence Diagram
  • Activity Diagram
  • Collaboration Diagram
  • Statechart Diagram
• Three represent ways you can organize and manage your application modules
  • Packages
  • Subsystems
  • Models
• See the UML 1.0 Introduction given under “Additional Online Resources” for a detailed overview of the various UML modeling notations
Introducing UML 2.0

• UML 2 defines 13 basic diagram types divided into two sets:
  • Structural Modeling Diagrams
    • Package Diagram
    • Class of Structural Diagram
    • Object Diagram
    • Composite Structure Diagram
    • Component Diagram
    • Deployment Diagram
  • Behavioral Diagrams
    • Use Case Diagram
    • Activity Diagram
    • State Machine Diagram
    • Communication Diagram
    • Sequence Diagram
    • Timing Diagram
    • Interaction Overview Diagram

• See the UML 2.0 tutorial given under “Additional Online Resources” for a detailed overview of the various UML modeling notations

Structural Modeling Diagrams

• Package diagrams
  • Used to divide the model into logical containers or 'packages' and describe the interactions between them at a high level

• Class or Structural diagrams
  • Define the basic building blocks of a model: the types, classes and general materials that are used to construct a full model

• Object diagrams
  • Show how instances of structural elements are related and used at run-time
**Structural Modeling Diagrams (cont.)**

- **Composite Structure diagrams**
  - Provide a means of layering an element's structure and focusing on inner detail, construction and relationships

- **Component diagrams**
  - Used to model higher level or more complex structures, usually built up from one or more classes, and providing a well defined interface

- **Deployment diagrams**
  - Show the physical disposition of significant artifacts within a real-world setting

---

**Behavioral Modeling Diagrams**

- **Use Case diagrams**
  - Used to model user/system interactions. They define behavior, requirements and constraints in the form of scripts or scenarios

- **Activity diagrams**
  - Have a wide number of uses, from defining basic program flow, to capturing the decision points and actions within any generalized process

- **State Machine diagrams**
  - Essential to understanding the instant to instant condition or "run state" of a model when it executes

- **Communication diagrams**
  - Show the network and sequence of messages or communications between objects at run-time during a collaboration instance
Behavioral Modeling Diagrams (cont.)

• Sequence diagrams
  • Closely related to Communication diagrams and show the sequence of messages passed between objects using a vertical timeline

• Timing diagrams
  • Fuse Sequence and State diagrams to provide a view of an object's state over time and messages which modify that state

• Interaction Overview diagrams
  • Fuse Activity and Sequence diagrams to provide allow interaction fragments to be easily combined with decision points and flows

Visual Modeling with UML 2.0
Use of UML Artifacts to Construct EAMF Views

<table>
<thead>
<tr>
<th>Diagram Name</th>
<th>Type</th>
<th>EAMF Disciplines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case</td>
<td>Dynamic</td>
<td>Analysis</td>
</tr>
<tr>
<td>Class</td>
<td>Static</td>
<td>Analysis</td>
</tr>
<tr>
<td>Activity</td>
<td>Dynamic</td>
<td>Analysis</td>
</tr>
<tr>
<td>State Machine</td>
<td>Dynamic</td>
<td>Analysis</td>
</tr>
<tr>
<td>Object</td>
<td>Static</td>
<td>Analysis</td>
</tr>
<tr>
<td>Package</td>
<td>Static</td>
<td>Analysis</td>
</tr>
<tr>
<td>Composite Structure</td>
<td>Static</td>
<td>Analysis</td>
</tr>
<tr>
<td>Deployment</td>
<td>Static</td>
<td>Analysis</td>
</tr>
<tr>
<td>Interaction Overview</td>
<td>Dynamic</td>
<td>Design</td>
</tr>
<tr>
<td>Sequence</td>
<td>Dynamic</td>
<td>Design</td>
</tr>
<tr>
<td>Communication</td>
<td>Dynamic</td>
<td>Design</td>
</tr>
<tr>
<td>Timing</td>
<td>Dynamic</td>
<td>Design</td>
</tr>
<tr>
<td>Component</td>
<td>Static</td>
<td>Design</td>
</tr>
</tbody>
</table>

UML 2.0 Domains

• UML defines the notation and semantics for the following domains:
  • User Interaction or Use Case Model
  • Interaction or Communication Model
  • State or Dynamic Model
  • Logical or Class Model
  • Physical Component Model
  • Physical Deployment Model

• UML 1.x used use case, logical, process, implementation, and deployment views instead of domain models
  • Views used specific (sets of) UML 1.x diagrams to look at overall models from various angles
Maintaining Models
Consistency/Coverage

