Lecture 1

Process and Method:
An Introduction to the
Rational Unified Process

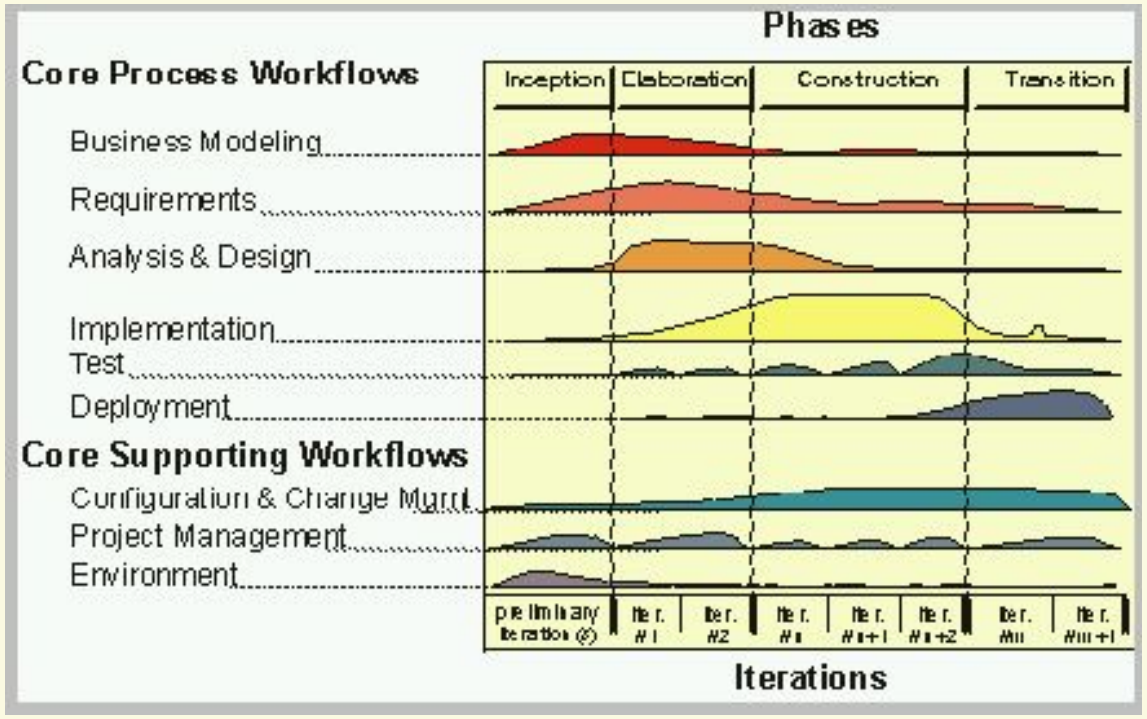
The Rational Unified Process

- RUP is a method of managing OO Software Development
- It can be viewed as a Software Development Framework which is extensible and features:
  - Iterative Development
  - Requirements Management
  - Component-Based Architectural Vision
  - Visual Modeling of Systems
  - Quality Management
  - Change Control Management

RUP Features

- Online Repository of Process Information and Description in HTML format
- Templates for all major artifacts, including:
  - RequisitePro templates (requirements tracking)
  - Word Templates for Use Cases
  - Project Templates for Project Management
- Process Manuals describing key processes

The Phases


Phases
Core Process Workflows
Inception
Elaboration
Construction
Transition
Business Modeling
Requirements
Analysis & Design
Implementation
Test
Deployment
Core Supporting Workflows
Configuration & Change Mgmt
Project Management
Environment
preliminary iteration(s)
iter. #1
iter. #2
iter. #n
iter. #n+1
iter. #n+2
iter. #m
iter. #m+1
Iterations

An Iterative Development Process...

