

UML Project Plan

Richard Felsinger, 960 Scotland Drive, Mt Pleasant, SC 29464 dick@felsinger.com 843-881-3648 1/29/2001

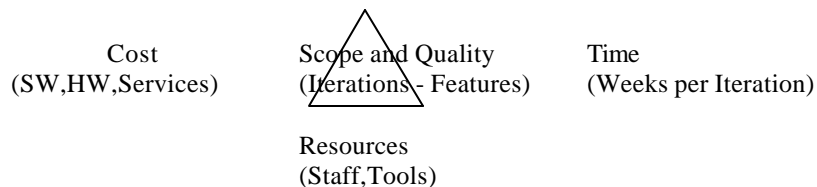
The purpose of this project plan is to provide a template for your project. There are a large number of templates and tables which you should fill-in with your project information, estimates, etc. The single most important reference in this plan is *The Rational Unified Process An Introduction Second Edition* by Philippe Kruchten. A sample UML model - Simplified Bank Account example is provided in the appendix to show examples of UML diagrams and specifications. To update this plan for your project:

- change the name OOProject to your project name,
- fill-in the various template forms with your project information,
- update this document to reflect your project plans and policies,
- get project team member feedback, approve, then place the updated project plan in a shared directory,
- execute the plan and monitor the project.

Our goal is that this project plan shall assist all project team members to work toward the successful completion of the project and to create a defect-free software product.

Introduction

An OO Project is a sequence of unique, complex, and connected activities having one goal or purpose, and that must be completed by a specific time, within budget, and according to specification. Key aspects of a project are shown below. Increasing "Scope and Quality" in the middle of the triangle will increase the "Cost", "Time", and "Resources".



Key aspects of OO Project Management compared to a non-OO Project Management are:

- planning and monitoring at various levels of scale/abstraction: Enterprise - Business Level, Project - System Level, Build/Release Level,
- using the Unified Process Phases: Inception - Definition, Elaboration - Planning, Construction - Modeling/Coding, Transition - Deployment to end users,
- using the Unified Process create the following models: Requirements, Analysis, Design, Implementation, and Testing for each Build/Release.
- using Unified Modeling Language elements and semantics,
- using Object-oriented size, complexity, and quality measures.

Grady Booch in *Object-Solutions - Managing the Object-Oriented Project* states "The central task of the software management team is to balance a set of incomplete, inconsistent, and shifting technical and non-technical requirements, to produce a system that is optimal for its essential minimal characteristics." Booch states "A successful software project is one whose deliverables satisfy and possibly exceed the end user's expectations, was developed in a timely and economical fashion, and is resilient to change and adaptation." Project management consists of planning, scheduling, staffing, resource allocation, and monitoring to create a defect free system "better, faster, cheaper".

Grady Booch in *Object-Solutions - Managing the Object-Oriented Project* states "The five habits of a successful object-oriented project include:

- A ruthless focus on the development of a system that provides a well-understood collection of essential minimal characteristics.
- The existence of a culture that is centered on results, encourages communication, and yet is not afraid to fail.
- The effective use of object-oriented modeling.
- The existence of a strong architectural vision.

- The application of a well-managed iterative and incremental development life cycle.”

Philippe Kruchten in *The Rational Unified Process An Introduction Second Edition* provides suggestions to support effective software engineering:

- Develop software iteratively.
- Manage requirements.
- Use component-based architectures.
- Verify software quality.
- Control changes to software.

The following are the recommended texts for the project:

The Unified Modeling Language User Guide by Grady Booch, James Rumbaugh, and Ivar Jacobson,
The Unified Modeling Language Reference Manual by James Rumbaugh, Ivar Jacobson, and Grady Booch,
The Unified Software Development Process by Ivar Jacobson, Grady Booch, and James Rumbaugh,
The Rational Unified Process An Introduction Second Edition by Kruchten.

Other references are listed at the end of the plan.

The following are the recommended standards:

The Unified Modeling Language - www.omg.org

Coding Standards - <http://java.sun.com/docs/codeconv/index.html> or <http://gee.cs.oswego.edu/dl/html/javaCodingStd.html>.

Enterprise Planning and Monitoring

The OOProject System should be modeled in terms of the level of scale/abstraction as shown below. It is important to know where the OOProject is in terms of the overall enterprise.

Levels of Scale/Abstraction

Level	Definition	UML	Example	OOProject
Global	Concerns languages, standards, policies that affect multiple enterprises		Internet - ANSI and IEEE Standards	
Enterprise	Organization with systems		XYZ Company	
Overall System - Group of Applications/	Requirements View: actors and the system Implementation View: components	Requirements: Actor + System Implementation: Components	Office 2000	Overall System including OOProject
System/Subsystem/Component - Application	Group of classes that operate together as a system or application	System Package or Component	Word 2000	OOProject System
Package	Group of classes	Package - tabbed box		
Collaboration	Group of classes that act together for a specific purpose - implements a pattern	Collaboration - dashed oval		
Class	Defines a group of objects	Class	Document	
Attribute - Operation	Attribute - Values; Operation - Service	Attribute - Operation	Document.Name - Document.Open()	

It is desirable to show the OOPProject System as a component in the larger system for the following reasons:

- sets the boundary of the OOPProject System,
- facilitates accurate communications to know the level of scale/abstraction,
- facilitates assigning responsibility for the OOPProject System and interacting components,
- speeds development if component interfaces (set of operations) are clearly defined.

Enterprise Business Modeling

Business Modeling is to model the enterprise as a whole. It is important for the OOPProject to support Enterprise short-term and long-term goals and to properly fit-into the Enterprise.

The Business Model provides the following: Vision Document, Organization Chart, Business Events and Processes (Use Cases), Business Actors, Workers, and Entities (Domain Model), Business Rules Catalog, Business Interfaces (Set of Operations), Business Patterns, Business Systems Architecture - Component Diagram, Glossary. See *The Rational Unified Process An Introduction* Second Edition by Krutchen and *Business Modeling with UML* by Eriksson and Penker.

	Business Model
Key UML Elements	Business Processes (Use Cases), Business Domain Objects
Key Concern	Model Business
Objective	Sufficient Business/Enterprise information
Static/Structural Diagrams	Business Domain Objects
Dynamic/Time Based Diagrams	Business Processes (Use Cases)
Tools	UML CASE, Requirements Tracking
Key Team Players	Business/System Analysts, Architect
Model Sign-off	Project Manager, Architect, Client/User

The following is a sample status table for the Enterprise Business Model:

Enterprise Business Model			
	Location - Reference	Number	Comment
Business Model			
Business Events			
Business Actors, Workers, Entities			
Business Interfaces			
Business Patterns			
Business Glossary			
Architecture - Components			

Benefits of Business Modeling are:

- supports defining good requirements leading to rapid, effective system development,
- supports creating a system that is correct, reliable, extensible, and reusable,
- supports communication, consistency and reduces redundancy.

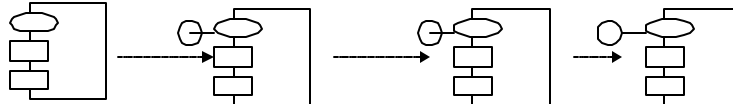
System Architecture for CBD - Component-Based Development

The OOPProject System is a part of a larger enterprise system consisting of components. **Component-based development (CBD)** is the creation and deployment of software-intensive systems assembled from components, as well as the development and harvesting of such components. It is desirable to have a **layered architecture of components** - an ordered set of virtual worlds, each built in terms of the ones below it and providing the basis of implementation for the ones above it.

Kruchten in *The Rational Unified Process An Introduction Second Edition* defines architecture as follows: "Architecture encompasses significant decisions about the following:

- The organization of a software system.
- The selection of structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaboration among those elements.
- The composition of these elements into progressively larger subsystems.
- The architectural style that guides this organization, these elements and their interfaces, their collaborations, and their composition."

Architecture refers to the organizational structure of a system, including its decomposition into parts, their connectivity, interaction mechanisms, and the guiding principles that inform the design of a system. The UML component diagram shown below has components with lollipops (interfaces). An interface is a set of operations without implementation.



Benefits of Component Based Development are:

- supports developing highly upgradable, modifiable systems with plug-in replacement components,
- supports communications by defining components with well-defined interfaces (set of operations),
- supports reusability by defining reusable components,
- supports a highly resilient system architecture,
- supports using standardized component frameworks, e.g. COM+, CORBA, EJB, etc,
- supports using commercially available components,
- provides a natural basis for configuration management and versioning.

Project Planning and Monitoring

Project Objectives and Overview

The OOPProject shall design, construct, and deploy the OOPProject System in accordance with the OOPProject Requirements. The objective is to create a system that is correct, reliable, understandable, extensible, and reusable. The system must meet all functional requirements, e.g. features (modeled with use cases). The system must meet non-functional requirements: usability, reliability, performance, and supportability.

	Description or Location	Comment
Project Name		
Project Description		
Project Objectives		
Project Functional Requirements Document		
Project Non-Functional Requirements Document		
Project Constraints		
Project Assumptions		
Project Standards	UML, Coding Standards, Other (exceptions, threads)	
Enterprise Business Model		
Project Goodness Guidelines	See Appendix	
Project Stereotypes, Tagged Values, and Constraints	See Appendix	
Sample Project UML Model	See Appendix	
Project Documentation	See Summary of Artifacts (Appendix B) in The Rational Unified Process An Introduction Second Edition by Kruchten	
Project Tools	Tutorials, Tapes, CDs, Books, Training Sessions	
Project Glossary		
Project Reuse Libraries	Component, Class, Operation, Pattern-UML	

	Collaboration	
Project UML Model Review	Bi-weekly or at completion of each iteration	

Benefits of defining project objectives are:

- supports communications by getting team members, the client, and others "on the same page",
- supports measurement of plan versus actual to monitor progress and identify potential problems,
- supports efficiency by getting team members focused on meeting the project objectives,
- supports setting effective planning and prioritization of activities to meet the project objectives.

Project Risks

Risk is an ongoing or impending concern that has a significant probability of adversely affecting the success of major milestones. If the risk occurs then there may be significant adverse affect on the project in terms of cost, schedule, and features.

Booch in *Object Solutions* states "What are the most serious risks factors that face any real project?"

- Inaccurate metrics
- Inadequate measurement
- Excessive schedule pressure
- Management malpractice
- Inaccurate cost estimating
- Silver bullet syndrome
- Creeping user requirements
- Low quality
- Low productivity
- Canceled projects"

To ensure that we meet project objectives, the OOProject shall identify and monitor all major risks. We must prepare for and avoid catastrophic "surprises" and unexpected events. The projected risks for the OOProject is shown below.

Risk Name	Description	Probability of Occurrence	Impact if Occurs	Avoidance Plan	Contingency Plan if Occurs	Comment
Database not delivered on schedule		10%	Delay of Project	Monitor Monthly		

Benefits of defining project risks are:

- supports effective planning to avoid "surprises",
- greatly increases the probability for a successful project,
- supports effective decision making for a successful project.

Project Phases and Scheduling

The OOProject shall follow the following the Unified Software Development Process as documented in *The Unified Software Development Process* by Ivar Jacobson, Grady Booch, and James Rumbaugh and *The Rational Unified Process An Introduction Second Edition* by Krutchen. This is an incremental iterative development process that emphasizes the delivery of progressively more complex software builds/releases. A **phase** is the span of time between two major milestones of a development process, e.g. inception, elaboration, construction, transition. The phases are described below.

Unified Process Phases

Source: *The Rational Unified Process An Introduction Second Edition* by Krutchen
With Number of Projected Weeks per Phase for a 52 Week Project

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

	5 weeks	16 weeks	26 weeks	5 weeks
Description	Define the scope of the project and develop business case	Plan the project, specify features, and baseline the architecture	Build the product. Software is brought from an executable architectural baseline to the point where it is ready to be transitioned to the user community	the software is turned into the hands of the user community
Products	Vision document, use case list, project glossary, business case (context, success criteria, financial forecast), risk assessment, project plan, business model	Use case model, non-functional requirements, software architecture, architectural prototype, iteration plan, development process, preliminary user manual	UML model (requirements, analysis, design, implementation, testing) and build/release for each iteration	Software product rollout to marketing, distribution, and sales teams
Estimated Time for 52 week project	10% - 5 weeks	30% - 16 weeks	50% - 26 weeks with 2 -3 week iteration	10% - 5 weeks
Estimated Effort/Resources	5%	20%	65%	10%
Key Personnel Roles	Project Manager, Architect, Business/System Analyst	Project Manager, Architect, Business/System Analyst	Project Manager, Architect, Business/System Analyst, Developer/Programmer , QA Tester	Project Manager, Architect
Milestone to be Achieved at end of Phase - Project Manager Sign-off	Lifecycle Objective Milestone	Lifecycle Architecture Milestone	Initial Operational Capability Milestone	Product Release Milestone

Benefits of having well-defined project phases are:

- supports having a well-managed project,
- supports communications so that the client and team members know the progression of the project,
- supports measurement of planned versus actual to identify problems early.

Project Staffing

The OOProject shall be staffed with person filling the following roles: Project Manager, Architect, Methodologist/Toolsmith, User, Business/System Analyst, Developer/Programmer, QA Tester, and others as required. The description of each role are:

Project Manager - manages all aspects of the project including schedules, resources, staffing, etc to meet the project objectives and to effect the project build/release software products.

Architect - oversees the technical aspects of the project including the overall system architecture of components, their interfaces (set of operations), and their communications. Responsible for the development and deployment infrastructure. Provides the Processing Environment (HW and SW Configuration List) and Implementation Model (component diagram and deployment diagram).

Methodologist/Toolsmith - oversees the use of UML and the Unified Process. Responsible to ensure the correctness and completeness of UML models. Provides the UML, Unified Process, and tools help desk. Creates CASE tool scripts for reporting and code generation.

Client/User - provides the user point of view and acts as the domain expert.

Business/System Analyst - leads and coordinates the requirements gathering, use case modeling, and class modeling in the Business Modeling, Requirements, and Analysis Models.

Developer/Programmer - creates all diagrams, specifications, and code in the Design Model.

QA Tester - creates the test plan, test cases, test procedures, and related testing documentation. Conducts tests and provides test case results.

Number Assigned Staff - Names/TBD - to be determined
With Number of Projected Weeks per Phase for a 52 Week Project

Roles	Inception Phase - 5 weeks	Elaboration Phase - 16 weeks	Construction Phase - 26 weeks	Transition Phase - 5 weeks
Project Manager	1 - John Smith	1 - John Smith	1 - John Smith	1 - John Smith
Architect	1 - ???	1 - ???	1 - ???	1 - ???
Client/User	1 - ???	1 - ???	1 - ???	1 - ???
Business/System Analysts	3 - ???, ???, ???	3 - ???, ???, ???	3 - ???, ???, ???	0
Developer/Programmer	0	3 - ???, ???, ???	3 - ???, ???, ???	1 - ???
QA Tester	0	1 - ???	1 - ???	1 - ???
Other	TBD	TBD	TBD	TBD
Total Assigned	6	10	10	5

Benefits of having well-defined roles for team members are:

- supports effective planning and decision making for a successful project,
- supports communication so that team members know their responsibilities,
- supports creating a quality system with different team members working on the system from different points of view.

Project Resources

Resources must be identified, budgeted, and controlled - both personnel and other resources, e.g. tools, equipment, services, etc.

Resources - Requested/Authorized/Used in each cell
With Number of Projected Weeks per Phase for a 52 Week Project

Resource Category	Inception Phase - 5 weeks	Elaboration Phase - 16 weeks	Construction Phase - 26 weeks	Transition Phase - 5 weeks
Personnel				
Services				
Software				
Equipment				
Travel				
Other				
Total				

Benefits of having well-defined roles for team members are:

- supports effective planning and decision making for a successful project,
- supports communication to identify required resources,
- supports creating a quality system and a satisfied customer.

Project Configuration Management and Versioning

The goal of project configuration management is to track and maintain the integrity of evolving project assets. These assets must be available for reuse. There are three independent functions:

- configuration management deals with the issue of artifact (asset/document) identification, versions, and dependencies;
- change request management deals with the capture and management of requested changes in artifacts (asset/document);
- status and measurement deals with project control information.

Project Assets and Documents

Artifact (asset/document)	Responsibility	Location	Current Version/Date	Tool	Comment

Benefits of having configuration management and versions are:

- supports communication so that team members are working on the latest version,
- supports efficiency by reducing redundant efforts,
- supports creating a quality system in which all parts fit together.

Project Requirements

The OOPProject shall maintain an up-to-date requirements document and a Requirements Traceability Table shown below.

Requirements Traceability Table (Partial)

Requirement Number	Requirement Name	Reference	Use Case Name	UML Element	Test Case	Description	Responsibility
1.1	DepositToSavings Account		DepositToSavingsAccount				

Benefits of having well-defined project requirements are:

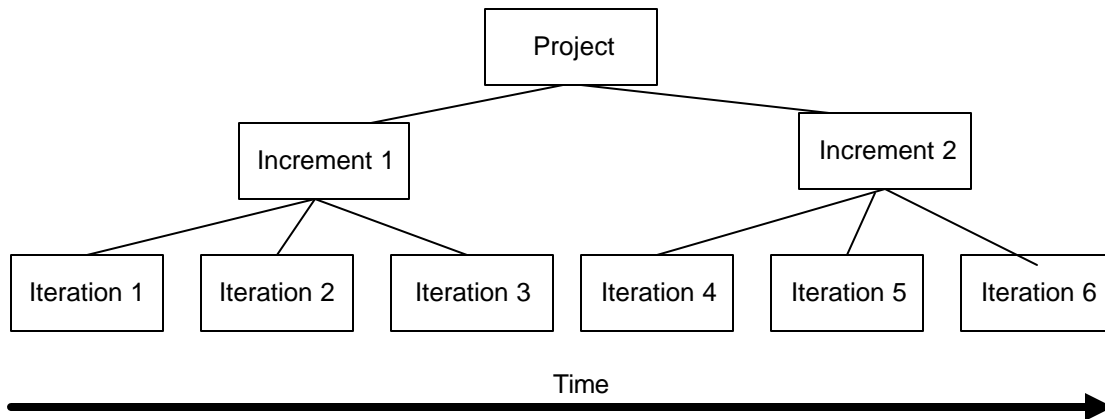
- supports communication so that team members are working to meet the requirements,
- supports defining use cases, use case increments (group of use cases) and build/release iterations (use case scenarios within an increment),
- supports identifying and resolving inconsistencies in requirements,
- supports creating a quality system in which the client requirements are fully met.

Iteration Planning and Monitoring

The OOPProject shall use an incremental and iterative software development approach as documented in the Unified Process. A **Use Case Increment** is a set of use cases that represent a complete subset of business functionality largely independent of other increments. A **Use Case Scenario** is a set of interactions for a use case, e.g. optimistic(simple), normal (moderate), or pessimistic (complex) scenario. An **iteration** is a sequence of activities with an established plan and evaluation criteria, resulting in an executable release. It is a complete pass through all phases of the software development, e.g. Requirements, Analysis, Design, Implementation, Testing for a use case increment leading to an executable release. A **product release** is a complete and consistent set of artifacts and includes a **software build**(an executable version of the system).