Component/Composite/Workflow
Business Services
Sample SOAD Interaction Model

Roadmap

Sample Enterprise Reference Architectures
- Information Management Aspects
- Service Management Aspects
- Service vs. Component Management
Information Management Aspects

• Front-End Aspects
  • Consumption
    • Search/Query, Portal, Analytics
  • Management
    • Modeling, Profiling, Indexing, Quality
  • Availability
    • Federation, ETL, Replication, Event
  • Collaboration
    • Workflow, Version Control
  • Security
    • Single Sign-On, Audit, Access Control

• Infrastructure Aspects
  • Heterogeneous Data Sources Access
    • Text/XML files, spreadsheets, messages, relational data, content data
  • Metadata Management & Storage

Service Management Aspects
Service vs. Component Management

Roadmap

Traditional Object Oriented Architecture Analysis & Design Approach (Review)
- From Requirements to Analysis
- Analysis Class
- Use Case Realization
- Example: Pay Invoice Use Case
- Pay Invoice Use Case: Flow of Events
- Analysis Class Diagram Realizing a Pay Invoice Use Case
- Collaboration Diagram Realizing a Pay Invoice Use Case
- Analysis and Design Workflow
- Summary: Architecture Engineering Steps
- Summary: Architecture Engineering Artifacts
From Requirements to Analysis

• Requirements specify the behavior of the largest-grained component: the system
  • Specifies the behaviors the system provides for and with the actors
  • “The system shall…”
• Object-oriented analysis is reduction of a system to a coarse set of discovered objects with responsibilities, so that the set of objects is able to fulfill the behavioral requirements of the user
• Analysis limits the effort to objects found in the domain – otherwise you are doing design

Analysis Class

• Analysis classes represent an early conceptual model for ‘things in the system which have responsibilities and behavior’
  • Class, stereotyped as «boundary», «entity» or «control»
    • Entity classes
      • Long-lived, real-life object or event in the application domain
      • Data and behavior
      • Usually persistent (saved in a file or database)
    • Boundary classes
      • Interaction between the system and its actors
      • At the system boundary
    • Control classes
      • Coordination and sequencing of system behavior
      • Transactions
Use-Case Realization

• A use-case realization is a description of how a particular use case is realized within the design model, in terms of collaborating objects.

Example: Pay Invoice Use Case

• The use case Pay Invoice is used by a Buyer to schedule invoice payments. The Pay Invoice use case then completes the payment on the due date.
• Pre-condition: The Buyer has received the goods or services ordered and at least one invoice from the system. The Buyer now plans to schedule the invoice(s).
• Post-condition: The use-case ends when the invoice has been paid or the invoice payment was canceled and no money was transferred.
Pay Invoice Use Case: Flow of Events

- Primary Scenario (i.e., basic path)
  - The buyer invokes the use case by beginning to browse the invoices received by the system. The system checks that the content of each invoice is consistent with the order confirmations received earlier (as part of the Confirm Order use case) and somehow indicates this to the buyer. The order confirmation describes which items will be delivered, when, where, and at what price.
  - The buyer decides to schedule an invoice for payment by the bank, and the system generates a payment request to transfer money to the seller’s account. Note that a buyer may not schedule the same invoice for payment twice.
  - Later, if there is enough money in the buyer’s account, a payment transaction is made on the scheduled date. During the transaction, money is transferred from the buyer’s account to the seller’s account, as described by the abstract use case Perform Transactions (which is used by Pay Invoice). The buyer and the seller are notified of the result of the transaction. The bank collects a fee for the transaction, which is withdrawn from the buyer’s account by the system.
  - The use-case instance terminates

Analysis Class Diagram Realizing a Pay Invoice Use Case
Collaboration Diagram Realizing a Pay Invoice Use Case

Analysis and Design Workflow
Summary: Architecture Engineering Steps

• Create an initial sketch of the architecture of the system
• Define an initial set of architecturally significant elements to be used as the basis for analysis
  • Issues that are typically architecturally significant include performance, scaling, process and thread synchronization, and distribution
• Define an initial set of analysis mechanisms
• Define the initial layering and organization of the system
• Define the use-case realizations to be addressed in the current iteration
• Identify analysis classes from the architecturally significant use cases
• Update the use-case realizations with analysis class interactions

Summary: Architecture Engineering Steps (cont.)