- Recognizes the reality of changing requirements
  - Caspers Jones's research on 8000 projects
    - 40% of final requirements arrived after the analysis phase, after development had already begun
- Promotes early risk mitigation, by breaking down the system into mini-projects and focusing on the riskier elements first
- Allows you to "plan a little, design a little, and code a little"
- Encourages all participants, including testers, integrators, and technical writers to be involved earlier on
- Allows the process itself to modulate with each iteration, allowing you to correct errors sooner and put into practice lessons learned in the prior iteration
- Focuses on component architectures, not final big bang deployments

An Incremental Development Process...

- Allows for software to evolve, not be produced in one huge effort
- Allows software to improve, by giving enough time to the evolutionary process itself
- Forces attention on stability, for only a stable foundation can support multiple additions
- Allows the system (a small subset of it) to actually run much sooner than with other processes
- Allows interim progress to continue through the stubbing of functionality
- Allows for the management of risk, by exposing problems earlier on in the development process

Goals and Features of Each Iteration

- The primary goal of each iteration is to slowly chip away at the risk facing the project, namely:
  - performance risks
  - integration risks (different vendors, tools, etc.)
  - conceptual risks (ferret out analysis and design flaws)
- Perform a “miniwaterfall” project that ends with a delivery of something tangible in code, available for scrutiny by the interested parties, which produces validation or correctives
- Each iteration is risk-driven
- The result of a single iteration is an increment--an incremental improvement of the system, yielding an evolutionary approach

Risk Management

- Identification of the risks
- Iterative/Incremental Development
- The prototype or pilot project
  - Booch’s “Tiger Team”
- Early testing and deployment as opposed to late testing in traditional methods

The Development Phases

- Inception Phase
- Elaboration Phase
- Construction Phase
- Transition Phase

Inception Phase

- Overriding goal is obtaining buy-in from all interested parties
- Initial requirements capture
- Cost Benefit Analysis
- Initial Risk Analysis
- Project scope definition
- Defining a candidate architecture
- Development of a disposable prototype
- Initial Use Case Model (10% - 20% complete)
- First pass at a Domain Model

Elaboration Phase

- Requirements Analysis and Capture
  - Use Case Analysis
    - Use Case (80% written and reviewed by end of phase)
    - Use Case Model (80% done)
    - Scenarios
      - Sequence and Collaboration Diagrams
      - Class, Activity, Component, State Diagrams
  - Glossary (so users and developers can speak common vocabulary)
  - Domain Model
    - to understand the problem: the system's requirements as they exist within the context of the problem domain
  - Risk Assessment Plan revised
  - Architecture Document

Construction Phase

- Focus is on implementation of the design:
  - cumulative increase in functionality
  - greater depth of implementation (stubs fleshed out)
  - greater stability begins to appear
  - implement all details, not only those of central architectural value
  - analysis continues, but design and coding predominate

Transition Phase

- The transition phase consists of the transfer of the system to the user community
- It includes manufacturing, shipping, installation, training, technical support and maintenance
- Development team begins to shrink
- Control is moved to maintenance team
- Alpha, Beta, and final releases
- Software updates
- Integration with existing systems (legacy, existing versions, etc.)

Elaboration Phase in Detail

- Use Case Analysis
  - Find and understand 80% of architecturally significant use cases and actors
  - Prototype User Interfaces
  - Prioritize Use Cases within the Use Case Model
  - Detail the architecturally significant Use Cases (write and review them)
- Prepare Domain Model of architecturally significant classes, and identify their responsibilities and central interfaces (View of Participating Classes)