Use Case Increments and Build/Release Iterations

The OOPProject a number of use case increments. Each use case increment has a number of build/release iterations generally requiring 3 - 4 weeks of effort depending upon the size of the build/release.



These are the steps:

- 1 - Identify all use cases (name only)
- 2 - Group use cases together to identify use case increments
- 3 - In each use case increment, identify build/release iterations
- 4 - In each build/release iteration, identify all use case scenarios (name only)

OOProject Increment/Iteration Plan (3 - 4 Week Iteration)

Increment Name	Use Cases	Build/Release Iterations	Use Case Scenarios
Increment 1	Use Case 1, 2, 3	Iteration 1 Optimistic/Simple, Iteration 2 Normal/Moderate, Iteration 3 Pessimistic/Complex	Iteration 1 Optimistic/Simple: UC1Opt, UC2Opt, UC3Opt Iteration 2 Normal/Moderate: UC1Nor, UC2Nor, UC3Nor Iteration 3 Pessimistic/Complex: UC1Pess, UC2Pess, UC3Pess
Increment 2	Use Case 3, 4, 5	Iteration 4 Optimistic/Simple, Iteration 5 Normal/Moderate, Iteration 6 Pessimistic/Complex	Iteration 4 Optimistic/Simple: UC4Opt, UC5Opt, UC6Opt Iteration 5 Normal/Moderate: UC4Nor, UC5Nor, UC6Nor Iteration 6 Pessimistic/Complex: UC4Pess, UC5Pess, UC6Pess

Below is a sample Increment/Iteration Plan Except from Appendix A

Increment Name	Use Cases	Build/Release Iterations	Use Case Scenarios
Deposits and Withdraws	Checking Deposit, Checking Withdraw, Saving Deposit, Saving Withdraw	Deposit and Withdraw Optimistic/Simple, Deposit and Withdraw Normal/Moderate, Deposit and Withdraw Pessimistic/Complex	CheckingDepositOptimistic, CheckingWithdrawOptimistic, SavingDepositOptimistic, SavingWithdrawOptimistic CheckingDepositNormal, CheckingWithdrawNormal, SavingDepositNormal, SavingWithdrawNormal CheckingDepositPessimistic, CheckingWithdrawPessimistic, SavingDepositPessimistic, SavingWithdrawPessimistic
Inquiries and Transfers	Checking Inquiry, Checking Transfer, Saving Inquiry, Saving Transfer	Inquiries and Transfers Optimistic/Simple, Inquiries and Transfers Normal/Moderate, Inquiries and Transfers Pessimistic/Complex	CheckingInquiryOptimistic, CheckingTransferOptimistic, SavingInquiryOptimistic, SavingTransferOptimistic CheckingInquiryNormal, CheckingTransferNormal, SavingInquiryNormal, SavingTransferNormal CheckingInquiryPessimistic, CheckingTransferPessimistic, SavingInquiryPessimistic, SavingTransferPessimistic
Overdrafts	CheckingOverdraft, SavingOverdraft	Overdraft Optimistic/Simple Overdraft Normal/Moderate Overdraft Pessimistic/Complex	CheckingOverdraftOptimistic, SavingOverdraftOptimistic CheckingOverdraftOptimistic, SavingOverdraftNormal CheckingOverdraftOptimistic, SavingOverdraftPessimistic

For each Build/Release Iteration, the following is scheduling and monitoring table. The UML Model is the current model location, e.g. XYZ\F:UMLModels\Iteration1Model.mdl.

OOProject Schedule Status

	Iteration 1 - Optimistic/Simple	Iteration 2 - Normal/Moderate	Iteration 3 - Pessimistic/Complex
UML Model			
Planned Start			
Revised Start			
Actual Start			
Planned Completion			
Revised Completion			
Actual Completion/Review			
% Complete			
Model Review Date			
Date Build Approved			
Comment			

UML Model reviews are scheduled bi-weekly or at the end of each iteration. Periodically, we may schedule UML Model reviews within an iteration for Requirements, Analysis, Design, and/or Implementation. All the applicable UML diagrams and specifications shall be placed in the Project Directory available for project staff review and comments. Source code and test results for the iteration shall be available for staff review and comments. The model review shall consist of a brief presentation of major UML diagrams and issues.

Benefits of having using use case increments and build/release iterations are:

- supports effective planning and decision making with a "little by little" rather than "do it all at once - big bang" development approach
- reduces project risks because the client sees tangible results,
- supports effective change-management,
- supports creating a quality system with phased deliveries.

Requirements Use Case Specification

The Use Case Specification is one of the major specifications to document OOProject Requirements. For each OOProject Use Case, collect the following information: Name, Trigger, Input Parameters, Output Return, Precondition/Exception Raised, Postcondition/Exception Raised, Basic/Optimistic Scenario, Alternative/Pessimistic Scenarios, Business Rules, Test Cases. The following is a sample Use Case Specification.

Use Case Specification for WithdrawFromCheckingAccount Use Case

Use Case Name: WithdrawFromCheckingAccount

Trigger: WithdrawFromCheckingAccount

Input Parameters: sAcctNum, nWithdraw

Output Return: sText

Precondition: ValidAccount = true and nWithdraw <= nCurrentBalance

Precondition Exception Raised: To be determined

Description/Transformation: nCurrentBalance = nCurrentBalance - nWithdraw

Postcondition: nCurrentBalance < nOldBalance

Postcondition Exception: None

Basic Scenario/Optimistic Scenario: Text - to be determined; Diagram - see WithdrawFromCheckingAccount - Optimistic Scenario Sequence Diagram

Alternative Scenarios/Pessimistic Scenario: Text - to be determined - Diagram - see WithdrawFromCheckingAccount Activity Diagram

Business Rules: ValidAccountRule, AdequateBalanceRule

Test Cases: 1 - Optimistic:Inputs: sAcctNum - BGates001, nWithdraw - 100, nCurrentBalance - 1000 Conditions: None, Output: "BGates001 withdraw \$100 OK and recorded", 2 ... To be determined

Input and Output Forms: See below

Input/Output Forms for WithdrawFromCheckingAccount Use Case:

Withdraw Request Form

Customer Account Number _____

Withdraw Amount _____

Button-Submit

Button-Clear

Withdraw Response Form

Customer Account Number _____

Withdraw Amount _____

Status _____

Button-OK

Benefits of having a well-defined use case specification form are:

- supports consistency in modeling use cases,
- supports completeness especially to identify precondition, postconditions, and business rules,
- is useful to interview domain experts.

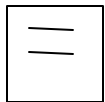
Unified Process Models in Construction Phase

In the construction phase, we create the major UML diagrams and specifications in the Unified Process are shown below:

System/Subsystem/Component

I - Requirements

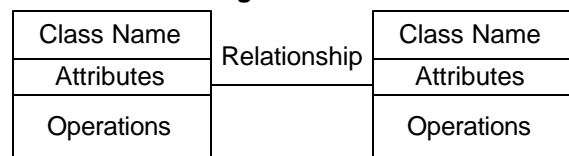
Requirements Statement/Product Capabilities



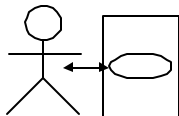
Package/Class/Object

II - Analysis

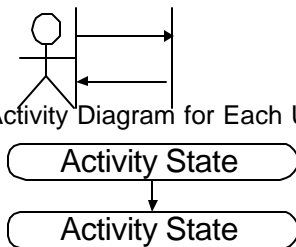
Class Diagram



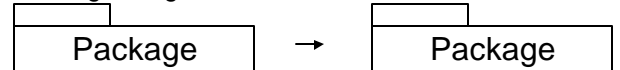
Use Case Diagram for All Use Cases



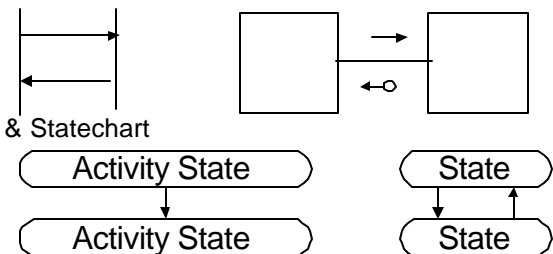
Sequence Diagram for Each Use Case Course



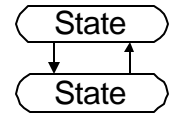
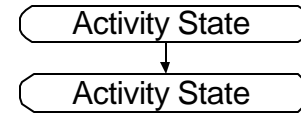
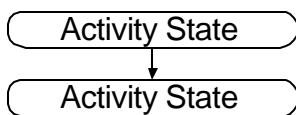
Package Diagram



Sequence and/or Collaboration Diagram



Activity Diagram for Each Use Case All Courses Activity Diagram & Statechart

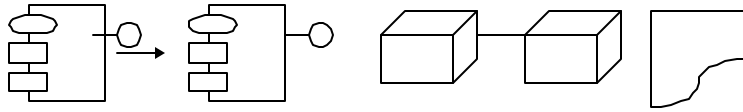


IV - Implementation

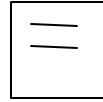
Processing Environment HW & SW
Component Diagram & Deployment Diagram
& Code

III - Design

Processing Environment HW & SW
Updated Class/Package/Sequence/
Collaboration/Activity/Statecharts



V Testing/Deployment →



Key aspects of these models in the Unified Process - Construction Phase are shown below. The key is to create all diagrams in these models for each build/release iteration (3 - 4 weeks).

	Requirements Model	Analysis Model	Design Model	Implementation Model	Testing Model
Key UML Elements	System, Actor, Use Case, Interaction	Business Package, Class, Object, Message	HW & SW Configuration, Package, Class Object, Message	Component, Node, Code	Test Plan and Test Cases
Key Concern	Model System as a Black Box	Model Business Elements in the Problem Domain with no implementation details	Update Analysis Diagrams/Specifications for a specific implementation, e.g. HW & SW Configuration.	Model physical elements for the distributed environment; Code to meet all requirements	Unit (Class/Operation) Tests, Integration/Overall System Tests
Objective - weak coupling - strong cohesion among elements	Sufficient information on all use cases/scenarios. All increments/iterations planned.	Simplest Business/Problem Domain Model to meet requirements	Sufficient information to generate maximum code or manually code	Optimum Component Architecture - Network friendly; Code that meets all requirements	Sufficient Testing that code meets all requirements
Static/Structural Diagrams	Block Diagram and Use Case Diagram Showing Actors	Package/Class Diagram	Package/Class Diagram	Component and Deployment Diagrams; Reversed Class Diagrams	--
Dynamic/Time Based Diagrams	Use Case Diagram, Sequence Diagram for each use case scenario, Activity Diagram for each use case	Sequence Diagram for each use case scenario, Statechart for each state-based class, Activity Diagram for each complex operation	Sequence Diagram for each use case scenario, Statechart for each state-based class, Activity Diagram for each complex operation	Optionally update sequence diagrams showing distributed messages	--
Tools	UML CASE, Requirements Tracking, CM	UML CASE, Requirements Tracking, CM	UML CASE, Requirements Tracking, CM	UML CASE, Requirements Tracking, CM, Testing	CM
Key Team Players	Business/System Analysts	Business/System Analyst	Developer	Architect, Developer	Developer/Tester
Model Sign-off	Project Manager, Architect, Client/User	Project Manager, Architect, Client/User	Project Manager, Architect	Project Manager, Architect	Project Manager, Architect, Client/User for

					Acceptance
--	--	--	--	--	------------

Metrics and Monitoring

Metrics provide a quantitative measure to monitor progress, make estimates, identify risks, and to identify high risk complex entities. Metrics contribute to effective project management and creating quality systems. There should be automated metric collection with the CASE tool and code analyzers. “If you can’t measure it, it’s not engineering.”

Management Metrics provide information on project schedule, resources, and other management concerns in terms of planned versus actual values. Sample project metrics: milestones completed, assigned people, costs, use case scenarios, key classes, support classes (GUI, collections, etc), packages per system, person-days per class, classes per developer, development iterations, etc. See Lorenz and Kidd *Object-Oriented Software Metrics*.

Project Metrics provide information on the system, packages, classes, and other elements. Project metrics are valuable to show changes over time and to indicate high risk complex elements.

Sample Project Metrics

	System Level Metrics - Number in the System	Class/Object Level Metrics - Number in the System	Class/Object Level Metrics - Number of & Average Number	Code Metrics - Number of & Average Number
Size Metrics	Analysis: Requirements, Actors, Component In Messages, Component Out Messages, Component Input Objects/Data, Component Output Objects/Data, Use Cases, Use Case Scenarios Design: Executable Components, Messages between Components, Nodes, Links between Nodes	Packages, Classes, Interfaces, Operation, Attributes, Relationships, Objects, Messages, States, Transitions, Exception Classes, Reused Classes		Lines of Code/System, LOC/Class, LOC/Operation LOC refers to NCSS - Non-comment Source Statements
Complexity Metrics Higher Ratio Suggests Greater Complexity	Use Case Scenarios/Use Case	Levels of Generalization,	Classes & Interfaces/Package, Attributes/Class, Operations/Class, Relationships/Class, Message Sends/Class, Message Sends/Operation, Parameters/Operation, Subclasses/Superclass	Weighted Operations/Class, McCabe Cyclomatic Complexity/Operation, Halsted/Operation, Length * (Fan-in * Fan-out) ²
Reuse Metrics	Reused Patterns, Reused Components	Reused Patterns, Reused Packages, Reused Classes		
Quality Metrics	Defects	Defects	Defects	Defects

Our goal is to use CASE and other tool monitoring. The following automatically generated tables will be used. As required more detailed reports may be generated.

Size Metrics - Number of UML Elements - CASE Tool Generated

	Iteration 1 - Optimistic/Simple	Iteration 2 - Normal/Moderate	Iteration 3 - Pessimistic/Complex
Actors			
Use Cases			
Use Case Scenarios			

Packages			
Classes			
Interfaces			
Attributes			
Operation			
Generalization Relationships			
Realizes Relationships			
Composition Relationships			
Shared Aggregation Relationships			
Dependency Relationships			
Objects			
Messages			
States			
Transitions			
Components			
Component Dependencies			
Nodes			
Node Links			
Test Cases			
Total SLOC-Source Lines of Code			
SLOC per Class			
SLOC per Operation			

Complexity Measures - CASE or Other Tool Generated Min/Max/Average Provided in Each Cell

	Iteration 1 - Optimistic/Simple	Iteration 2 - Normal/Moderate	Iteration 3 - Pessimistic/Complex
Classes & Interfaces per Package			
Attributes per Class			
Operations per Class			
Parameters per Operation			
Message Sends/ per Operation			
Message Sends per Class			
Relationships per Class			
Weighted Operations per Class			
McCabe Cyclomatic Complexity per Operation			
Halsted per Operation			
Length * (Fan-in * Fan-out) ²			

Benefits of having project metrics and monitoring are:

- supports communication so that team members are working on the latest version,
- supports identifying risks and problems early to meet cost, schedule, and other objectives,
- supports efficiency by reducing redundant efforts,
- supports creating a quality system in which all parts fit together.

Reuse

Our goal is to promote reuse in the OOProject. There are several forms of reuse as documented by Scott Ambler in *Building Object Applications That Work* (SIGS Books, 1997):

Operation Reuse is the reuse of complex operations, such as utility operation or complex algorithmic operations.

Class Reuse is the reuse of classes. Class reuse is accomplished by sharing common classes or collections of functions and procedures. Class reuse leads to code reuse.

Inheritance Reuse is to use inheritance to take advantage of behavior implemented in existing classes.

Template Reuse is typically a form of documentation reuse. It refers to using a common set of layouts for key development artifacts—documents, models, and source code.

Component Reuse is the use of pre-built, fully encapsulated components. Examples of components are Java Beans and ActiveX components.

Pattern Reuse is the use of documented patterns such as documented in *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Patterns in Java* Vol 1 and 2 by Mark Grand, *A System of Patterns* by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, and other books.

Framework Reuse is the use of collections of classes that implement the basic functionality of a common technical or business domain together. Examples of frameworks are the San Francisco Framework.

Artifact Reuse is to use of previously created development artifacts—use cases, standards documents, domain-specific models, procedures and guidelines, and other applications.

In the OOProject we will maintain the Reuse Table.

Reuse by Iteration

	Iteration 1 - Optimistic/Simple	Iteration 2 - Normal/Moderate	Iteration 3 - Pessimistic/Complex
Operations from Class/Operation Library			
Classes from Class/Operation Library			
Patterns from Pattern/Collaboration Library			

Benefits of reuse are:

- deduces effort and resources since a reused element has already been documented, constructed, and tested,
- supports high quality system based upon high quality reused elements,

Quality Assurance and Testing

Quality Assurance is to ensure adequate processes, resources, and management to create quality products within required constraints that are defect-free. Quality factors are reliability, correctness, extensibility, reusability, portability, maintainability, understandability, usability, etc. Quality Assurance Activities primarily for correctness and defect discover include:

UML Model Reviews - these include the presentation of diagrams and specifications.

Walkthroughs (Optional) - a role-playing technique to check the completeness and consistency of O-O models. A person represents the system, an actor, object, or other element. Starting with a system in message complete use case scenarios are traced through the system leading to system out messages.

CASE Tool Checks - automated checks for consistency and completeness of diagrams and specifications.

Code Inspections - examining source code and reversed class diagrams.

Testing - executing a system, component, class, operation or other element with test cases to validate that the element accomplishes requirements and to verify correctness.

Correctness Proofs (Optional) - using formal method with mathematical formalisms to establish the correctness of analysis/design models and code.

QA Activities During the Construction Phase

QA Activities	Requirements Model	Analysis Model	Design Model	Implementation Model	Testing
QA Planning	Project Plan	Project Plan	Project Plan	Project Plan, Coding Standards	Testing Plan,
UML Model Reviews & Inspections	System Requirements Project Plan See Requirement Model Checklist	Analysis Models of Diagrams & Specifications See Analysis Model Checklist	Design Models of Diagrams & Specifications See Design Model Checklist	CASE Tool Scripts, Test Cases, Code Inspection See Implementation Model Checklist	Test Cases, Code Inspection See Testing Checklist
Tool Checks	Requirements Traceability	CASE Tool Checks of Analysis Models	CASE Tool Checks of Design Models	Compiler, Code Analyzer, CASE Reverse Engineering, Testing Tools	Testing Tools for Unit Tests, System Tests, Acceptance Tests

Operation Specification

Operation Specification is a key specification that is useful to support correctness. Below is a sample Operation Specification:

Use Case Name: withdraw

Trigger: withdraw

Input Parameters: nWithdraw : int

Output Return: boolean

Precondition: nWithdraw <= nCurrentBalance

Precondition Exception Raised: exInsufficientFunds

Description/Transformation: nCurrentBalance = nCurrentBalance - nWithdraw

Postcondition: nCurrentBalance < priorCurrentBalance

Postcondition Exception: exIncorrectBalance

Basic Scenario/Optimistic Scenario: See withdrawFromCheckingAccount Sequence Diagram

Alternative Scenarios/Pessimistic Scenario: See withdrawFromCheckingAccount Activity Diagram

Business Rules: ValidAccountRule, AdequateBalanceRule

Reversed Engineered Class Diagrams for each Build/Release

Reverse Engineering is to create UML diagrams and specifications from source code, e.g. Java, C++, etc. The class diagram shall be automatically created from source code by the CASE tool. Other diagrams, e.g. use case, sequence, state,

etc, must be manually created using the reversed class diagram and interviewing domain experts. The following are the steps to develop a Reverse Engineered UML Model for each Build/Release Iteration

- 1 - Select one or more UML CASE tools to reverse engineer OOProject source code. Set the reverse options in the UML CASE tool.
- 2 - Collect the source code for the build/release.
- 3 - For each directory/package reverse engineer the source code to create the reversed class diagram.
- 4 - Verify and update the reversed class diagram to ensure that the diagram is accurate showing classes, attributes, operations, and relationships (realization, generalization, association, shared aggregation, composition, and dependency).
- 5 - Create a glossary - data dictionary listing and defining all major terms and other reports from the reversed class diagram.
- 6 - After examining the reversed class diagram and reports, create a list of recommended code changes for correctness, compliance with coding standards, etc.