• How to Staff
  • Small team with cross-functional team members
  • The team should include members with domain experience who can identify key abstractions
  • The team should also have experience with model organization and layering

• Work Guidelines
  • The work is best done in several sessions, perhaps performed over a few days (or weeks and months for very large systems)
  • Iterate between Architectural Analysis and Use-Case Analysis
Roadmap

Clean Separation Between Analysis and Design

• Analysis versus Design
• Not Top-Down or Bottom-Up
• Review: Use Case Analysis Steps
• Analysis Results
• Grow the Design
• Designer Role

Analysis versus Design

• Analysis
  • Focus on understanding the problem
  • Idealized design
  • Behavior
  • Functional structure
  • Functional requirements
  • A small model

• Design
  • Focus on understanding the solution
  • Operations and attributes
  • Performance
  • Close to real code
  • Object lifecycles
  • Non-functional requirements
  • A large model
Analysis versus Design

• Analysis
  • Understand the problem and begin to develop a visual model of what you are trying to build
    • Independent of implementation and technology concerns
  • Translate the functional requirements into software concepts
    • Get a rough cut at the objects that comprise the system
      • Focus on behavior
      • Focus on structure: coupling/separating concerns

• Design
  • Refine the model into a form that will allow a seamless transition into implementation

“If you tried to manage all of the analysis and design issues in one go, your brain would explode on all but the most trivial developments.”

• Mental switch of focus
  • Analysis: problem domain
    • I do care about what the system has to do
    • I don’t care (as much) about memory, persistence, databases, languages, etc.
  • Design: solution domain
    • I do care about memory, persistence, databases, etc.
    • I don’t care (as much) about understanding the problem

• In practice, Analysis and Design are closely related and often iterative
  • To understand a problem, produce a solution
  • The “as much” above is a balance of focus
Not Top-Down or Bottom-Up  
– It is Middle-In/Out

- The use cases define a middle level
- At the middle level, identify analysis classes that collaborate to realize the use cases
- From the middle level, move up and down
  - Up: Defining subsystems
  - Down: Defining design classes
- Understanding and solving the problem is iterative
  - Middle-to-Top, Top-to-Middle, Middle-to-Bottom, and Bottom-to-Middle

Review: Use-Case Analysis - Steps

- Supplement the Use-Case Description
- 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 architectural analysis mechanisms
- Unify analysis classes
- Review
Analysis Results

- Architecture
  - Packages organized into Layers – focused on upper layers
  - Analysis mechanisms
  - Key abstractions (classes)

- Classes
  - Classes named to reflect their roles and responsibilities
  - Classes defined
    - Responsibilities, Attributes
    - Associations
    - Classes are internally cohesive
  - Classes assigned to packages
    - Closely related classes are in the same packages
    - Packages are loosely coupled

- Use-case realizations
  - All required functionality is realized – all flows are realized
  - Each class has the attributes and responsibilities it needs to carry out its assigned responsibility

Grow the Design
Designer Role

- The designer role defines the responsibilities, operations, attributes, and relationships of one or several classes, and determines how they will be adjusted to the implementation environment

- Skills and Knowledge
  - Use-case modeling techniques
  - System requirements
  - Software design techniques, including object-oriented analysis and design techniques, and the Unified Modeling Language
  - Technologies with which the system will be implemented

Roadmap

Planning Design Alternatives

- Step into Design
- Purpose of “Grow the Design” Workflow
- Context of Use Case Design
- Background of Use Case Design
- Verifying for Consistency
- Purpose of Use Case Design
- Use Case Design Steps
Step into Design

• Next steps are to refine the analysis model into a design model
• Two perspectives for further decomposition
  • Architecture: further detailing of parts and their relationships
    • Packages ➔ Subsystems
    • Analysis mechanisms ➔ Design mechanisms
  • Class: further detailing of the parts
    • Analysis classes ➔ Design classes, components, and interfaces
• A third perspective assures integration ("Re-composition")
  • Use-case realization -- Design

Purpose of “Grow the Design” Workflow

• Provide the natural transition from analysis activities to design activities, identifying:
  • appropriate design elements from analysis elements
  • appropriate design mechanisms from related analysis mechanisms
• Maintain the consistency and integrity of the architecture, ensuring that:
  • new design elements identified for the current iteration are integrated with pre-existing design elements.
  • maximal re-use of available components and design elements is achieved as early as possible in the design effort.
Context of Use-Case Design

- Based on prior analysis and design, we have
  - An initial architecture design
  - Defined major elements of our system (subsystems, their interfaces, the design classes, the processes, and threads) and their relationships
  - An understanding of how processes map to system hardware
- Now, in use-case design, we concentrate on how a use case is implemented and make sure there is consistency from beginning to end
- Use-case design is where the design elements meet the architectural mechanisms
  - The use-case realization from analysis is refined to include the design elements, using the patterns of interaction defined for the architectural mechanisms