Use Case Analysis

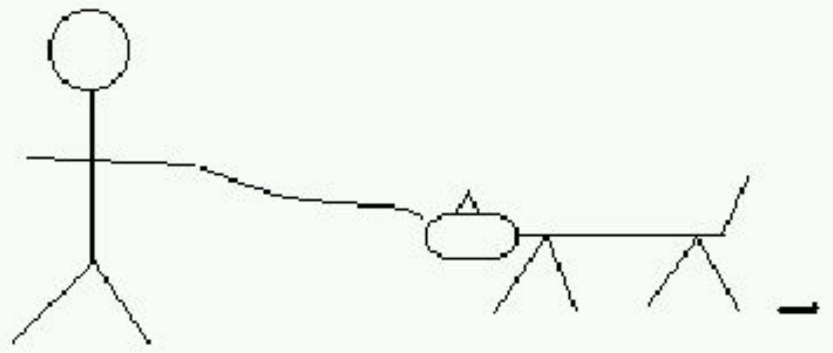
- What is a Use Case?
  - A sequence of actions a system performs that yields a valuable result for a particular actor.
- What is an Actor?
  - A user or outside system that interacts with the system being designed in order to obtain some value from that interaction
- Use Cases describe scenarios that describe the interaction between users of the system and the system itself.
- Use Cases describe WHAT the system will do, but never HOW it will be done.

What’s in a Use Case?

- Define the start state and any preconditions that accompany it
- Define when the Use Case starts
- Define the order of activity in the Main Flow of Events
- Define any Alternative Flows of Events
- Define any Exceptional Flows of Events
- Define any Post Conditions and the end state
- Mention any design issues as an appendix
- Accompanying diagrams: State, Activity, Sequence Diagrams
- View of Participating Objects (relevant Analysis Model Classes)
- Logical View: A View of the Actors involved with this Use Case, and any Use Cases used or extended by this Use Case

Use Cases Describe Function not Form

- Use Cases describe *WHAT* the system will do, but never *HOW* it will be done.
- Use Cases are Analysis Products, not Design Products.



Next, the System Operator Actor's dog Fifo introduces a design object into the Analysis Phase.

Figure 1: FIFO Example

Use Cases Describe Function not Form

- Use Cases describe WHAT the system should do, but never HOW it will be done
- Use cases are Analysis products, not design products

Benefits of Use Cases

- Use cases are the primary vehicle for requirements capture in RUP
- Use cases are described using the language of the customer (language of the domain which is defined in the glossary)
- Use cases provide a contractual delivery process (RUP is Use Case Driven)
- Use cases provide an easily-understood communication mechanism
- When requirements are traced, they make it difficult for requirements to fall through the cracks
- Use cases provide a concise summary of what the system should do at an abstract (low modification cost) level.