Benefits of Reverse Engineering are:

- visually displays hard-to-read code,
- identifies poorly written code early, e.g. spaghetti code,
- promotes following project coding standards, e.g. capitalization, prefixes, naming, etc,
- improves the quality of code.

Testing

Testing shall occur throughout the project lifecycle. As presented in *The Rational Unified Process An Introduction Second Edition* by Krutchen, there are the following testing dimensions:

- Quality: Reliability, Functionality - required use cases, Performance.
- Stages of testing:
 - Unit Tests - smallest testable elements of the system are tested individually, e.g. component, collaboration, class, operation
 - Integration Tests - integrated units (components or subsystems) are tested
 - System Test - complete system is tested - end to end
 - Acceptance Test - complete system tested by end users to ensure readiness for deployment
- Types of tests:
 - Benchmark Test
 - Configuration Test
 - Function Test
 - Installation Test
 - Integrity Test
 - Load Test
 - Performance Test

The OOProject Test Plan shall include the following:

- Test Cases - the set of test inputs, conditions, and expected results - See Test Specification below.
- Test Procedures - the set of "how to" setup, execute, and evaluate test results .
- Test Scripts - high level programs for automated testing, e.g. Testing Tool Script
- Test Classes and Components - drivers, stubs, and other programs for testing

The following is the OOProject Test Case Specification:

Test Case Specification:

Test Use Case Name:

Use Case Name:

Use Case Scenario Name:
 Trigger:
 Input Parameters:
 Output Return:
 Precondition:
 Precondition Exception Raised:
 Description/Transformation:
 Postcondition:
 Postcondition Exception:
 Comments:

Testing by Iteration - Planned/Completed/% in each cell

	Reference/ Location	Iteration 1 - Optimistic/Simple	Iteration 2 - Normal/Moderate	Iteration 3 - Pessimistic/Complex
Unit Tests - Operations				
Unit Tests - Class				
Unit Tests - Component				
Integration Tests				
System Tests - End to End				
User Acceptance Test				
Quality factor comments: reliability, correctness, extensibility, reusability, portability, maintainability, understandability, usability				

Benefits of testing are:

- supports identifying defects early thereby reducing the costing of fixing each defect,
- supports identifying risks and problems early,
- supports the proper interaction and integration of components,
- supports creating a quality defect-free system.

Summary

This project plan is to assist all project team members to work toward the successful completion of the project, to create a defect-free software product, and to ensure a satisfied customer.

Project Plan Approval:

Project Manager Approval & Date

Architect Approval & Date

Approval & Date

Appendices

Tasks to Create a Complete UML Model

Based upon *The Rational Unified Process An Introduction* Second Edition by Krutchen

0 - Business Modeling - Enterprise Level

Review the Business Model: Vision Document, Organization Chart, Business Events and Processes (Use Cases), Business Actors, Workers, and Entities (Domain Model), Business Rules Catalog, Business Interfaces (Set of Operations), Business Patterns, Business Systems Architecture - Component Diagram, Glossary. See *The Rational Unified Process An Introduction* Second Edition by Krutchen and *Business Modeling with UML* by Eriksson and Penker.

I - Requirements - System/Subsystem/Component Level (Implementation Language Independent)

1 - **Requirements** - Review the Requirements Statement, System Drawing, and System Block Diagram (Customer Provided). Optionally, create a System Collaboration Diagram (Context Diagram) showing object/data inputs and outputs to set the system boundary. Optionally, create the Requirements Traceability Table listing Requirement Name, Number, Reference, Use Case, UML Element, Test Case, Description, Responsibility, etc. Optionally, review management plans, schedules, risks, naming/coding standards, methodology plans - UML process/stereotypes/properties/constraints.

2 - **All Use Cases** - Create the **Use Case Diagram** showing all use cases. Optionally, show use case relationships (includes, extends, generalization) and/or hierarchical use cases (high level to detailed). Identify the use case increments and iterations.

3 - **Each Use Case** - Create a **Use Case Specification** for each use case stating use case name, trigger, inputs, outputs, precondition/exception, postcondition/exception, basic and alternative scenarios (optimistic to pessimistic), business rules. Optionally, create a **Sequence Diagram** for each use case scenario. Optionally, create input/output forms and test case for each use case.

4 - **Each Use Case All Scenarios** - Create an **Activity Diagram** for each use case showing all use case scenarios (optimistic to pessimistic). Optionally, show all scenarios/paths for a use case in text, flow chart, or other diagram.

5 - **Information** - Create the **Product Capabilities** listing non-functional requirements: usability (reliability, performance, security, human factors), generality (portability, compatibility), timing, space, memory, etc. Optionally create a semantic data model (High Order Concept Model (HOCM)) showing all major elements/concepts inside and outside the system to be the basis for Class/Package Diagram.

II - Analysis - Class and Object Level (Implementation Language Independent)

6 - **Classes and Packages** - Create the list of candidate classes, **CRC Cards** (Class Responsibility Collaboration), **Class Diagram**, and **Package Diagram**. Optionally create an **Object Diagram** showing attribute values.

7 - **Objects and Messages** - Create a **Sequence Diagram** for each use case - optimistic scenario. Optionally, create a **Sequence Diagram** for all other use case scenarios. Alternatively, create a **Collaboration Diagram** for each use case scenario.

8 - **States and Transitions** - Create a **State Diagram** for each state-based class showing states, events, conditions, and actions.

9 - **Operations** - Create an **Operation Specification** showing preconditions, transformations, postconditions, and exceptions for each complex operation. Optionally, create an **Activity Diagram** for each complex operation showing the sequence of activity states, conditions, and actions.

III - Design - Class and Object Level for a Specific Processing Environment

10 - **Processing Environment** - Create the processing environment consisting of the planned Implementation H/W and S/W Configuration List: operating system, language, class libraries, components, GUI, distribution - object request broker, persistent data storage, etc. Optionally, list potential patterns, component standard (Active X, Java Bean, CORBA), naming conventions, coding standards, code generation scripts, tools (CASE, compiler, configuration management, testing, etc).

11 - **Updated Analysis Diagrams and Specifications** - Update all diagrams and specifications to add detail for the Processing Environment including data types, visibility, parameters/returns, support classes, operation detail (precondition/exception, transformation, postcondition/exception), etc. Optionally provide implementation of patterns, e.g. polymorphic operations, exceptions (exception superclass/subclasses), threads, data access, transactions, security, message queuing, etc. Goal: diagrams and specifications provide adequate information for manual coding or code generation.

IV - Implementation - System/Subsystem/Component Level for a Specific Processing Environment

12 - **Implementation Processing Environment, Component Standard and Component Patterns** - Update the Processing Environment to show provided components (GUI, data access, transactions, distribution, message queuing, security, etc), component standard (Active X, Enterprise Java Bean, CORBA), and component to component patterns, e.g. small single operation component, class based component, session per user - entity component, package based component, large multi-package component.

13 - **Implementation Components** - Create the **Component Diagram** showing all required components and files with the dependency relationship, e.g. .EXE, DLL, .OCX, .LIB, .TXT, .HLP, etc. Optionally show interfaces (lollipops) and create an **interface diagram** showing exposed operations. Optionally show IDL (Interface Definition Language) code, e.g. CORBA IDL, Microsoft IDL, Java.

14 - **Implementation Nodes (Processors and Devices)** - Create the **Deployment Diagram** showing all required processors, devices, and other equipment, e.g. client network computer, Windows PC, NT Server, Transaction Server, Web Server, Mail Server, Fax, Printer, Network, etc.

V - Construction

15 - **Coding Standards and Code Generation Scripts** - Update coding standards and code generation scripts. Coding standards list sample code showing code for all major UML elements and relationships and policy for inheritance, interfaces, exceptions, threads, etc.

16 - **Code Each Component and Reverse Engineer Diagrams**

VI - Testing

17 - **Testing Plan** - Update the Testing Plan to list test cases (name, input, output, conditions), test procedures (step by step instructions for each test case), test components (drivers, harnesses, scripts).

18 - **Tests** - Conduct tests, e.g. class/operation tests (Unit Tests), individual component tests (use case based), overall system - multiple component tests (use case based integration/acceptance testing).

VII - Model and Construct Other Components As Required

19 - **GUI/User/External Interface Components** - Optional - Create the GUI component (windows, menus, dialog boxes, panels).

20 - **Persistent Storage Components** - Optional - Create the persistent storage component - data storage tables/stored procedures/triggers.

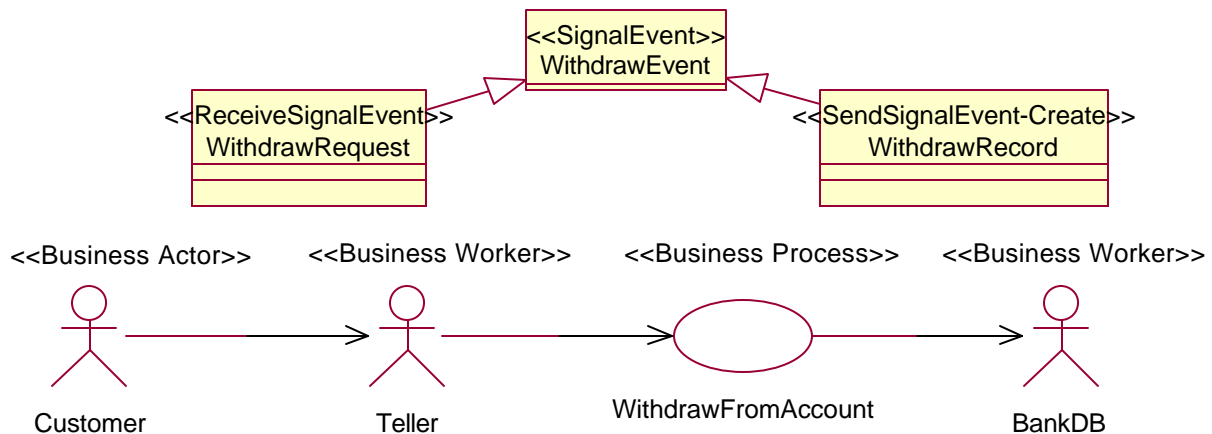
UML CASE Study - BankApp with Rational Rose

Preliminary - Business Enterprise Models

Business Vision, Objectives, and Organization - Provided Separately

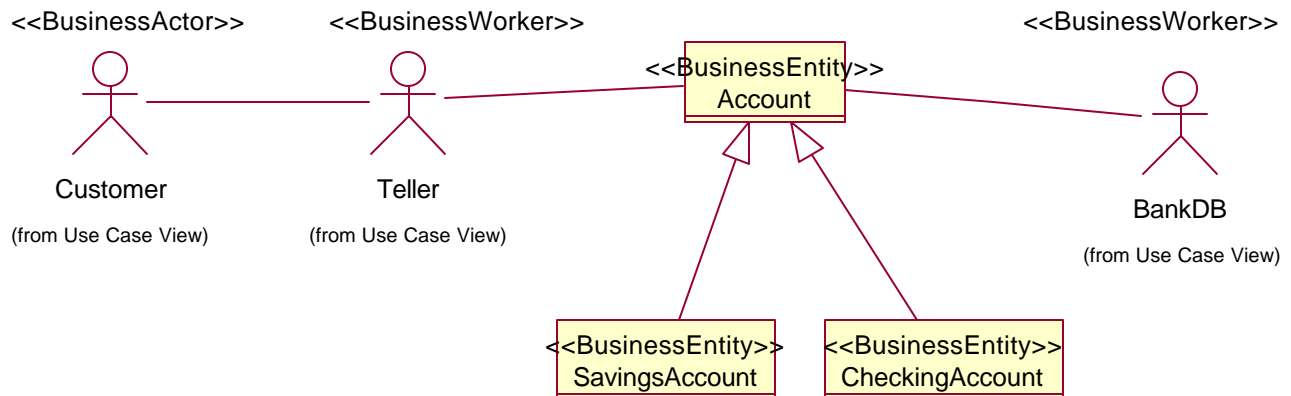
Business Events (UML Signal Events -Named Stimulus Form or Document) and Processes (UML Use Cases)

Process Name	Actors	Events/Inputs	Transformation	Events/Output	Constraints	Description	Reference	Point of Contact
Withdraw From Account	Customer, Teller, BankDB	Withdraw Request	Update Account	Withdraw Record				



Business Actors, Business Workers, and Business Entities (Problem Domain Entities)

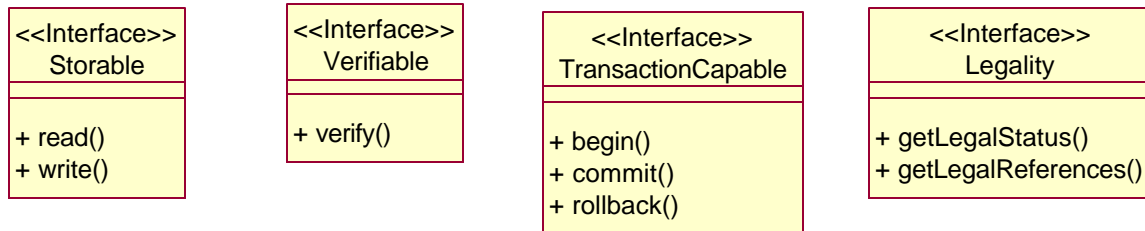
Business Actor (UML Actor)	Business Worker (UML Actor)	Business Entity (UML Class)
Customer	Teller	Account, SavingsAccount, CheckingAccount



Business Rules Catalog

Rule Identifier	Actor, Entity, Process	Description:If Conditional..Then Action	Areas	Reference	Point of Contact
ValidAccount	Account	If AccountNum is Valid then Account is Valid			

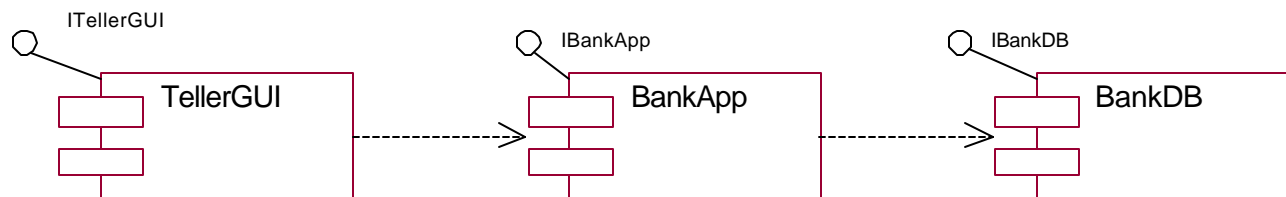
Business Interfaces (Named Set of Operations) - Provided by Architect



Business Patterns Catalog - See Business Modeling with UML by Eriksson and Penker

Business Glossary - to be completed

Business Systems Architecture - Provided by Architect



I - Requirements Models

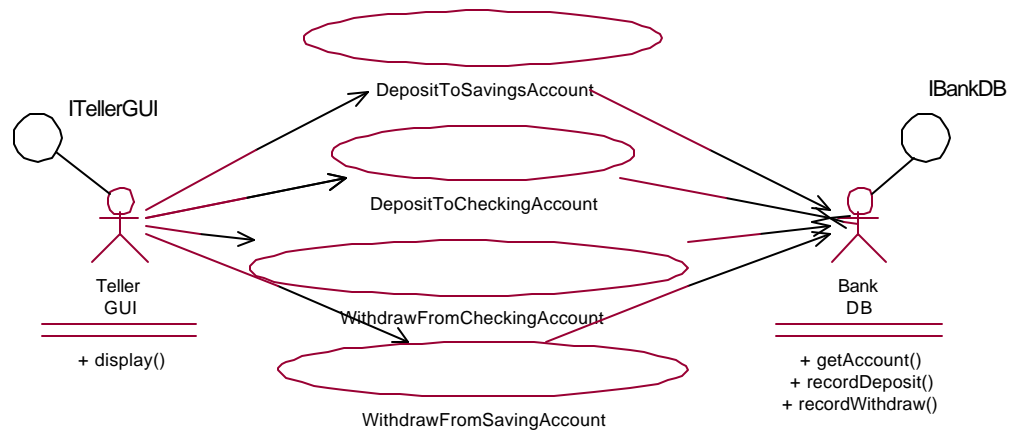
1 - Requirements: The BankApp shall manage checking and savings account deposits and withdraws. Future: inquiries, transfers, overdraft, etc. **Inputs/Outputs** TellerGUI to BankApp: sAcctNum, nDeposit, nWithdraw, sText. Inputs/Outputs BankApp to BankDB: sAcctNum, nDeposit, nWithdraw.

1 - Requirements Traceability Table: Requirement Number, Name, Reference, Use Case Name, UML Element, Test Case, Description, Responsibility.

Requirements Traceability Table (Partial)

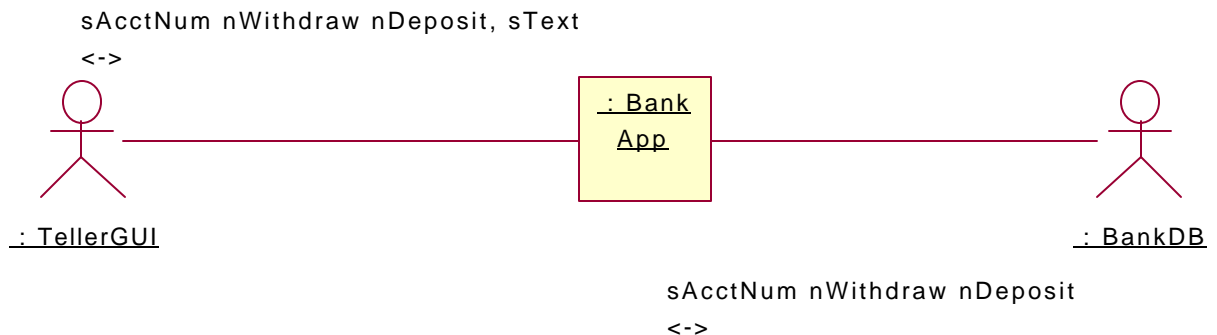
Requirement Number	Requirement Name	Reference	Use Case Name	UML Element	Test Case	Description	Responsibility
1.1	DepositToSavingsAccount		DepositToSaving sAccount	BankPkg			
1.2	DepositToCheckingAccount		DepositToChecki ngAccount	BankPkg			
1.3	WithdrawFromSaving Account		WithdrawFromSa vingAccount	BankPkg			
1.4	WithdrawFromChecki ngAccount		WithdrawFromCh eckingAccount	BankPkg			

2 - Requirements - Use Case Diagram - All Use Cases



Rose Use Case Diagram: In Browser Window select Use Case View; Rename Main to be Use Case Diagram; Place actors, use cases, and relationships (Rose Unidirectional Association and Generalization) on the diagram; Select each actor - right mouse to enter actor operations; Select Tools - Check Model; Select File - Save.