Background of Use-Case Design

- Use-Case Design is a cross-checking, consistency-imposing activity
  - Focus on a “use-case thread” to make sure everything still fits together
  - Continually review the interaction diagrams, looking for
    - inconsistencies
    - missing information
    - opportunities for reuse
    - etc.
  - Make sure we have operations for the whole path of the flow of events
Verifying for Consistency

- For each use case, across all use cases, verify that
  - All the necessary behavior to support a use-case implementation has been distributed among the appropriate participating classes
  - The use-case flows naturally over the participating design elements
  - All associations between design elements (classes or subsystems) needed for the use case realizations have been defined
  - All the attributes needed for the use-cases have been defined
  - Refine the analysis-level use cases to included the defined design model elements (which are the design classes and subsystems that the analysis classes have “morphed into”)
  - Incorporate applicable architectural design mechanisms

Purpose of Use-Case Design

- To refine use-case realizations in terms of interactions.
- To refine requirements on the operations of design classes.
- To refine requirements on the operations of subsystems and/or their interfaces.
- To refine requirements on the operations of capsules
Use-Case Design

Multiple design teams, each with a different focus

- Class diagrams
- Sequence diagrams
- Refined flow-of-events descriptions

Use-Case Design Steps

- Describe interaction between design objects
- Simplify sequence diagrams using subsystems
- Describe persistence related behavior
  - Read, write, delete persistent objects
  - Model transactions
  - Handle error conditions
  - Handle concurrency control
- Refine the flow of events description
- Unify classes and subsystems
- Evaluate your results
Use-Case
Design Steps

- Describe interaction between design objects
- Simplify sequence diagrams using subsystems
- Describe persistence related behavior
  - Read, write, delete persistent objects
  - Model transactions
  - Handle error conditions
  - Handle concurrency control
- Refine the flow of events description
- Unify classes and subsystems
- Evaluate your results

Review: Use-Case Realization

- A use-case realization is a description of how a particular use case is realized within the design model, in terms of collaborating objects
Review: From Analysis Classes to Design Elements

- Analysis classes will seldom retain their same structure through design
  - They may be expanded, collapsed, combined, or even deleted in design

Use-Case Realization Refinement

- Identify participating objects
- Allocate responsibilities amongst objects
- Model messages between objects
- Describe processing resulting from messages
- Model associated class relationships
- In essence, re-draw the analysis realizations, now using the design elements the analysis models were refined into, making sure that the allocation of responsibility is still accurate
Use-Case Realization
Refinement Steps

• Identify each object that participates in the flow of the use-case
• Represent each participating object in a sequence diagram

• Incrementally incorporate applicable architectural mechanisms

Use-Case
Design Steps

• Describe interaction between design objects
• Simplify sequence diagrams using subsystems
• Describe persistence related behavior
  • Read, write, delete persistent objects
  • Model transactions
  • Handle error conditions
  • Handle concurrency control
• Refine the flow of events description
• Unify classes and subsystems
• Evaluate your results
With Subsystems & Architectural Mechanisms

- Look at the interaction diagrams
- For each class that has been refined into a subsystem, replace the class with the associated subsystem interface
  - Any interactions that describe HOW the subsystem should implement the service should be deferred until subsystem design
- Incrementally incorporate any applicable architectural mechanisms
  - Use the patterns of behavior defined for the mechanisms
  - May include the introduction of new design elements and messages
- Update the dynamic and static parts (sequence diagrams and view of participating classes diagram)

Representing Subsystems on a Sequence Diagram

- Interfaces
  - Represents any model element that realizes the interface
  - No message should be drawn from the interface
- Proxy class
  - Represents a specific subsystem
  - Messages can be drawn from the proxy
Example: Incorporating Subsystem Interfaces

Analysis Classes

- CourseCatalogSystem
  - getCourseOfferings()

- BillingSystem
  - submitBill()

Design Elements

Example: Incorporating Subsystem Interfaces (Before)

Replace with subsystem interface

Sequence Diagram: Register for Courses / Register for Courses - Basic Flow (Submit Schedule)

1. // create schedule()
1.2. // display course offerings()
1.1. // get course offerings()
1.1.1. // get course offerings(forSemester)
1.3. // display blank schedule()
1.1.1.1. // get course offerings()
2. // select 4 primary and 2 alternate offerings()
2.1. // create schedule with offerings()
2.1.1. // create schedule with offerings()
2.1.2. // add schedule(Schedule)
2.1.1.1. // get course offerings()
1.2. // add schedule(Schedule)