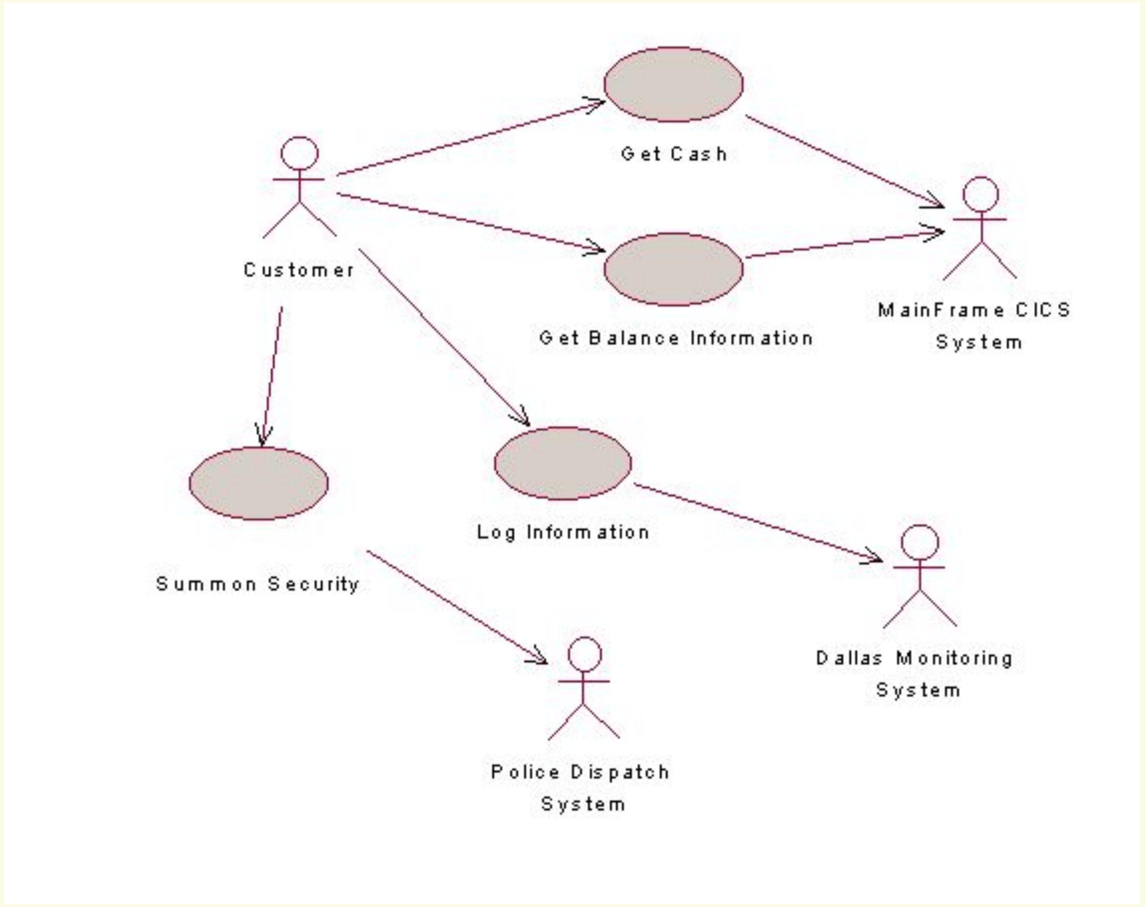
Difficulties with Use Cases

- As functional decompositions, it is often difficult to make the transition from functional description to object description to class design
- Reuse at the class level can be hindered by each developer "taking a Use Case and running with it". Since UCs do not talk about classes, developers often wind up in a vacuum during object analysis, and can often wind up doing things their own way, making reuse difficult
- Use Cases make stating non-functional requirements difficult (where do you say that X must execute at Y/sec?)
- Testing functionality is straightforward, but unit testing the particular implementations and non-functional requirements is not obvious

Use Case Model Survey

- The Use Case Model Survey is to illustrate, in graphical form, the universe of Use Cases that the system is contracted to deliver.
- Each Use Case in the system appears in the Survey with a short description of its main function.
  - Participants:
    - Domain Expert
    - Architect
    - Analyst/Designer (Use Case author)
    - Testing Engineer

Sample Use Case Model Survey



Analysis Model

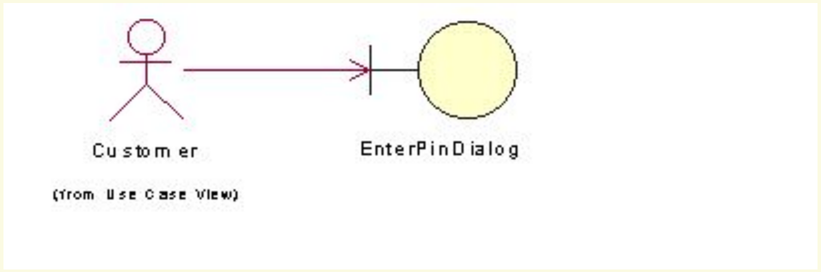
- In Analysis, we analyze and refine the requirements described in the Use Cases in order to achieve a more precise view of the requirements, without being overwhelmed with the details
- Again, the Analysis Model is still focusing on WHAT we're going to do, not HOW we're going to do it (Design Model). But what we're going to do is drawn from the point of view of the developer, not from the point of view of the customer
- Whereas Use Cases are described in the language of the customer, the Analysis Model is described in the language of the developer:
  - Boundary Classes
  - Entity Classes
  - Control Classes

Why spend time on the Analysis Model, why not just "face the cliff"?