2 - Requirements - High Level Collaboration Diagram (Context Diagram)



Rose High Level Collaboration Diagram: In Browser Window select Use Case View; Select Browse - Interaction Diagram - Use Case View - <New>; Select Collaboration Diagram; Enter Diagram Name; Place objects representing actors on the diagram; Double-click each object then select the actor name from the pull-down list; Place one object in the center of the diagram to represent the system; Double-click the object and enter the system name; Select Rose Object Link symbol and drag between actors and the system ; Select the Rose Text Box "ABC" and enter names of passed objects/data; Select Tools - Check Model; Select File - Save.

2 - Requirements - Use Case Increments:

Increment 1: Checking and Saving Account Deposits and Withdraws

Increment 2: Inquiries and Transfers

Increment 3: Overdrafts

Iterations within each Increment: optimistic, normal, pessimistic

Increment Name	Use Cases	Build/Release Iterations	Use Case Scenarios
Deposits and	Checking Deposit,	Deposit and Withdraw	CheckingDepositOptimistic, CheckingWithdrawOptimistic,

Withdraws	Checking Withdraw, Saving Deposit, Saving Withdraw	Optimistic/Simple, Deposit and Withdraw Normal/Moderate, Deposit and Withdraw Pessimistic/Complex	SavingDepositOptimistic, SavingWithdrawOptimistic CheckingDepositNormal, CheckingWithdrawNormal, SavingDepositNormal, SavingWithdrawNormal CheckingDepositPessimistic, CheckingWithdrawPessimistic, SavingDepositPessimistic, SavingWithdrawPessimistic
Inquiries and Transfers	Checking Inquiry, Checking Transfer, Saving Inquiry, Saving Transfer	Inquiries and Transfers Optimistic/Simple, Inquiries and Transfers Normal/Moderate, Inquiries and Transfers Pessimistic/Complex	CheckingInquiryOptimistic, CheckingTransferOptimistic, SavingInquiryOptimistic, SavingTransferOptimistic CheckingInquiryNormal, CheckingTransferNormal, SavingInquiryNormal, SavingTransferNormal CheckingInquiryPessimistic, CheckingTransferPessimistic, SavingInquiryPessimistic, SavingTransferPessimistic
Overdrafts	CheckingOverdraft, SavingOverdraft	Overdraft Optimistic/Simple Overdraft Normal/Moderate Overdraft Pessimistic/Complex	CheckingOverdraftOptimistic, SavingOverdraftOptimistic CheckingOverdraftOptimistic, SavingOverdraftNormal CheckingOverdraftOptimistic, SavingOverdraftPessimistic

3 - Requirements Use Case Specification: Name, Trigger, Input Parameters, Output Return, Precondition/Exception Raised, Postcondition/Exception Raised, Basic/Optimistic Scenario, Alternative/Pessimistic Scenarios, Business Rules, Test Cases

Use Case Specification for WithdrawFromCheckingAccount Use Case

Use Case Name : WithdrawFromCheckingAccount

Trigger: WithdrawFromCheckingAccount

Input Parameters: sAcctNum, nWithdraw

Output Return: sText

Precondition: ValidAccount = true and nWithdraw <= nCurrentBalance

Precondition Exception Raised: To be determined

Description/Transformation: nCurrentBalance = nCurrentBalance - nWithdraw

Postcondition: nCurrentBalance < nOldBalance

Postcondition Exception: None

Related Use Cases: Generalization, Includes, Extends/Extension Point: None

Basic Scenario/Optimistic Scenario: Text - to be determined; Diagram - see WithdrawFromCheckingAccount -

Optimistic Scenario Sequence Diagram

Alternative Scenarios/Pessimistic Scenario: Text - to be determined - Diagram - see WithdrawFromCheckingAccount Activity Diagram

Business Rules: ValidAccountRule, AdequateBalanceRule

Test Cases: 1 - Optimistic:Inputs: sAcctNum - BGates001, nWithdraw - 100, nCurrentBalance - 1000 Conditions: None, Output: "BGates001 withdraw \$100 OK and recorded", 2 ... To be determined

Input and Output Forms : See below

Input/Output Forms for WithdrawFromCheckingAccount Use Case:

Withdraw Request Form

Customer Account Number _____

Withdraw Amount _____

Button-Submit

Button-Clear

Withdraw Response Form

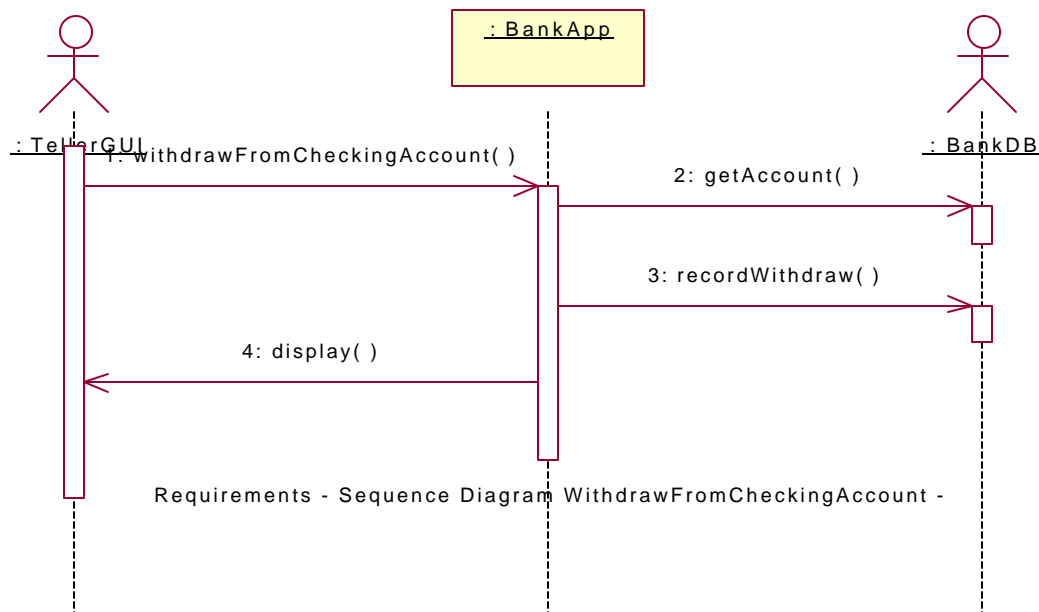
Customer Account Number _____

Withdraw Amount _____

Status _____

Button-OK

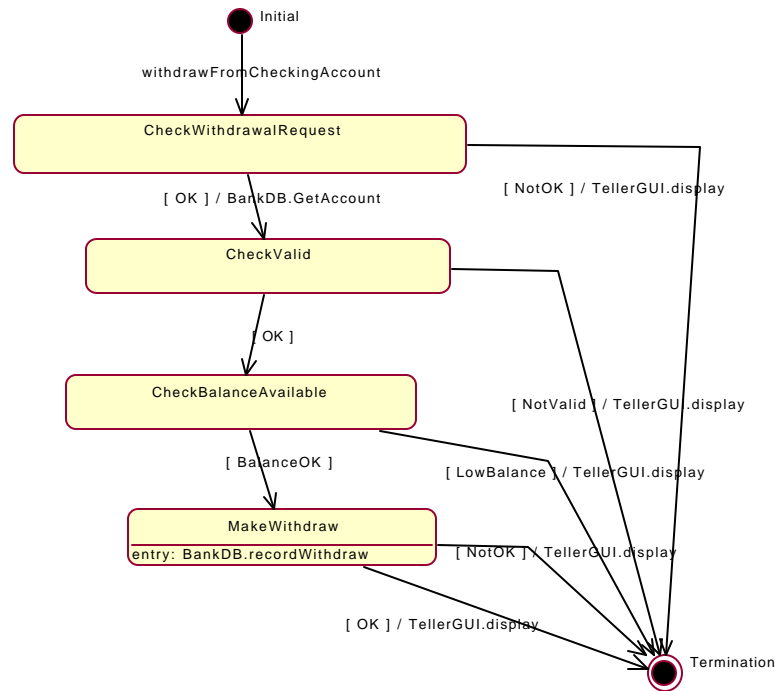
3 - Requirements - Sequence Diagram WithdrawFromCheckingAccount - Optimistic Scenario



Note: getApp required to get the top level application for the first transaction

Rose Sequence Diagram: In Browser Window select Use Case View; Select the Use Case Diagram; Select a use case; Select Browse - Interaction Diagram - Use Case View - <New>; Select Sequence Diagram; Enter Diagram Name, e.g. WithdrawFromCheckingAccount-OptimisticScenario; Place objects representing actors on the diagram; Double-click each object then select the actor name from the pull-down list; Place one object in the center of the diagram to represent the system; Double-click the object and enter the system name from the pull-down list; Select Rose Object Message symbol and drag between actors and the system; Select Tools - Check Model; Select File - Save.

4 - Requirements Activity Diagram for WithdrawFromCheckingAccount Use Case - All Scenarios

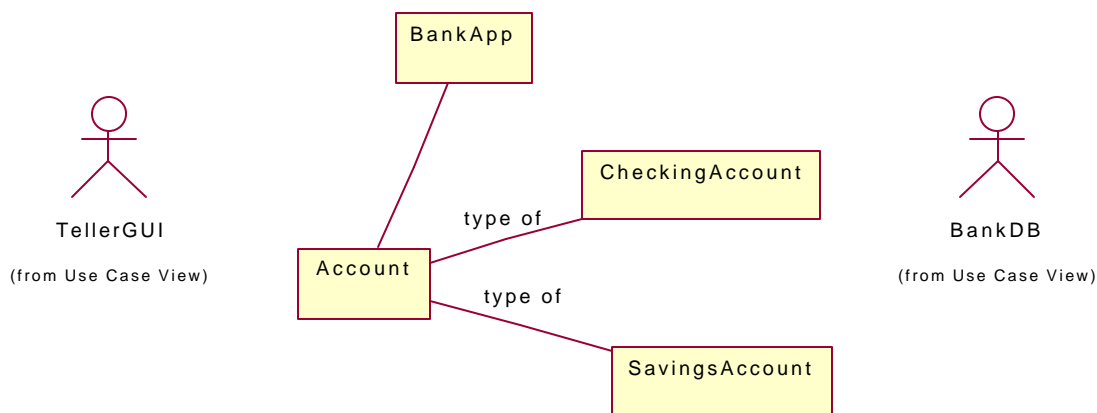


Note: getApp required to get the top level application for the first transaction

Rose State - Activity Diagram: In Browser Window select Use Case View; Select the Use Case Diagram to display the diagram; Select a use case; Select Browse - State Diagram; If “State Diagram” is grayed out, then go back to the use case diagram and re-select a use case; Place activity states on the diagram; Place transitions on the diagram by dragging between states; Select Tools - Check Model; Select File - Save.

5 -Requirements - Product Capabilities: High Reliability, 10 concurrent users, 2 second response time.

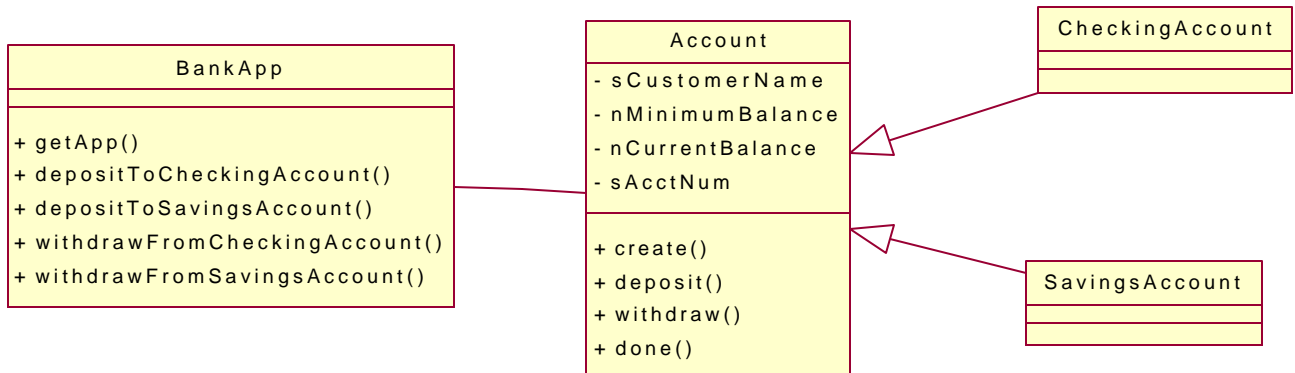
5 - Requirements/Analysis - High Order Concept Model: External Actors: TellerGUI, BankDB
Internal Entities: BankApp, Account, CheckingAccount, SavingsAccount



Rose High Level Concept Model Diagram: Recommend do the HOCM with pencil and paper. Alternatively, create a Rose Class Diagram without attributes and operations.

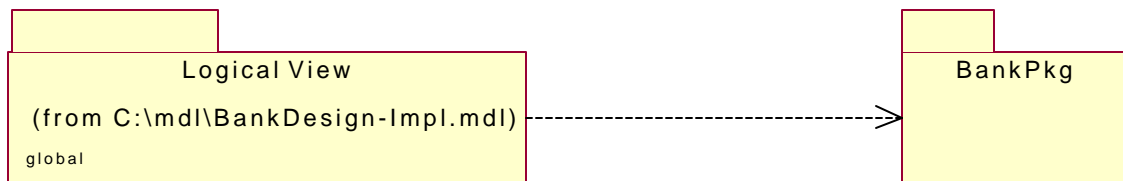
II - Analysis Model - Rough Sketch

6 - Analysis Class Diagram - Simplest Structure



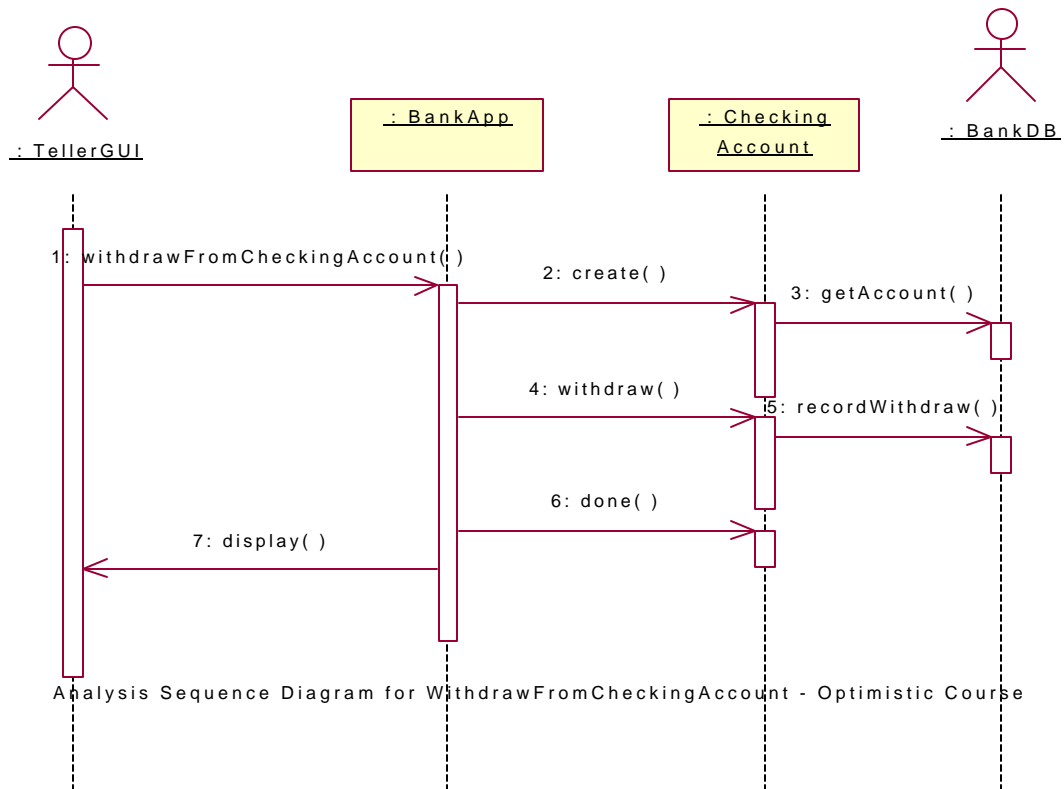
Rose Class Diagram: See Using Rational Rose

6 - Analysis Package Diagram



Rose Package Diagram: In Browser Window select Logical View; Select Browse - Class Diagram - Logical View - <New>. Enter the Package Diagram Name; Place packages on the diagram; To place a dependency relationship, select the dependency arrow from the Toolbar then drag from the source package to the destination package; In the Browser drag each class to the appropriate package; Select Tools - Check Model; Select File - Save.

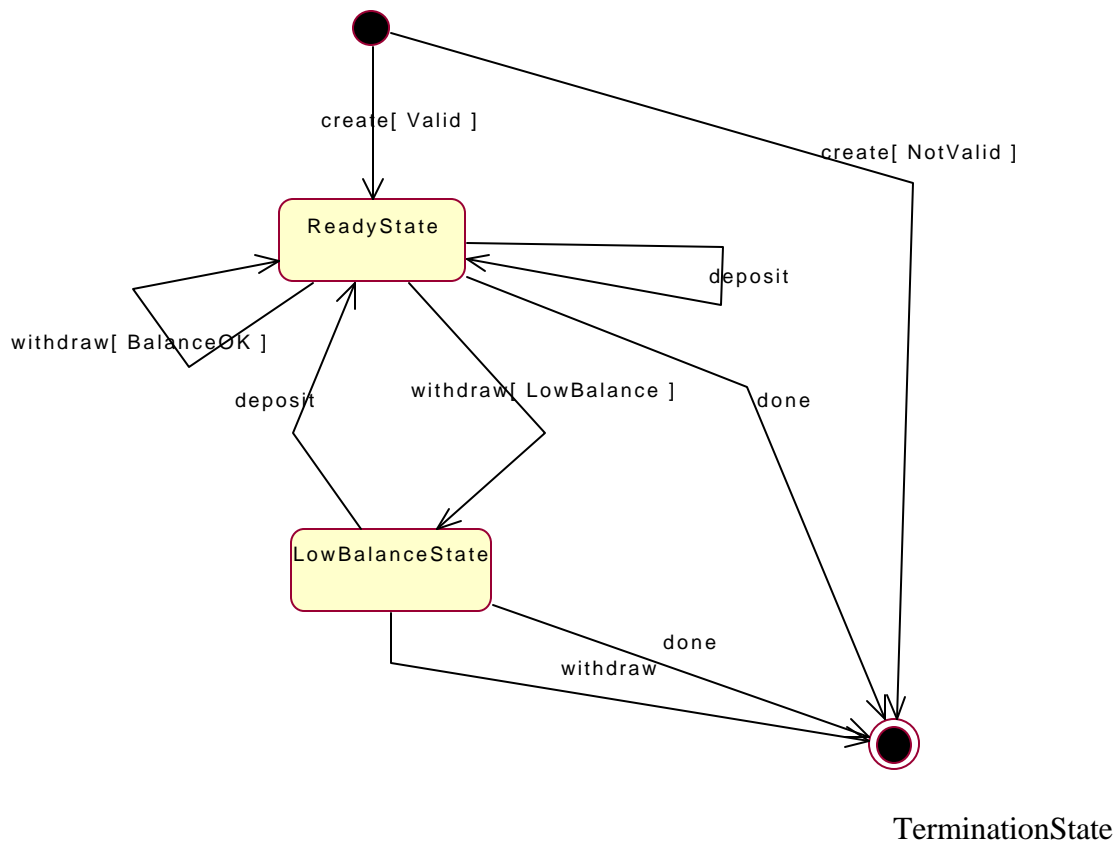
7 - Analysis Sequence Diagram for WithdrawFromCheckingAccount - Optimistic Scenario



Note: getApp required to get the top level application for the first transaction

Rose Sequence Diagram: See Using Rational Rose

8 - Analysis State Diagram for Account Class



Rose State - Activity Diagram: In Browser Window select Logical View; Select the Class Diagram to display the diagram; Select a class; Select Browse - State Diagram; If “State Diagram” is grayed out, then go back to the class diagram and re-select a class; Place states on the diagram; Place transitions on the diagram by dragging between states; Select Tools - Check Model; Select File - Save.