At this point the Submit Schedule subflow is executed.
Example: Incorporating Subsystem Interfaces (After)

Encapsulating Subsystem Interactions

- Interactions can be described at several levels of abstraction
- Subsystem internal interactions can be described in their own interaction diagrams
When to Encapsulate Sub-Flows in a Subsystem

- Encapsulate a sub-flow when it:
  - occurs in multiple use-case realizations
  - has reuse potential
  - is complex and easily encapsulated
  - is the responsibility of one person/team
  - produces a well-defined result
  - is encapsulated within a single Implementation Model component

- Make sure that what you are abstracting is worth abstracting

Guidelines: Encapsulating Subsystem Interactions

- Subsystems should be represented by their interfaces on interaction diagrams
- Messages to subsystems are modeled as messages to the subsystem interface
- Messages to subsystems correspond to operations on the subsystem interface
- Interactions within subsystems are modeled in subsystem design
Advantages of Encapsulating Subsystem Interactions

• Use-case realizations
  • Are less cluttered
  • Can be created before the internal designs of subsystems are created
    • parallel development
  • Are more generic and easier to change
    • subsystems can be substituted

Parallel Subsystem Development

• Concentrate on requirements that affect subsystem interfaces
• Outline required interfaces
• Model messages that cross subsystem boundaries
• Draw interaction diagrams in terms of subsystem interfaces for each use-case
• Refine the interfaces needed to provide messages
• Develop each subsystem in parallel

Use subsystem interfaces as project synchronization points
Use-Case
Design Steps

• Describe interaction between design objects
• Simplify sequence diagrams using subsystems
• Describe persistence related behavior
  • Model transactions
  • Read, write, delete persistent objects
  • Handle error conditions
  • Handle concurrency control
• Refine the flow of events description
• Unify classes and subsystems
• Evaluate your results

Modeling
Transactions

• What is a Transaction?
  • Atomic operation invocations
  • “All or nothing”
  • Assure consistency in changes
• Modeling options
  • Textually (scripts)
  • Explicit messages
• Error conditions
  • Rollback
  • Failure modes
  • May require separate interaction diagrams
Use-Case
Design Steps

• Describe interaction between design objects
• Simplify sequence diagrams using subsystems
• Describe persistence related behavior
  • Model transactions
  • Read, write, delete persistent objects
  • Handle error conditions
  • Handle concurrency control
• Refine the flow of events description
• Unify classes and subsystems
• Evaluate your results

Detailed Flow of Event Description Options

• Annotate the interaction diagrams to clarify behavior
  • Timing
  • Conditional behavior
  • Clarification or constraint on operation behavior
  • Control flow (decision steps, looping, branching)
• Use UML notes or use free-form text
Use-Case Design Steps

• Describe interaction between design objects
• Simplify sequence diagrams using subsystems
• Describe persistence related behavior
  • Model transactions
  • Read, write, delete persistent objects
  • Handle error conditions
  • Handle concurrency control
• Refine the flow of events description
• Unify classes and subsystems
• Evaluate your results

Design Model Unification Considerations

• Model element names should describe their function
  • Avoid similar names
  • Avoid synonyms
• Merge similar model elements
• Use inheritance to abstract model elements
• Keep model elements and flows of events consistent
  • When updating a model element, also update the affected use-case realizations
Use-Case
Design Steps

• Describe interaction between design objects
• Simplify sequence diagrams using subsystems
• Describe persistence related behavior
  • Model transactions
  • Read, write, delete persistent objects
  • Handle error conditions
  • Handle concurrency control
• Refine the flow of events description
• Unify classes and subsystems
• Evaluate your results

Checkpoints: Use-Case Design

• Is package/subsystem partitioning logical and consistent?
• Are the names of the packages/subsystems descriptive?
• Do the public package classes and subsystem interfaces provide a single, logically consistent set of services?
• Do the package/subsystem dependencies correspond to the relationships between the contained classes?
• Do the classes contained in a package belong there according to the criteria for the package division?
• Are there classes or collaborations of classes which can be separated into an independent package/subsystem?
Checkpoints: Use-Case Design

- Have all the main and/or sub-flows for this development iteration been handled?
- Has all behavior been distributed among the participating design elements?
- Has behavior been distributed to the right design elements?
- If there are several interaction diagrams for the use-case realization, is it easy to understand which diagrams relate to which flow of events?

Any Questions?