- By performing analysis, designers can inexpensively come to a better understanding of the requirements of the system
- By providing such an abstract overview, newcomers can understand the overall architecture of the system efficiently, from a 'bird's eye view', without having to get bogged down with implementation details.
- The Analysis Model is a simple abstraction of what the system is going to do from the point of view of the developers. By "speaking the developer's language", comprehension is improved and by abstracting, simplicity is achieved
- Nevertheless, the cost of maintaining the AM through construction is weighed against the value of having it all along.

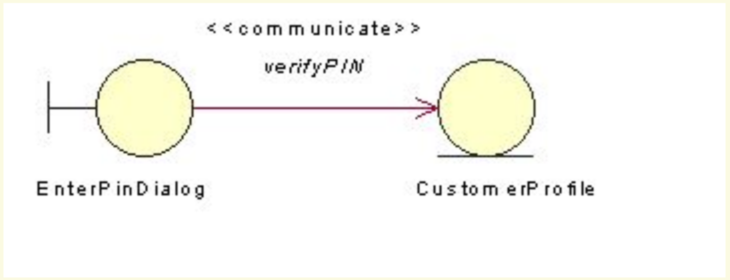
Boundary Classes

- Boundary classes are used in the Analysis Model to model interactions between the system and its actors (users or external systems)
- Boundary classes are often implemented in some GUI format (dialogs, widgets, beans, etc.)
- Boundary classes can often be abstractions of external APIs (in the case of an external system actor)
- Every boundary class must be associated with at least one actor:



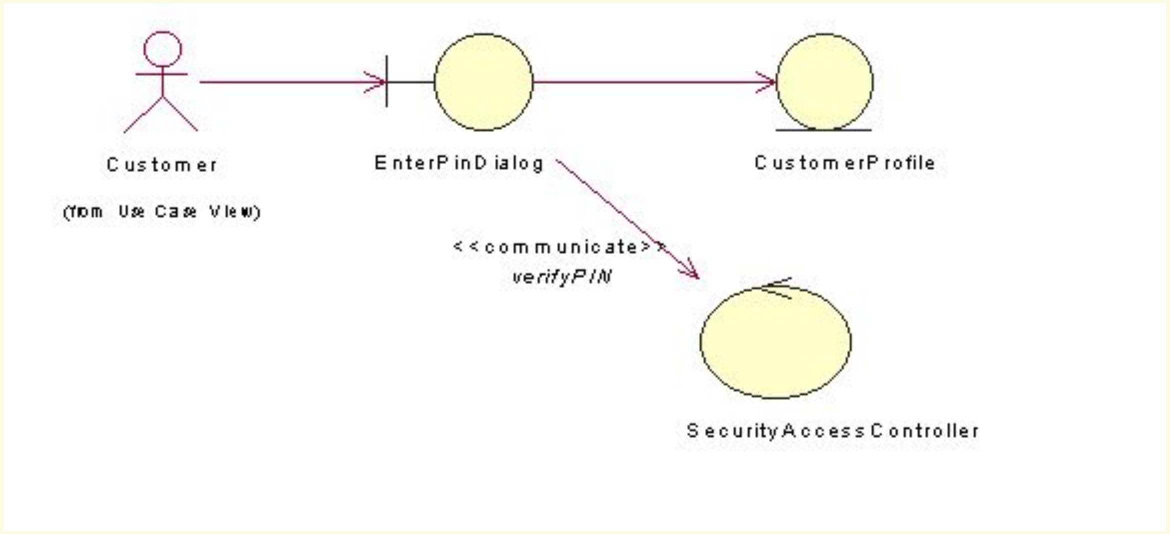
Entity Classes

- Entity classes are used within the Analysis Model to model persistent information
- Often, entity classes are created from objects within the business object model or domain model



Control Classes

- Control classes model abstractions that coordinate, sequence, transact, and otherwise control other objects
- In Smalltalk MVC mechanism, these are controllers
- Control classes are often encapsulated interactions between other objects, as they handle and coordinate actions and control flows.


Customer
(from Use Case View)
EnterPinDialog
CustomerProfile
<<communicate>>
verifyPIN
SecurityAccessController