9 - Analysis - Complex Operations : To be determined - Activity Diagram and/or Operation Specification for each operation: name, inputs, precondition/exception, transformation, postcondition/exception, business rules, description

Rose Specifications: Display the class diagram; Select a class; Press the Right Mouse Button to display the Specification Dialog Box; Select a tab, e.g. Operations Tab; Double-click an operation; Fill-in the operation information. Select Tools - Check Model; Select File - Save.

III - Design Models - Basis for Coding

10 - Design Processing Environment: Linux Version 6.2, GNU C++ Version 6.2, C++ Standard Library, CORBA 3.

10 - Design Potential Patterns:

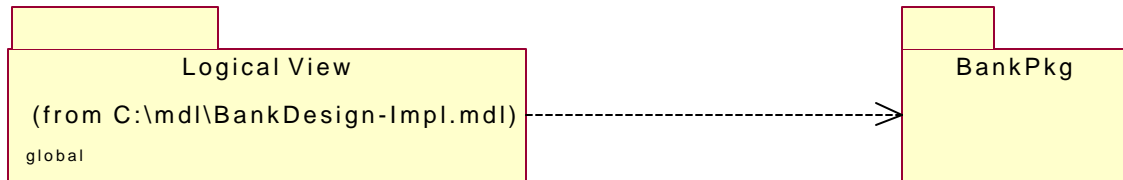
Enterprise: Distributed CORBA Based, components with public interfaces

System (Component to Component): Layered, Session - Entity, Callbacks, Publisher - Subscriber

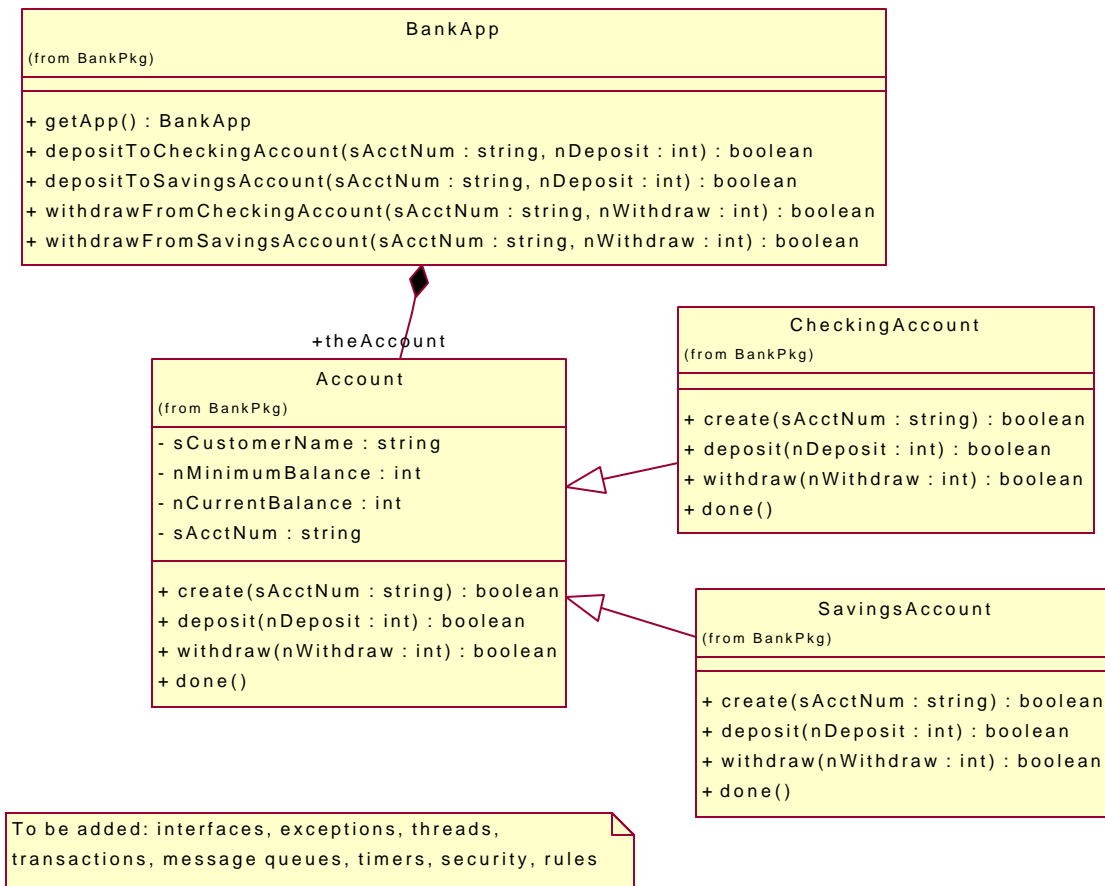
Component: Application - Document, Controller - Entity - Boundary, Facade

Class Design: UML, Factory, Transaction

11 - Design Package Diagram

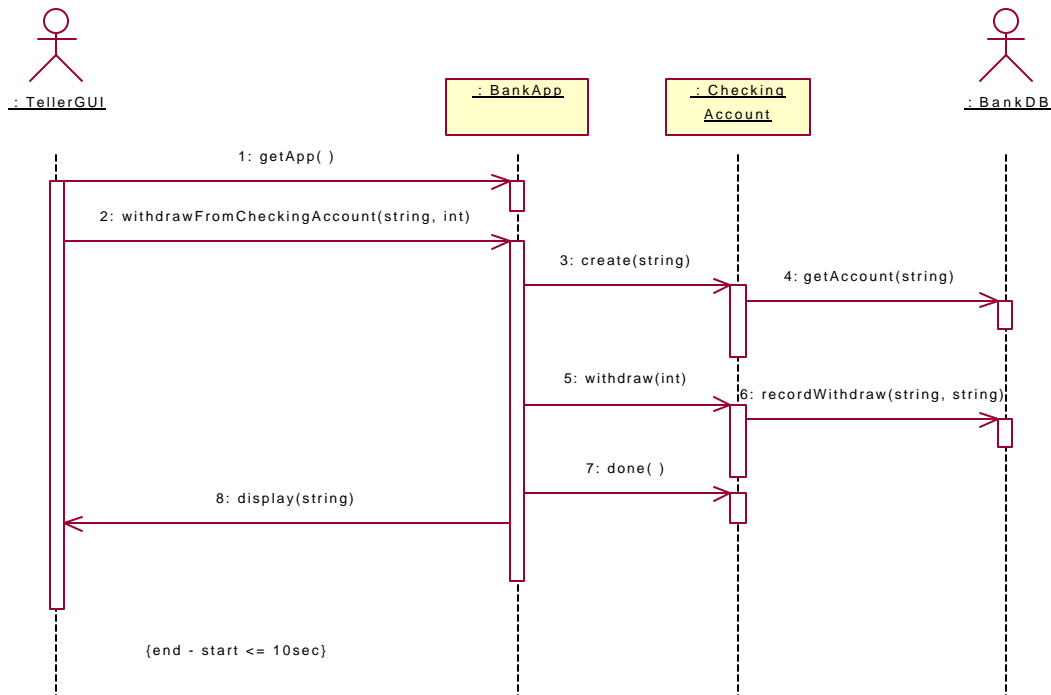


11 - Design Class Diagram Showing Types and Parameters - Goal is completeness for coding



Note: CheckingAccount and SavingsAccount will provide implementation of the polymorphic operations.

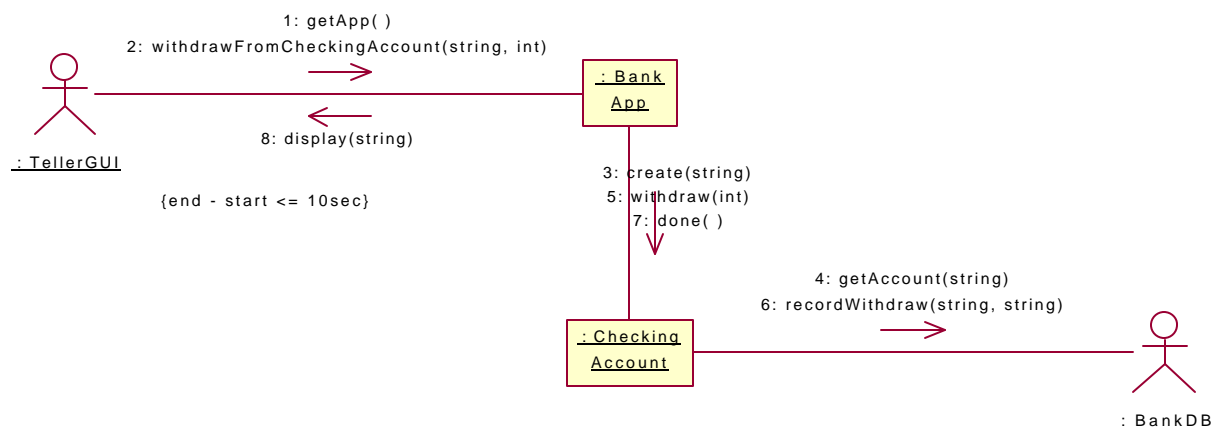
11 - Design Sequence Diagram WithdrawFromCheckingAccount - Optimistic Scenario



Note: getApp required to get the top level application for the first transaction

11 - Design Collaboration Diagram WithdrawFromCheckingAccount - Optimistic Scenario

In Rational Rose open sequence diagram and press F5 to automatically create collaboration diagram.



11 - Design Operation Specification for withdraw() in CheckingAccount Class

Use Case Name : withdraw

Trigger: withdraw

Input Parameters: nWithdraw : int

Output Return: boolean

Precondition: nWithdraw <= nCurrentBalance

Precondition Exception Raised: exInsufficientFunds

Description/Transformation: nCurrentBalance = nCurrentBalance - nWithdraw

Postcondition: nCurrentBalance < priorCurrentBalance

Postcondition Exception: exIncorrectBalance

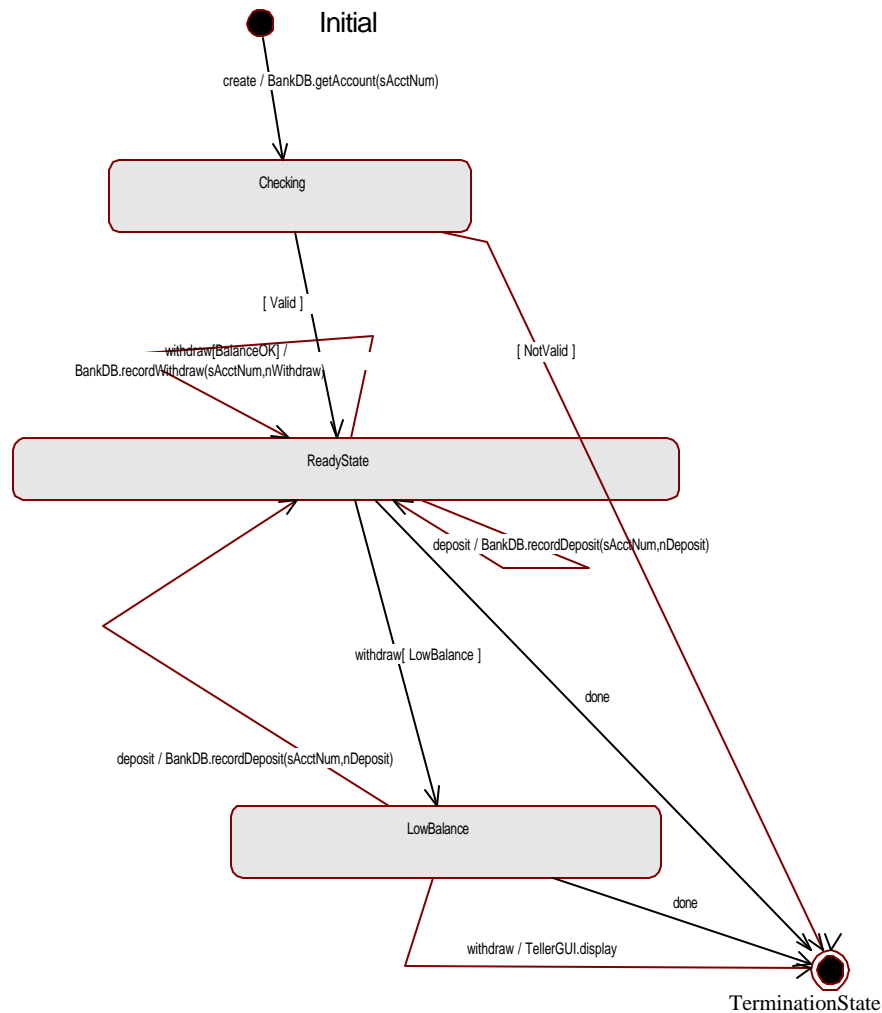
Basic Scenario/Optimistic Scenario: See withdrawFromCheckingAccount Sequence Diagram

Alternative Scenarios/Pessimistic Scenario: See withdrawFromCheckingAccount Activity Diagram

Business Rules: ValidAccountRule, AdequateBalanceRule

11 - Design Exception Classes: Exception Superclass with Exception(),Exception(string); Exception Subclasses: exInsufficientFunds with exInsufficientFunds() and exInsufficientFunds(string); exIncorrectBalance with exIncorrectBalance() and exIncorrectBalance(string).

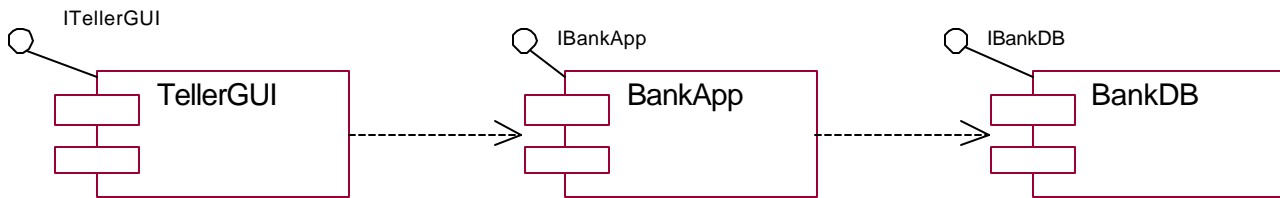
11 - Design State Diagram for Checking Account Class



IV - Implementation Models

12 - Design Processing Environment: UNIX, C++, CORBA

13 - Implementation Component Diagram



Implementation Files: TellerGUI.exe, BankApp.exe, BankDB.exe

Component Interface Alternatives: 1) BankApp has single interface IBankApp with all operations exposed

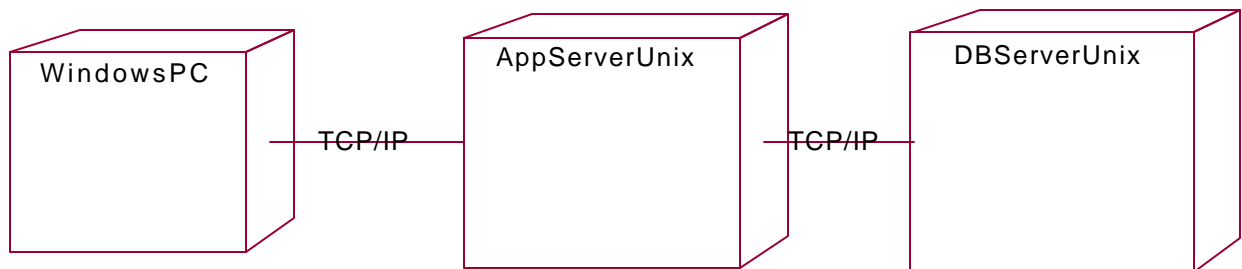
2) BankApp exposes IBankApp, ICheckingAccount, & ISavingAccount Interfaces

3) BankApp exposes IBankApp, IWithdraw, IDeposit, ICheckingAccount, & ISavingAccount Interfaces

CORBA IDL/C++ needed to describe interfaces

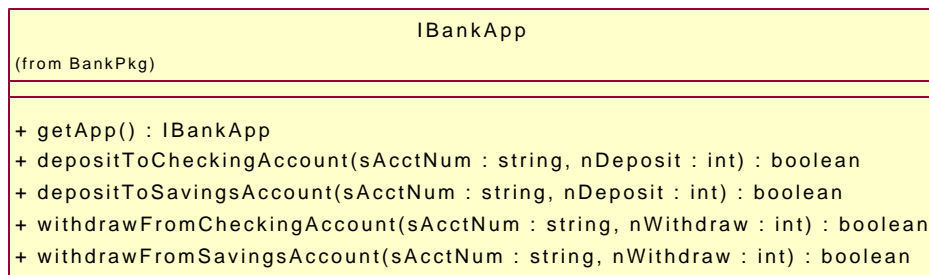
Rose Component Diagram: In Browser Window select Component View; Rename Main to be Component Diagram; Double-click the diagram name to display the diagram; Place component and dependency relationships (drag from the client component to the supplier component) on the diagram; In the Browser drag each class to the appropriate component; Select Tools - Check Model; Select File - Save.

14 - Implementation Deployment Diagram



Rose Deployment Diagram: In Browser Window select Deployment View; Double-click to display the diagram; Place nodes and connection relationships on the diagram; Select Tools - Check Model; Select File - Save.

//Interface for BankApp System Using CORBA



//Sample IDL Interface Code

```

module BankApp {
interface IBankApp {
    exception exInsufficientFunds;
    IBankApp getApp();
    boolean depositToCheckingAccount (in string sAcctNum, in int nDeposit) ;
    boolean depositToSavingAccount (in string sAcctNum, in int nDeposit) ;
    boolean withdrawFromCheckingAccount (in string sAcctNum, in int nWithdraw)raises (exInsufficientFunds);
    boolean withdrawFromSavingAccount (in string sAcctNum, in int nWithdraw) raises (exInsufficientFunds);
};};

//Sample Java Interface Code Using Remote Method Invocation
import java.rmi.*;
package BankApp;
public interface IBankApp extends java.rmi.Remote {
    boolean depositToCheckingAccount (String sAcctNum, int nDeposit) throws java.rmi.RemoteException ;
    boolean depositToSavingAccount (String sAcctNum, int nDeposit) throws java.rmi.RemoteException ;
    boolean withdrawFromCheckingAccount (String sAcctNum, int nWithdraw) throws java.rmi.RemoteException;
    boolean withdrawFromSavingAccount (String sAcctNum, int nWithdraw) throws java.rmi.RemoteException;
}

//Sample Microsoft IDL Interface Code for COM - Simplified
library BankAppLib {
dispinterface IBankApp {
    IBankApp getApp();
    boolean depositToCheckingAccount (BSTR sAcctNum, int nDeposit) ;
    boolean depositToSavingAccount (BSTR sAcctNum, int nDeposit) ;
    boolean withdrawFromCheckingAccount (BSTR sAcctNum, int nWithdraw);
    boolean withdrawFromSavingAccount (BSTR sAcctNum, int nWithdraw);
};
coclass BankApp {
    dispinterface IBankApp;
}; };

//Sample SOAP (Simple Object Access Protocol) SDL (Service Description Language) with XML - Incomplete
<?xml version='1.0'?>
<serviceDescription name='BankApp'
    xmlns='urn:schemas-xmlsoap-org:sdl:2000-01-25'
    xmlns:dt='http://www.w3.org/1999/XMLSchema'
    xmlns:IBankApp='IBankApp'>
<import namespace='IBankApp' location='#IBankApp'/>
<soap xmlns='urn:schemas-xmlsoap-org:soap-sdl-2000-01-25'>
    <interface name='IBankApp'>
        <requestResponse name='WithdrawFromCheckingAccount'>
            <request ref='IBankApp:WithdrawFromCheckingAccount' />
            <response ref='IBankApp:WithdrawFromCheckingAccountResponse' />
        </requestResponse>
    </interface>
</service>
<addresses>
    <address uri='http://myserver/IBankApp.asp' />
</addresses>
<implements name='IBankApp' />
</service>
</soap>
<IBankApp:schema id='IBankApp' targetNamespace='IBankApp' xmlns='http://www.w3.org/1999/XMLSchema'>
    <element name='WithdrawFromCheckingAccount'>
    </element>
    <element name='WithdrawFromCheckingAccountResponse'>
        <type>
            <element name='return' type='dt:boolean' />
        </type>
    </element>
</IBankApp:schema>
</serviceDescription>

```

V - Construction

Coding/Naming Standards; Interface Code - CORBA IDL; C++ Code; CASE Tool Scripts/VBA for custom reports/code generation

Rose Code Generation - Requires Rose Professional or Enterprise Version. See Help Topic Code Generation. Basic Steps: 1 - Check Model 2 - Create Components 3 - Map/assign classes to components 4 - Set Code Generation Properties 5 - Select a class, component, or package 6 - Generate code 7 - Examine generated code.

Rose Reverse Engineering - Requires Rose Professional or Enterprise Version. See Help Topic Reverse Engineering. Use the Model Update Tool.

VI - Testing

Total System/Integration Testing - All Components/Subsystems; Component Testing - Each Component
Unit Testing - Each Class. **Test Case Specification:**

Test Use Case Name:

Use Case Name:

Use Case Scenario Name:

Trigger:

Input Parameters:

Output Return:

Precondition:

Precondition Exception Raised:

Description/Transformation:

Postcondition:

Postcondition Exception:

Comments:

VII - Model and Construct Other Components

TellerGUI Forms; BankDB Tables: tblWithdraw, tblDeposit, tblSavingAccount, tblCheckingAccount

UML Stereotypes, Tagged-values, and Constraints

O-O Element	<<Sample Stereotype>>	{Sample Tagged value - property}	{Sample Constraint}
Actor	Human user, machine, interacting software system, device		
Use Case	Use Cases: abstract, concrete, extension, included, parent, child; use case relationships : communicates, includes, extends, specializes		
Package	Façade, framework, stub, subsystem, system, boundary, controller, entity, process, category, processor group, service group, use case group package relationships : access, import	namespace, package	
Class	Metaclass, powertype, stereotype, utility, process, thread, implementationClass, type, interface, class, datatype, boundary, controller, entity, exception, signal, template, enumeration, transaction	abstract, interface, parameterized, final, concrete, leaf, root	
Operation	Constructor, query, update, destructor	abstract, class - static, final - const, synchronized, native, inline, friend, isQuery, sequential, guarded, concurrent, isPolymorphic (may be overridden)	
Parameter	In - may not be modified, out – may be modified to communication information to caller, inout – may be modified, return		
Attribute	Read only, write only, read write	changeable, addOnly, frozen-final – const, class - static, derived	
Relationship	Generalization : implementation, subclass, subtype, implements interface/realizes; extends (inherits); Association : association, composition aggregation, shared aggregation; Dependency between classes/objects : bind, derive, friend, instanceOf, instantiate, powertype, refine, uses;	final – const, friend, mutable, not mutable, navigable, not navigable, ordered, not ordered	Generalization : complete, incomplete, overlapping, disjoint; Association : implicit, or, changeable, addOnly, frozen
State	Wait state, action state, activity state, sub-state, initial state, final state, history state, decision, fork, join	enumerated type, class	

Event	Call event, signal event, change event, time event		
Action	Call, return, send, create, destroy		
Object	Interface, boundary, controller, entity, exception, signal event, utility, thread	transient, persistent ; UML Link End: association, global, local, parameter, self; Other: automatic, dynamic, static	new, destroyed, transient, persistent
Message	Call, synchronous, asynchronous, balking, timeout, periodic; Interaction between objects : become, call, copy	UML Request: broadcast, vote	
Component	Executable, document, file, library, table, dll, CORBA/Java Component		
Node	Processor, device, memory, network; Link between nodes: TCP-IP, RS-232, 10-T Ethernet, USB		
Constraint	Invariant, metaclass, precondition, postcondition, powertype		

Sample tagged values for all elements: **documentation, location, semantics**

O-O Goodness Guidelines for All Modeling Elements

Guidelines may be found in Grady Booch's *Object Solutions - Managing the Object-Oriented Project*. C++ coding guidelines may be found in Scott Meyer's *Effective C++ - 50 Ways to Improve Your Programs and Designs* and in Arthur Riel's *Object-Oriented Design Heuristics*.

- Simplest possible - Clear meaningful name
- Complete Specification including stereotype, property, and constraints
- Consistent name and semantics between diagrams
- Supports weak coupling between elements and strong cohesion within an element
- Distribute processing (intelligence) rather than centralize processing (intelligence)
- Supports use of patterns and reusable elements

O-O Element	O-O Goodness Guideline
System	Has a well-defined layered architecture Use reusable patterns (architecture, design, idioms)
Actor	Represents a role
Use Case	Represents a use/function of the system; May have optimistic, normal, and pessimistic scenarios; All concrete use cases identified; Each use case has an activity diagram showing all paths; Largely independent use case increments identified;
Package	Is the primary element in large systems; Classes in the package are highly cohesive
Class	Provides a single abstraction of something in the problem or solution domain Has a well-defined set of 3 - 5 responsibilities Is simple, understandable, extensible, and adaptable Exposes minimum functionality Is dependent upon as few other classes as possible (weak coupling) Attributes and operation are cohesive Has operations for object creation, copy, assignment, equality check, etc
Attribute	Has private or protected visibility Cohesive – supports the basic purpose of the class Is initialized. Has accessor operations if required
Operation	Has appropriate visibility - private, protected, public Implemented with a small number of lines of code Has few number of parameters If complex has preconditions/thrown exceptions and postconditions/thrown exceptions Subclass preconditions should be equal to or weaker than superclass preconditions Subclass postconditions should be equal to or stronger than superclass postconditions Cohesive – supports the basic purpose of the class Operation may be sequential or concurrent (thread)
Generalization	Superclass/subclasses have interface (behavioral) inheritance with polymorphic operations Polymorphic operations have identical signatures (simplest) or conforming signatures (more complex) Superclass/subclass levels should not exceed 5 - 6 levels Superclasses should be abstract - No recursive generalization

Realization	Implementing class implements all operations specified in the interface; Prefer interfaces to multiple inheritance; Prefer interfaces to multiple inheritance
Association & Aggregation-Composition Relationships	Has private or protected visibility; Has a role name to be used in coding Minimum number of relationships for weak coupling Minimum inverse - 2 way relationships for weak coupling Class with association has public accessor operations to get/set/modify associated objects - Class with association does not create, copy, or destroy associated objects Aggregate (whole) class has no public accessor operations to get/set/modify part objects Aggregate (whole) class creates, copies, and destroys part objects Aggregation-composition may have an inverse association but not an inverse aggregation-composition Favor aggregation-composition over inheritance
State	State has a class; Part States with the same transitions into a composite state Initial and Final States are shown
Transition	Each event has an operation in a class or there is a processEvent(Event) operation; Transitions show all possible combinations of events, conditions, and actions including all paths in an activity diagram.
Object	Object is an instance of a class; Object is sequential or concurrent (active object) with wait semantics
Message	Message invokes an operation defined a class Message may be sequential call or concurrent (synchronous, asynchronous, balking, timeout)
Component	Exports one or more interfaces (set of operations)
Node	Represents a physical processor, device, or other hardware; Provides a crisp abstraction of something drawn from the vocabulary of the hardware; Directly deploys a set of components that reside on the node; Exposes the minimum attributes and operations that are relevant; Is connected to other nodes that reflects the topology of the system
All Elements	Simplest possible - Clear meaningful name Complete Specification including Stereotype, tagged value - property, and constraints Consistent name and semantics between diagrams Supports weak coupling between elements and strong cohesion within an element Distribute processing (intelligence) rather centralize processing (intelligence) Supports use of patterns and reusable elements

Requirements Model Checklist

Category	Check	Comment
Project Plan	Documents the development project in terms of cost/schedule/performance, Major risks/workarounds, QA factors (reliability, correctness, extensibility, etc), Reuse plan (patterns, components, classes, operations/utilities), Documentation plan (user manual, help system, tutorials), Staffing for overall project (project manager, architect, client/user, methodologist/toolsmith, Business/System Analyst, developer/programmer, tester, reusable component/class librarian, technical documentor), Staff for 10 - 12 member development teams, O-O roadmap (diagrams, specifications, code), tools (requirements tracing, CASE, compiler, code analyzers, testing), policies (standard library, threads, exceptions, etc), training/help desk, sample project documentation provided	
Requirements Statement	Sufficient to identify system use cases, system operations and the system boundary in terms of use cases, system in messages, system out messages, system input objects/data, and system output objects/data	
UML Diagram and Specification Checks	Use case diagram shows use cases for major system operations in Requirements Statement; System sequence diagram exists for each use case scenario for optimistic, normal, pessimistic, and other circumstances System collaboration diagram shows the system, actors, system in messages, system out messages, system input objects/data, and system output objects/data Use case specification show preconditions/thrown exceptions, transformation, and postconditions	

	Activity diagram for each use case shows all scenarios/paths for the use case	
UML Element Checks	System - Has a well-defined layered architecture; Use reusable patterns (architecture, design, idioms) Actor - Represents a role Use Case - Represents a use/function of the system; May have optimistic, normal, and pessimistic scenarios; All concrete use cases identified; Each use case has an activity diagram showing all paths; Largely independent use case increments identified;	
CASE tool check	Shows no major diagram/specification inconsistencies	
Walkthrough (role play) - Optional	Each use case scenario with a person assigned to each actor and the system	
Documentation Review	All required documents are up to date	

Analysis Model Checklist

Category	Check	Comment
UML Diagrams and Specifications	<ul style="list-style-type: none"> - Class Diagram - Each class has 2 or more attributes and 2 or more operations. Classes with a common purpose are grouped together in a package - Sequence diagram showing objects and messages for each use case scenario. Each message invokes an operation shown in a class on the class diagram. Each object is an instance of a class on the class diagrams - Statechart shows state based behavior for a class on the class diagram. Each event invokes an operation shown in a class on the class diagram. Each event is shown as a message on the sequence diagram - Operation specification for complex operations show preconditions/thrown exceptions, transformation, and postconditions 	
UML Element Checks	<p>Package - Is the primary element in large systems; Classes in the package are highly cohesive</p> <p>Class - Provides a single abstraction of something in the problem or solution domain</p> <p>Has a well-defined set of 3 - 5 responsibilities</p> <p>Is simple, understandable, extensible, and adaptable</p> <p>Exposes minimum functionality</p> <p>Is dependent upon as few other classes as possible (weak coupling)</p> <p>Attributes and operation are cohesive</p> <p>Has operations for object creation, copy, assignment, equality check, etc</p> <p>Attribute - Has private or protected visibility</p> <p>Cohesive – supports the basic purpose of the class</p> <p>Is initialized. Has accessor operations if required</p> <p>Operation - Has appropriate visibility - private, protected, public</p> <p>Implemented with a small number of lines of code</p> <p>Has few number of parameters</p> <p>If complex has preconditions/thrown exceptions and postconditions/thrown exceptions</p> <p>Subclass preconditions should be equal to or weaker than superclass preconditions</p> <p>Subclass postconditions should be equal to or stronger than superclass postconditions</p> <p>Cohesive – supports the basic purpose of the class</p> <p>Operation may be sequential or concurrent (thread)</p> <p>Generalization - Superclass/subclasses have interface (behavioral) inheritance with polymorphic operations</p> <p>Polymorphic operations have identical signatures (simplest) or conforming signatures (more complex)</p> <p>Superclass/subclass levels should not exceed 5 - 6 levels</p> <p>Superclasses should be abstract - No recursive generalization</p> <p>Realization - implementing class implements all operations specified in the interface; Prefer interfaces to multiple inheritance</p> <p>Association & Aggregation-Composition Relationships</p> <p>Has private or protected visibility; Has a role name to be used in coding</p> <p>Minimum number of relationships for weak coupling</p> <p>Minimum inverse - 2 way relationships for weak coupling</p>	

	<p>Class with association has public accessor operations to get/set/modify associated objects - Class with association does not create, copy, or destroy associated objects</p> <p>Aggregate (whole) class has no public accessor operations to get/set/modify part objects</p> <p>Aggregate (whole) class creates, copies, and destroys part objects</p> <p>Aggregation-composition may have an inverse association but not an inverse aggregation-composition</p> <p>Favor aggregation-composition over inheritance</p> <p>State - State has a class; Part States with the same transitions into a composite state</p> <p>Initial and Final States are shown</p> <p>Transition - Each event has an operation in a class or there is a processEvent(Event) operation; Transitions show all possible combinations of events, conditions, and actions including all paths in an activity diagram.</p> <p>Object - Object is an instance of a class; Object is sequential or concurrent (active object) with wait semantics</p> <p>Message - Message invokes an operation defined a class</p> <p>Message may be sequential call or concurrent (synchronous, asynchronous, balking, timeout)</p>	
Walkthrough (role play) - optional	Check each use case scenario with a person assigned to each object	
CASE tool check	Check shows no major diagram/specification inconsistencies	
Documentation Review	All required documents are up to date	

Design Model Checklist

Category	Check	Comment
UML Diagrams and Specifications	<p>All class/object level analysis models (diagrams and specifications) are updated for the H/W and S/W Configuration List and are sufficiently detailed to generate code or manually create code</p> <p>- See Analysis Model Checks</p>	
UML Elements	See Analysis Model Checks	
Walkthrough (role play) - optional	Check each use case scenario with a person assigned to each object	
CASE tool check	Check shows no major diagram/specification inconsistencies	
Documentation Review	All required documents are up to date	

Implementation Model Checklist - Includes Code

Category	Check	Comment
UML Diagrams and Specifications	<p>H/W and S/W Configuration List sufficiently shows the required components to implement the classes, objects, and other elements in the problem domain, graphic user interfaces/external interfaces, persistence, and distribution</p> <p>- Component Diagram shows all system executable and other executable components that a user requires</p> <p>- Deployment Diagram shows the physical elements that has the system and other executable components</p> <p>- All system level analysis models (diagrams and specifications) are updated for the H/W and S/W Configuration List</p>	
UML Elements	<p>Component - Exports one or more interfaces (set of operations)</p> <p>Node - Represents a physical processor, device, or other hardware; Provides a crisp abstraction of something drawn from the vocabulary of the hardware; Directly deploys a set of components that reside on the node; Exposes the minimum attributes and operations</p>	

	that are relevant; Is connected to other nodes that reflects the topology of the system	
CASE tool check	Shows no major diagram/specification inconsistencies	
CASE tool scripts	Written to generate code for the coding standard	
Generated code	Compiles without errors or major warnings	
Reverse Engineering Diagrams	Class diagrams accurately reflect the source code	
Code Inspection/Code Analyzer	Check that production execution code implements the required use case scenarios and meets Coding Standard guidelines	
Build/Release Code Inspection	Tests system operations with reconditions/transformations/postconditions/thrown exceptions, system in messages, system out messages, system input objects/data, system states, etc for all use cases.	
Documentation Review	All required documents are up to date	

Test Model Checklist

Category	Check	Comment
Test Plan	Test Plan up to date	
UML Diagrams and Specifications	N/A	
UML Elements	N/A	
CASE tool check	N/A	
Test Cases	All unit tests complete All integration tests complete All system tests complete Other planned tests complete: benchmark, configuration, function, installation, integrity, load, performance, stress All acceptance tests complete	
Documentation Review	All required documents are up to date	

Code Inspection Checklist

(adapted from PSE 2000 <http://www.iam.unibe.ch/~scg/Archive/Lectures/PSE2000/WWW/>)

This checklist is aimed at reviews of the *maintainability aspect* of source code. It is geared towards Java code but can probably be adapted easily for other languages.

This checklist is to be used in conjunction with **reverse engineered class diagrams**. Reverse engineering is the process to use a CASE tool to read source code and to create the class diagram or other diagram from the source code.

The following quote details how the checklist is to be used:

To do the inspection, go through the code line by line, attempting to fully understand what you are reading. At each line or block of code, skim through the inspection checklist, looking for questions which apply. For each applicable question, find whether the answer is "yes." A yes answer means a probable defect. Write it down. You will notice that some of the questions are very low-level and concern themselves with syntactical details, while others are high-level and require an understanding of what a block of code does. Be prepared to change your mental focus. See <http://www.ics.hawaii.edu/~johnson/FTR/Bib/Baldwin92.html>.

The Code Inspection Protocol can be used to record the defects that are found.

The defect types listed here have been assigned a severity level. The meaning of this levels are the following:

Level	Meaning
Severe (S)	This type of defect strongly hinders maintenance and evolution of the software system by introducing inflexible structures.

Dangerous (D)	This type of defect heightens the likeliness that maintenance programmers unwillingly introduce errors when changing the system.
Impedimental (I)	This type of defect makes the code harder to read and understand.

As *cosmetic defects* we consider things that can be improved automatically, e.g. by a pretty printer. These defects are thus not taken into account and also not in the checklist.

The Checklist

The checklist is grouped into the following sections:

- Comments
- Names
- Variable Names
- Method Names
- Aliases
- Coding
- Design
- Object-oriented Design
- Code Layout
- Code Duplication

Defect Type	Defect Detecting Question	Examples	Severity	Comment
Comments				
1.	Do the comments fail to accurately explain what the code does?		I	
2.	Are the comments superfluous?	The classic bad guy: <code>i++; /* add 1 to i */</code>	I	
3.	Are variables (global, local and instance variables) uncommented? Example facts to comment on: Usage Units of measure Bounds, Legal values Implied/displayed number of decimal points Display format Data entry rules (e.g. must enter)		I	
4.	Does a method definition comment fail to document which of its parameters the method is going to change?		D	
Names				
5.	Are acronyms used instead of spelling names out?	<code>cntBkr</code> instead of <code>centralBanker</code>	I	
6.	Are different spellings used for the same word?	<code>colors</code> , <code>colours</code> , and <code>kulerz</code>	I	
7.	Are different names used for same-valued variables or methods of the same functionality?	<code>display</code> , <code>show</code> , <code>present</code> used for the same action	I	
8.	Are variable names used that have a typographically similar spelling?	Easily misinterpreted names: <code>ilIl1</code> and <code>o008</code> Easily confused: <code>parseInt</code> and <code>parseInt</code>	D	
9.	Are names built using abstract, cloudy words?	Examples of unclear vocabulary: <code>it</code> , <code>everything</code> , <code>data</code> , <code>handle</code> , <code>stuff</code> <code>do</code> , <code>routine</code> , <code>perform</code> <code>PerformDataFunction</code> , <code>DoIt</code> , <code>HandleStuff</code>	I	

10.	Are naming conventions ignored: Do class names start with lower case letters? Do variable names start with upper case letters? Do names of constants contain lower case letters?		I	
11.	Does capitalization of internal words change in variable names? (When variable names are constructed by gluing words directly together, e.g. <code>singingInTheRain</code> , all but the first words are called internal words. They are distinguished by capitalizing the initial letter)	<code>inputFileName := "foo.in"</code> <code>outputFilename := "foo.out"</code>	I	
Method Names				
12.	Does the method name fail to mention the side effects the method effectuates?	A method named <code>isValid(x)</code> as a side effect converts <code>x</code> to binary and stores the result in a database.	D	
13.	Is the class name used for methods other than constructors? (Possible in Java!)		I	
Variable Names				
14.	Do labels of fields on a GUI have different names from the variables that are displayed/entered there?	the field labeled "Postal Code" feeds the associated variable "zip"	D	
15.	Are the loop variable names <code>i</code> and <code>j</code> used for conceptually different (non-integer) values?	<code>i := 3.1415;</code> <code>j := "HelloDolly";</code>	D	
Aliases				
16.	Are constant parameters literally inserted into the code?	<code>100</code> instead of <code>MAXBUFFER</code> <code>open(" /home/user/project/100.log.txt ")</code> instead of <code>open(logFilename)</code>	D-S	
17.	Are constant names used interchangeably with the literal value?	<code>while(items <= MAXBUFFER) {</code> <code> if(items = 100) { ...</code>	D	
18.	Are dependencies between constant parameters hidden?	<code>UPPERBOUND := 100;</code> <code>LOWERBOUND := 50;</code> instead of <code>UPPERBOUND := 100;</code> <code>LOWERBOUND :=</code> <code>UPPERBOUND/2;</code>	I	
Coding				
19.	Are temporary variables used for two unrelated purposes?	<code>int i;</code> <code>for(i=0;i<n;i++) { ... }</code> ... <code>i := euclidDistance(v,w);</code>	D	
20.	Are variables defined at scopes that are wider than they could be?	Instance variables defined instead of local variables Global variables defined instead of local variables	D	
Design				
21.	For data that can be converted into different formats: Is one format chose to do all computations in and do the conversions only right after input or before output?	Domains which have frequently converted data: Currencies, temperature, length, weight	S	

Object-oriented Design				
22.	Is the encapsulation principle violated?	Instance variables are defined public Implementation revealing methods appear in the public interface of the class	S	
23.	Is polymorphism simulated?	Type tests in conjunction with case statements: <pre>switch(p.phoneType()) { case POTSPhone: ... break; case ISDNPhone: ... break; default: ... }</pre>	S	
24.	Is instance data stored in class (static) variables?	This could be the case when the author implicitly assumed that the class will only have one instance at runtime.	S	
25.	Are multiple conceptually identical methods used where an enumerated constants as parameter would suffice to have only one method?	Having there methods <pre>setLeftAlignment setRightAlignment setCenterAlignment</pre> instead of writing only <pre>setAlignment(int alignment)</pre> where alignment can have the values left, right, center	S	
Code Layout				
26.	Is more than one statement written per line?	This practice can save temporary variables but makes the code harder to read	I	
27.	Are if/else blocks missing the enclosing { } if it is not syntactically necessary?	This can lead to deceptive layouts: <pre>if (a) if (b) x = y; else x = z;</pre>	D	
28.	Is the nesting level of () exceeding 5? (The number 5 is arbitrary, readability can already be lost at lower levels)		D	
29.	Is the nesting level of { } blocks exceeding 7? (The number 7 is arbitrary, readability can already be lost at lower levels)		D	
30.	Are methods longer than 200 lines? (The number 200 is arbitrary, the overview can already be lost at lower line counts)	<u>FAMOOS</u> anecdotal evidence: while investigating a real industrial software system, a method was found which was 5000 lines long and was named <code>createButton()</code> .	D	
Code Duplication				
31.	Is code duplicated?	parts of methods entire methods parts of classes (missing polymorphism)	S	

This checklist is partly based on How To Write Unmaintainable Code from <http://mindprod.com/>.

UML Glossary

This glossary defines the terms that are used to describe the Unified Modeling Language (UML) and the Meta Object Facility (MOF). In addition to UML and MOF specific terminology, it includes related terms from OMG standards and object-oriented analysis and design methods, as well as the domain of object repositories and meta data managers. Glossary entries are organized alphabetically and MOF specific entries are identified as '[MOF]'.

Notation Conventions

The entries in the glossary usually begin with a lowercase letter. An initial uppercase letter is used when a word is usually capitalized in standard practice. Acronyms are all capitalized, unless they traditionally appear in all lowercase. When one or more words in a multi-word term is enclosed in brackets, it indicates that those words are optional when referring to the term. For example, *use case [class]* may be referred to as simply *use case*.

The following conventions are used in this glossary:

- Contrast: <term> Refers to a term that has an opposed or substantively different meaning.
- See: <term> Refers to a related term that has a similar, but not synonymous meaning.
- Synonym: <term> Indicates that the term has the same meaning as another term, which is referenced.
- Acronym: <term> Indicates that the term is an acronym. The reader is usually referred to the spelled-out term for the definition, unless the spelled-out term is rarely used.

abstract class A class that cannot be directly instantiated. Contrast: *concrete class*.

abstraction The essential characteristics of an entity that distinguish it from all other kinds of entities. An abstraction defines a boundary relative to the perspective of the viewer.

action The specification of an executable statement that forms an abstraction of a computational procedure. An action typically results in a change in the state of the system, and can be realized by sending a message to an object or modifying a link or a value of an attribute.

action sequence An expression that resolves to a sequence of actions.

action state A state that represents the execution of an atomic action, typically the invocation of an operation.

activation The execution of an action.

active class A class whose instances are active objects. See: *active object*.

active object An object that owns a thread and can initiate control activity. An instance of active class. See: *active class, thread*.

activity graph A special case of a state machine that is used to model processes involving one or more classifiers. Contrast: *statechart diagram*.

actor [class] A coherent set of roles that users of use cases play when interacting with these use cases. An actor has one role for each use case with which it communicates.

actual parameter Synonym: *argument*.

aggregate [class] A class that represents the “whole” in an aggregation (whole-part) relationship. See: *aggregation*.

aggregation A special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part. See: *composition*.

analysis The part of the software development process whose primary purpose is to formulate a model of the problem domain. Analysis focuses what to do, design focuses on how to do it. Contrast: *design*.

analysis time Refers to something that occurs during an analysis phase of the software development process. See: *design time, modeling time*.

architecture The organizational structure and associated behavior of a system. An architecture can be recursively decomposed into parts that interact through interfaces, relationships that connect parts, and constraints for assembling parts. Parts that interact through interfaces include classes, components and subsystems.

argument A binding for a parameter that resolves to a run-time instance. Synonym: *actual parameter*. Contrast: *parameter*.

artifact A piece of information that is used or produced by a software development process. An artifact can be a model, a description, or software. Synonym: *product*.

association The semantic relationship between two or more classifiers that specifies connections among their instances.

association class A model element that has both association and class properties. An association class can be seen as an association that also has class properties, or as a class that also has association properties.

association end The endpoint of an association, which connects the association to a classifier.

attribute A feature within a classifier that describes a range of values that instances of the classifier may hold.

behavior The observable effects of an operation or event, including its results.

behavioral feature A dynamic feature of a model element, such as an operation or method.

behavioral model aspect A model aspect that emphasizes the behavior of the instances in a system, including their methods, collaborations, and state histories.

binary association An association between two classes. A special case of an n-ary association.

binding The creation of a model element from a template by supplying arguments for the parameters of the template.

boolean An enumeration whose values are true and false.

boolean expression An expression that evaluates to a boolean value.

cardinality The number of elements in a set. Contrast: *multiplicity*.

child In a generalization relationship, the specialization of another element, the parent. See: *subclass*, *subtype*. Contrast: *parent*.

call An action state that invokes an operation on a classifier.

class A description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment. See: *interface*.

classifier A mechanism that describes behavioral and structural features. Classifiers include interfaces, classes, datatypes, and components.

classification The assignment of an object to a classifier. See *dynamic classification*, *multiple classification* and *static classification*.

class diagram A diagram that shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships.

client A classifier that requests a service from another classifier. Contrast: *supplier*.

collaboration The specification of how an operation or classifier, such as a use case, is realized by a set of classifiers and associations playing specific roles used in a specific way. The collaboration defines an interaction. See: *interaction*.

collaboration diagram A diagram that shows interactions organized around the structure of a model, using either classifiers and associations or instances and links. Unlike a sequence diagram, a collaboration diagram shows the relationships among the instances. Sequence diagrams and collaboration diagrams express similar information, but show it in different ways. See: *sequence diagram*.

comment An annotation attached to an element or a collection of elements. A note has no semantics. Contrast: *constraint*.

compile time Refers to something that occurs during the compilation of a software module. See: *modeling time*, *run time*.

component A physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. A component represents a physical piece of implementation of a system, including software code (source, binary or executable) or equivalents such as scripts or command files.

component diagram A diagram that shows the organizations and dependencies among components.

composite [class] A class that is related to one or more classes by a composition relationship. See: *composition*.

composite aggregation Synonym: *composition*.

composite state A state that consists of either concurrent (orthogonal) substates or sequential (disjoint) substates. See: *substate*.

composition A form of aggregation association with strong ownership and coincident lifetime as part of the whole. Parts with non-fixed multiplicity may be created after the composite itself, but once created they live and die with it (i.e., they share lifetimes). Such parts can also be explicitly removed before the death of the composite. Composition may be recursive. Synonym: *composite aggregation*.

concrete class A class that can be directly instantiated. Contrast: *abstract class*.

concurrency The occurrence of two or more activities during the same time interval. Concurrency can be achieved by interleaving or simultaneously executing two or more threads. See: *thread*.

concurrent substate A substate that can be held simultaneously with other substates contained in the same composite state. See: *composite state*. Contrast: *disjoint substate*.

constraint A semantic condition or restriction. Certain constraints are predefined in the UML, others may be user defined. Constraints are one of three extensibility mechanisms in UML. See: *tagged value*, *stereotype*.

container 1. An instance that exists to contain other instances, and that provides operations to access or iterate over its contents. (for example, arrays, lists, sets). 2. A component that exists to contain other components.

containment hierarchy A namespace hierarchy consisting of model elements, and the containment relationships that exist between them. A containment hierarchy forms a graph.

context A view of a set of related modeling elements for a particular purpose, such as specifying an operation.

datatype A descriptor of a set of values that lack identity and whose operations do not have side effects. Datatypes include primitive pre-defined types and user-definable types. Pre-defined types include numbers, string and time. User-definable types include enumerations.

defining model [MOF] The model on which a repository is based. Any number of repositories can have the same defining model.

delegation The ability of an object to issue a message to another object in response to a message. Delegation can be used as an alternative to inheritance. Contrast: *inheritance*.

dependency A relationship between two modeling elements, in which a change to one modeling element (the independent element) will affect the other modeling element (the dependent element).

deployment diagram A diagram that shows the configuration of run-time processing nodes and the components, processes, and objects that live on them. Components represent run-time manifestations of code units. See: *component diagrams*.

derived element A model element that can be computed from another element, but that is shown for clarity or that is included for design purposes even though it adds no semantic information.

design The part of the software development process whose primary purpose is to decide how the system will be implemented. During design strategic and tactical decisions are made to meet the required functional and quality requirements of a system.

design time Refers to something that occurs during a design phase of the software development process. See: *modeling time*. Contrast: *analysis time*.

development process A set of partially ordered steps performed for a given purpose during software development, such as constructing models or implementing models.

diagram A graphical presentation of a collection of model elements, most often rendered as a connected graph of arcs (relationships) and vertices (other model elements). UML supports the following diagrams: class diagram, object diagram, use case diagram, sequence diagram, collaboration diagram, state diagram, activity diagram, component diagram, and deployment diagram.

disjoint substate A substate that cannot be held simultaneously with other substates contained in the same composite state. See: *composite state*. Contrast: *concurrent substate*.

distribution unit A set of objects or components that are allocated to a process or a processor as a group. A distribution unit can be represented by a run-time composite or an aggregate.

domain An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area.

dynamic classification A semantic variation of generalization in which an object may change its classifier. Contrast: *static classification*.

element An atomic constituent of a model.

entry action An action executed upon entering a state in a state machine regardless of the transition taken to reach that state.

enumeration A list of named values used as the range of a particular attribute type. For example, `RGBColor = { red, green, blue }`. Boolean is a predefined enumeration with values from the set { false, true }.

event The specification of a significant occurrence that has a location in time and space. In the context of state diagrams, an event is an occurrence that can trigger a transition.

exit action An action executed upon exiting a state in a state machine regardless of the transition taken to exit that state.

export In the context of packages, to make an element visible outside its enclosing namespace. See: *visibility*. Contrast: *export* [OMA], *import*.

expression A string that evaluates to a value of a particular type. For example, the expression “(7 + 5 * 3)” evaluates to a value of type number.

extend A relationship from an extension use case to a base use case, specifying how the behavior defined for the extension use case augments (subject to conditions specified in the extension) the behavior defined for the base use case. The behavior is inserted at the location defined by the extension point in the base use case. The base use case does not depend on performing the behavior of the extension use case. See *extension point*, *include*.

facade A stereotyped package containing only references to model elements owned by another package. It is used to provide a ‘public view’ of some of the contents of a package.

feature A property, like operation or attribute, which is encapsulated within a classifier, such as an interface, a class, or a datatype.

final state A special kind of state signifying that the enclosing composite state or the entire state machine is completed.

fire To execute a state transition. See: *transition*.

focus of control A symbol on a sequence diagram that shows the period of time during which an object is performing an action, either directly or through a subordinate procedure.

formal parameter Synonym: *parameter*.

framework 1. A stereotyped package consisting mainly of patterns. See: *pattern*. 2. An architectural pattern that provides an extensible template for applications within a specific domain.

generalizable element A model element that may participate in a generalization relationship. See: *generalization*.

generalization A taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional information. An instance of the more specific element may be used where the more general element is allowed. See: *inheritance*.

guard condition A condition that must be satisfied in order to enable an associated transition to fire.

implementation A definition of how something is constructed or computed. For example, a class is an implementation of a type, a method is an implementation of an operation.

Implementation inheritance The inheritance of the implementation of a more specific element. Includes inheritance of the interface. Contrast: *interface inheritance*.

import In the context of packages, a dependency that shows the packages whose classes may be referenced within a given package (including packages recursively embedded within it). Contrast: *export*.

include A relationship from a base use case to an inclusion use case, specifying how the behavior for the base use case contains the behavior of the inclusion use case. The behavior is included at the location which is defined in the base use case. The base use case depends on performing the behavior of the inclusion use case, but not on its structure (i.e., attributes or operations). See *extend*.

inheritance The mechanism by which more specific elements incorporate structure and behavior of more general elements related by behavior. See *generalization*.

instance An entity to which a set of operations can be applied and which has a state that stores the effects of the operations. See: *object*.

interaction A specification of how stimuli are sent between instances to perform a specific task. The interaction is defined in the context of a collaboration. See *collaboration*.

interaction diagram A generic term that applies to several types of diagrams that emphasize object interactions. These include collaboration diagrams and sequence diagrams.

interface A named set of operations that characterize the behavior of an element.

interface inheritance The inheritance of the interface of a more specific element. Does not include inheritance of the implementation. Contrast: *implementation inheritance*.

internal transition A transition signifying a response to an event without changing the state of an object.

layer The organization of classifiers or packages at the same level of abstraction. A layer represents a horizontal slice through an architecture, whereas a partition represents a vertical slice. Contrast: *partition*.

link A semantic connection among a tuple of objects. An instance of an association. See: *association*.

link end An instance of an association end. See: *association end*.

message A specification of the conveyance of information from one instance to another, with the expectation that activity will ensue. A message may specify the raising of a signal or the call of an operation.

metaclass A class whose instances are classes. Metaclasses are typically used to construct metamodels.

meta-metamodel A model that defines the language for expressing a metamodel. The relationship between a meta-metamodel and a metamodel is analogous to the relationship between a metamodel and a model.

metamodel A model that defines the language for expressing a model.

metaobject A generic term for all metaentities in a metamodeling language. For example, metatypes, metaclasses, metaattributes, and metaassociations.

method The implementation of an operation. It specifies the algorithm or procedure associated with an operation.

Model [MOF] An abstraction of a physical system, with a certain purpose..See: *physical system*. Usage note: In the context of the MOF specification, which describes a meta-metamodel, for brevity the meta-metamodel is frequently to as simply the model.

model aspect A dimension of modeling that emphasizes particular qualities of the metamodel. For example, the structural model aspect emphasizes the structural qualities of the metamodel.

model elaboration The process of generating a repository type from a published model. Includes the generation of interfaces and implementations which allows repositories to be instantiated and populated based on, and in compliance with, the model elaborated.

model element [MOF] An element that is an abstraction drawn from the system being modeled. Contrast: *view element*. In the MOF specification model elements are considered to be metaobjects.

modeling time Refers to something that occurs during a modeling phase of the software development process. It includes analysis time and design time. Usage note: When discussing object systems, it is often important to distinguish between modeling-time and run-time concerns. See: *analysis time*, *design time*. Contrast: *run time*.

module A software unit of storage and manipulation. Modules include source code modules, binary code modules, and executable code modules. See: *component*.

multiple classification A semantic variation of generalization in which an object may belong directly to more than one classifier. See: *static classification*, *dynamic classification*.

multiple inheritance A semantic variation of generalization in which a type may have more than one supertype. Contrast: *single inheritance*.

multiplicity A specification of the range of allowable cardinalities that a set may assume. Multiplicity specifications may be given for roles within associations, parts within composites, repetitions, and other purposes. Essentially a multiplicity is a (possibly infinite) subset of the non-negative integers. Contrast: *cardinality*.

multi-valued [MOF] A model element with multiplicity defined whose Multiplicity Type:: upper attribute is set to a number greater than one. The term multi-valued does not pertain to the number of values held by an attribute, parameter, etc. at any point in time. Contrast: *single-valued*.

n-ary association An association among three or more classes. Each instance of the association is an n-tuple of values from the respective classes. Contrast: *binary association*.

name A string used to identify a model element.

namespace A part of the model in which the names may be defined and used. Within a namespace, each name has a unique meaning. See: *name*.

node A node is classifier that represents a run-time computational resource, which generally has at least a memory and often processing capability. Run-time objects and components may reside on nodes.

object An entity with a well-defined boundary and identity that encapsulates state and behavior. State is represented by attributes and relationships, behavior is represented by operations, methods, and state machines. An object is an instance of a class. See: *class*, *instance*.

object diagram A diagram that encompasses objects and their relationships at a point in time. An object diagram may be considered a special case of a class diagram or a collaboration diagram. See: *class diagram*, *collaboration diagram*.

object flow state A state in an activity graph that represents the passing of an object from the output of actions in one state to the input of actions in another state.

object lifeline A line in a sequence diagram that represents the existence of an object over a period of time. See: *sequence diagram*.

operation A service that can be requested from an object to effect behavior. An operation has a signature, which may restrict the actual parameters that are possible.

package A general purpose mechanism for organizing elements into groups. Packages may be nested within other packages.

parameter The specification of a variable that can be changed, passed, or returned. A parameter may include a name, type, and direction. Parameters are used for operations, messages, and events. Synonyms: *formal parameter*. Contrast: *argument*.

parameterized element The descriptor for a class with one or more unbound parameters. Synonym: *template*.

parent In a generalization relationship, the generalization of another element, the child. See: *subclass*, *subtype*. Contrast: *child*.

participate The connection of a model element to a relationship or to a reified relationship. For example, a class participates in an association, an actor participates in a use case.

partition 1. activity graphs: A portion of an activity graphs that organizes the responsibilities for actions. See: *swimlane*. 2. architecture: A set of related classifiers or packages at the same level of abstraction or across layers in a layered architecture. A partition represents a vertical slice through an architecture, whereas a layer represents a horizontal slice. Contrast: *layer*.

pattern A template collaboration.

persistent object An object that exists after the process or thread that created it has ceased to exist.

postcondition A constraint that must be true at the completion of an operation.

precondition A constraint that must be true when an operation is invoked.

primitive type A pre-defined basic datatype without any substructure, such as an integer or a string.

process 1. A heavyweight unit of concurrency and execution in an operating system. Contrast: *thread*, which includes heavyweight and lightweight processes. If necessary, an implementation distinction can be made using stereotypes. 2. A software development process—the steps and guidelines by which to develop a system. 3. To execute an algorithm or otherwise handle something dynamically.

projection A mapping from a set to a subset of it.

property A named value denoting a characteristic of an element. A property has semantic impact. Certain properties are predefined in the UML; others may be user defined. See: tagged value.

pseudo-state A vertex in a state machine that has the form of a state, but doesn't behave as a state. Pseudo-states include initial and history vertices.

physical system 1. The subject of a model. 2. A collection of connected physical units, which can include software, hardware and people, that are organized to accomplish a specific purpose. A physical system can be described by one or more models, possibly from different viewpoints. Contrast: *system*.

published model [MOF] A model which has been frozen, and becomes available for instantiating repositories and for the support in defining other models. A frozen model's model elements cannot be changed.

qualifier An association attribute or tuple of attributes whose values partition the set of objects related to an object across an association.

realization A relationship between classifiers, in which one classifier specifies a contract that another classifier guarantees to carry out.

receive [a message] The handling of a stimulus passed from a sender instance. See: *sender, receiver*.

receiver [object] The object handling a stimulus passed from a sender object. Contrast: *sender*.

receive signal event - a signal (asynchronous stimulus) that is handled by the receiver entity.

reception A declaration that a classifier is prepared to react to the receipt of a signal.

reference 1. A denotation of a model element. 2. A named slot within a classifier that facilitates navigation to other classifiers. Synonym: *pointer*.

refinement A relationship that represents a fuller specification of something that has already been specified at a certain level of detail. For example, a design class is a refinement of an analysis class.

relationship A semantic connection among model elements. Examples of relationships include associations and generalizations.

repository A facility for storing object models, interfaces, and implementations.

requirement A desired feature, property, or behavior of a system.

responsibility A contract or obligation of a classifier.

reuse The use of a pre-existing artifact.

role The named specific behavior of an entity participating in a particular context. A role may be static (e.g., an association end) or dynamic (e.g., a collaboration role).

run time The period of time during which a computer program executes. Contrast: *modeling time*.

scenario A specific sequence of actions that illustrates behaviors. A scenario may be used to illustrate an interaction or the execution of a use case instance. See: *interaction*.

schema [MOF] In the context of the MOF, a schema is analogous to a package which is a container of model elements. Schema corresponds to an MOF package. Contrast: *metamodel, package*.

semantic variation point A point of variation in the semantics of a metamodel. It provides an intentional degree of freedom for the interpretation of the metamodel semantics.

send [a message] The passing of a stimulus from a sender instance to a receiver instance. See: *sender, receiver*.

send signal event is a signal (asynchronous stimulus) that is created by a sender entity and sent to a receiver entity.

sender [object] The object passing a stimulus to a receiver object. Contrast: *receiver*.

sequence diagram A diagram that shows object interactions arranged in time sequence. In particular, it shows the objects participating in the interaction and the sequence of messages exchanged. Unlike a collaboration diagram, a sequence diagram includes time sequences but does not include object relationships. A sequence diagram can exist in a generic form (describes all possible scenarios) and in an instance form (describes one actual scenario). Sequence diagrams and collaboration diagrams express similar information, but show it in different ways. See: *collaboration diagram*.

signal The specification of an asynchronous stimulus communicated between instances. Signals may have parameters.

signature The name and parameters of a behavioral feature. A signature may include an optional returned parameter.

single inheritance A semantic variation of generalization in which a type may have only one supertype. Synonym: *multiple inheritance* [OMA]. Contrast: *multiple inheritance*.

single valued [MOF] A model element with multiplicity defined is single valued when its Multiplicity Type::upper attribute is set to one. The term single-valued does not pertain to the number of values held by an attribute, parameter, etc., at any point in time, since a single-valued attribute (for instance, with a multiplicity lower bound of zero) may have no value. Contrast: *multi-valued*.

specification A declarative description of what something is or does. Contrast: *implementation*.

state A condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event. Contrast: *state* [OMA].

statechart diagram A diagram that shows a state machine. See: *state machine*.

state machine A behavior that specifies the sequences of states that an object or an interaction goes through during its life in response to events, together with its responses and actions.

static classification A semantic variation of generalization in which an object may not change classifier. Contrast: *dynamic classification*.

stereotype A new type of modeling element that extends the semantics of the metamodel. Stereotypes must be based on certain existing types or classes in the metamodel. Stereotypes may extend the semantics, but not the structure of pre-existing types and classes. Certain stereotypes are predefined in the UML, others may be user defined. Stereotypes are one of three extensibility mechanisms in UML. See: *constraint, tagged value*.

stimulus The passing of information from one instance to another, such as raising a signal or invoking an operation. The receipt of a signal is normally considered an event. See: *message*.

string A sequence of text characters. The details of string representation depend on implementation, and may include character sets that support international characters and graphics.

structural feature A static feature of a model element, such as an attribute.

structural model aspect A model aspect that emphasizes the structure of the objects in a system, including their types, classes, relationships, attributes, and operations.

subactivity state A state in an activity graph that represents the execution of a non-atomic sequence of steps that has some duration.

subclass In a generalization relationship, the specialization of another class; the superclass. See: *generalization*. Contrast: *superclass*.

submachine state A state in a state machine which is equivalent to a composite state but its contents is described by another state machine.

substate A state that is part of a composite state. See: *concurrent state, disjoint state*.

subpackage A package that is contained in another package.

subsystem A grouping of model elements that represents a behavioral unit in a physical system. A subsystem offers interfaces and has operations. In addition, the model elements of a subsystem can be partitioned into specification and realization elements. See *package*. See: *physical system*.

subtype In a generalization relationship, the specialization of another type; the supertype. See: *generalization*. Contrast: *supertype*.

superclass In a generalization relationship, the generalization of another class; the subclass. See: *generalization*. Contrast: *subclass*.

supertype In a generalization relationship, the generalization of another type; the subtype. See: *generalization*. Contrast: *subtype*.

supplier A classifier that provides services that can be invoked by others. Contrast: *client*.

swimlane A partition on a activity diagram for organizing the responsibilities for actions. Swimlanes typically correspond to organizational units in a business model. See: *partition*.

synch state A vertex in a state machine used for synchronizing the concurrent regions of a state machine.

system A top-level subsystem in a model. Contrast: *physical system*.

tagged value The explicit definition of a property as a name-value pair. In a tagged value, the name is referred as the tag. Certain tags are predefined in the UML; others may be user defined. Tagged values are one of three extensibility mechanisms in UML. See: *constraint*, *stereotype*.

template Synonym: *parameterized element*.

thread [of control] A single path of execution through a program, a dynamic model, or some other representation of control flow. Also, a stereotype for the implementation of an active object as lightweight process. See *process*.

time event An event that denotes the time elapsed since the current state was entered. See: *event*.

time expression An expression that resolves to an absolute or relative value of time.

timing mark A denotation for the time at which an event or message occurs. Timing marks may be used in constraints.

top level A stereotype of package denoting the top-most package in a containment hierarchy. The *topLevel* stereotype defines the outer limit for looking up names, as namespaces “see” outwards. For example, *TopLevel* subsystem represents the top of the subsystem containment hierarchy.

trace A dependency that indicates a historical or process relationship between two elements that represent the same concept without specific rules for deriving one from the other.

transient object An object that exists only during the execution of the process or thread that created it.

transition A relationship between two states indicating that an object in the first state will perform certain specified actions and enter the second state when a specified event occurs and specified conditions are satisfied. On such a change of state, the transition is said to fire.

type A stereotype of class that is used to specify a domain of instances (objects) together with the operations applicable to the objects. A type may not contain any methods. See: *class*, *instance*. Contrast: *interface*.

type expression An expression that evaluates to a reference to one or more types.

uninterpreted A placeholder for a type or types whose implementation is not specified by the UML. Every uninterpreted value has a corresponding string representation. See: *any* [CORBA].

usage A dependency in which one element (the client) requires the presence of another element (the supplier) for its correct functioning or implementation.

use case [class] The specification of a sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system. See: *use case instances*.

use case diagram A diagram that shows the relationships among actors and use cases within a system.

use case instance The performance of a sequence of actions being specified in a use case. An instance of a use case. See: *use case class*.

use case model A model that describes a system’s functional requirements in terms of use cases.

utility A stereotype that groups global variables and procedures in the form of a class declaration. The utility attributes and operations become global variables and global procedures, respectively. A utility is not a fundamental modeling construct, but a programming convenience.

value An element of a type domain.

vertex A source or a target for a transition in a state machine. A vertex can be either a state or a pseudo-state. See: *state*, *pseudo-state*.

view A projection of a model, which is seen from a given perspective or vantage point and omits entities that are not relevant to this perspective.

view element A view element is a textual and/or graphical projection of a collection of model elements.

view projection A projection of model elements onto view elements. A view projection provides a location and a style for each view element.

visibility An enumeration whose value (public, protected, or private) denotes how the model element to which it refers may be seen outside its enclosing namespace.

UML Process Terms

UML - The Unified Modeling Language is a standard modeling language for software - a language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system.

UML Process - a software development process that is based upon the UML that is iterative, architecture-centric, use-case driven, and risk-driven. It is organized around the workflows (phases) of requirements, analysis, design, implementation, construction, testing. The process is a set of steps intended to reach a goal, e.g. to efficiently and predictably deliver a software product to meet the needs of your organization.

Increment - a set of use cases that represent a complete subset of business functionality largely independent of other increments.

Iteration - a complete pass through all phases of the software development, e.g. Requirements, Analysis, Design, Implementation, Coding for a use case increment.

Architecture - The organizational structure of a system, including its decomposition into parts, their connectivity, interaction mechanisms, and the guiding principles that inform the design of a system.

Business Model - The set of documents that describe a business or enterprise at a very high level.

Component-based development (CBD) - The creation and deployment of software-intensive systems assembled from components, as well as the development and harvesting of such components.

Middleware and Enterprise Java Bean Glossary

Application server - A server program that allows the installation of application specific software components, in a manner so that they can be remotely invoked, usually by some form of remote object method call.

Bean-managed persistence - When an Enterprise JavaBean performs its own long-term state management.

Bytecode - In the context of Java, bytecode is the platform-independent executable program code.

Clustering - Aggregating multiple servers together to form a service pool of some kind, usually for achieving redundancy or improving performance.

Component standard - A definition of how software components cooperate, and in particular the roles and interfaces of each. In the context of Java middleware, component standards usually include specifications of the middleware interfaces exposed to the components, and the component interfaces required by the middleware.

Container managed persistence - When an Enterprise JavaBean server manages a bean's long-term state.

CORBA - Standard maintained by the Object Management Group (OMG), called the Common Object Request Broker Architecture.

COS Naming - CORBA standard for object directories.

Data source - This is the term used by the JTA and JDBC specifications to refer to persistent repository of data. It usually represents a database. It also may refer to an object that makes database connections available (i.e. a driver).

DCOM - Microsoft's Distributed Component Object Model.

Enterprise JavaBeans (EJB) - A server component standard developed by Sun Microsystems.

Entity bean - An Enterprise JavaBean that maintains state across sessions, and may be looked up in an object directory by its key value.

Failover - The ability to respond resiliently to a component failure by switching to another component.

IDL - interface description language, CORBA's syntax for defining object remote interfaces.

IIOP - Internet Inter-ORB Protocol, CORBA's wire protocol for transmitting remote object method invocations.

ISAPI - Microsoft's C++ API for coding application extensions for its Internet Information Server.

Java Naming and Directory Interface - The Java standard API for accessing directory services, such as LDAP, COS Naming, and others.

Java Transaction API - Java API for coding client demarcated transactions, and for building transactional data source drivers.

JNDI - Java Naming and Directory Interface.

JTA - Java Transaction API.

JTS - The Java Transaction Service, which in the Java binding for the CORBA Transaction Service. Provides a way for middleware vendors to build interoperable transactional middleware.

JVM - Java virtual machine.

LDAP - Lightweight Directory Access Protocol, a protocol for directory services, derived from X.500.

Middleware - Software that runs on a server, and acts as either an application processing gateway or a routing bridge between remote clients and data sources or other servers, or any combination of these.

NSAPI - Netscape's C language API for adding application extensions to their Web servers.

OMG - Object Management Group, an organization that defines and promotes object oriented programming standards.

OODB - object-oriented database.

OODBMS - object-oriented database management system.

ORB - object request broker, the primary message routing component in a CORBA product.

Passivate - To place an object in a dormant state when it is not being accessed, such that it can later be returned to an active and usable state.

Persistence - Maintaining state over a long time, especially across sessions.

Pooling - Maintaining a collection of objects, servers, connections, or other resources for ready access, so that one does not need to be created anew each time one is needed.

RMI - Remote Method Invocation, the Java standard technology for building distributed objects whose methods can be invoked remotely across a network.

RMI over IIOP - Using the CORBA IIOP wire protocol from an RMI API.

Servlet - An application extension to a Java Web server.

Session bean - An Enterprise JavaBean that does not maintain its state from one session to the next. Appears to the client as if the bean was created just for that client.

Skeleton - A server-side software component that serves to relay remote calls from a client to the methods of a servant running in a server. Usually a skeleton is automatically generated by a special compiler.

SOAP - Simple Object Access Protocol for passing XML documents between distributed applications.

SQLJ - An extended Java syntax for embedding SQL-like commands in a Java program.

Stub - A client-side software component that serves to forward remote calls to a remote server, and receive the subsequent responses. Usually automatically generated by a special compiler.

Three-tier - An architecture in which a remote client accesses remote data sources via an intervening server.

Transaction manager - A software component that coordinates the separate transactions of multiple data sources, so that they behave as a single unified transaction. Requires data source drivers that can participate in this kind of coordination. Also usually provides the ability to monitor transactions and provide statistics.

Transactional - When an operation has the property that it either completes, or if it does not complete due to a failure, it either undoes its own effects or has the ability to complete at a later time when the failure is repaired.

References

UML References

The Unified Modeling Language User Guide by Grady Booch, James Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language Reference Manual* by James Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Software Development Process* by Ivar Jacobson, Grady Booch, and James Rumbaugh, *The Complete UML Training Course* by Grady Booch, James Rumbaugh, Ivar Jacobson; *UML in a Nutshell* by Sinan Si Alhir, *The Object Constraint Language* by Jos Warmer and Anneke Kleppe, *Applying Use Cases* by Geri Schneider and Jason P. Winters, *The Rational Unified Process An Introduction Second Edition* by Krutchen, *Object Solutions Managing the Object-Oriented Project* by Grady Booch, *Objects, Components, and Frameworks with UML - The Catalysis Approach* by Desmond D'Souza and Alan Wills, *Use Case Driven Object Modeling with UML - A Practical Approach* by Doug Rosenberg; *Use Case: Requirements in Context* by Daryl Kulak, *Business Modeling with UML Business Patterns at Work* by Hans-Eric Eriksson and Magnus Penker, *Analysis Patterns Reusable Object Models* by Martin Fowler, *Building Object Applications That Work* (SIGS Books, 1997) by Scott Ambler; *Object-Oriented Software Metrics* by Lorenz and Kidd

Patterns References

Design Patterns by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Patterns in Java Vol 1 and 2* by Mark Grand *A System of Patterns* by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal

Enterprise Java References

Developing Java Enterprise Applications by Stephen Asbury and Scott R. Weiner, *Client/Server Programming with Java and CORBA* by Robert Orfali and Dan Harkey, *Java Application Frameworks* by Asbury and Giovanni, *Enterprise Java Beans* by Valeski
Key Web Sites - www.rational.com.com, www.omg.org, www.cetus-links.org, www.sema4usa.com, **Free Magazines-** Software Development - www.sdmagazine.com/sdonline/fr_subs.html; Distributed Computing - www.distributedcomputing.com; Application Development Trends - www.adtmag.com

Richard Felsing, 960 Scotland Dr, Mt Pleasant, SC 29464 843-881-3648 dfelsing@home.com 1/29/2001