SWE 621: Software Design

Lecture Notes on Software Design

Spring Semester 2002

Hassan Gomaa

Dept of Information & Software Engineering
George Mason University
Fairfax, VA

SWE 621: Software Design
Table Of Contents -1

SWE 621: Software Design
Table Of Contents -2

SWE 621: Software Design
Table Of Contents -3

SWE 621: Software Design
Course Content

- Introduction to Software Design
  - Software Design Process
  - Design Concepts
  - Introduction to Software Design Methods
- Survey of Software Design Methods
- Object-Oriented Analysis and Modeling Method
- Object-Oriented Design Method
- Course Review

Software Design

What is design?

  noun: mental plan, preliminary sketch or outline

  verb: to conceive in the mind; to invent

What is software design?

  As a product

    Output of design process

  As a process

    Approach to doing design

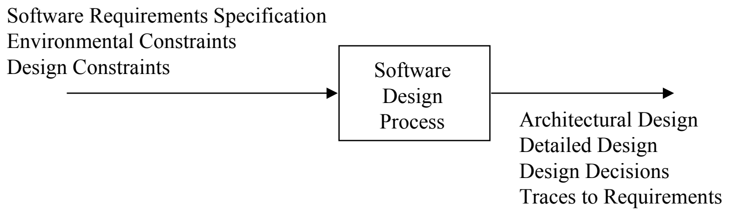
Software Design Activities

Architectural design
- Structure system into components
- Define the interfaces between components

Detailed design of each component
- Define internal logic
- Define internal data structures

Data design
- Define file structures
- Logical database design

Software Design



Inputs To Software Design

- Software requirements specification
  - Describes WHAT system shall do not HOW
  - External view of system to be developed
- Environmental constraints
  - Hardware, language, system usage
- Design constraints
  - Design method
  - Design notation

Outputs From Software Design

- Architectural Design
  - Overall description of software structure
    - Textual and Graphical
  - Specification of software components and their interfaces
  - Data Design
- Detailed Design of each component
  - Internal logic
  - Internal data structures
- Design decisions made
  - Design rationale
- Traces to requirements

Software Design Process

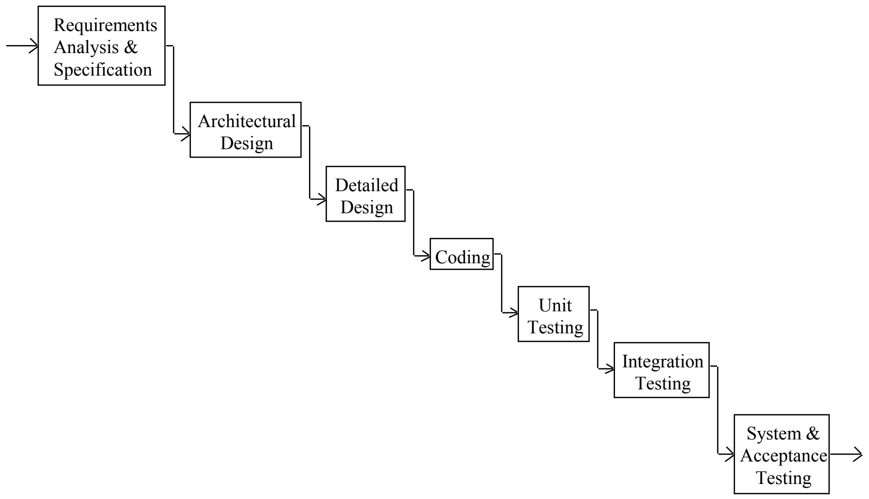
Reference: Gomaa text, Chapter 5

Software life cycle (a.k.a. software process)
- Phased approach to software development

Software life cycle (a.k.a. process) models
- Waterfall – limitations of Waterfall Model
- Incremental - evolutionary prototyping
- Exploratory - throwaway prototyping
- Spiral model – risk driven process model

Software Life Cycle

Waterfall Model



Software Life Cycle Model
Software Definition

- Requirements Analysis and Specification
  - Analysis of user's problem
  - Specification of "what" system shall provide user
- Architectural Design
  - Specification of "how" system shall be structured into components
  - Specification of interfaces between components
  - Data Design

Software Life Cycle Model
Software Construction

- Detailed Design
  - Internal design of individual components
    - Design of logic and data structures
- Coding
  - Map component design to code
- Unit Testing
  - Test individual components

Software Life Cycle Model
Software Integration and Test

Integration Testing

Gradually combine components and test combinations

System Testing

Test of entire system against software requirements

Acceptance Test

Test of entire system by user prior to acceptance

Software Life Cycle Model
Software Maintenance

Modification of software system after installation and acceptance

Fix software errors

Improve performance

Address changes in user requirements

Often implies significant software redesign

Limitations of Waterfall Model

Does not show iteration in software life cycle
Does not show overlap between phases
Software requirements are tested late in life cycle
Operational system available late in life cycle

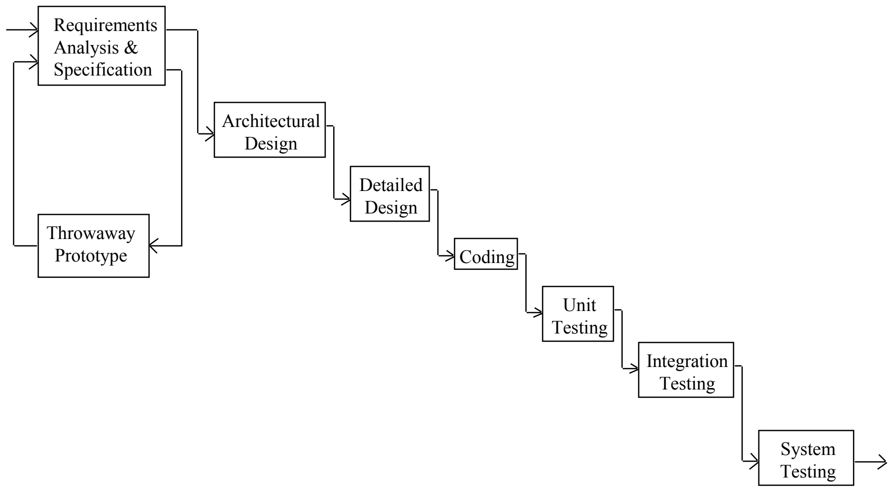
Prototyping During Requirements Phase

Problem
- Software requirements are tested late in life cycle

Solution
- Use throw-away prototyping

Help ensure requirements are understood
Also first attempt at designing system
- Design of key file and data structures
- Design of user interface
- Early design tradeoffs

Impact of Throwaway Prototyping on Software Life Cycle

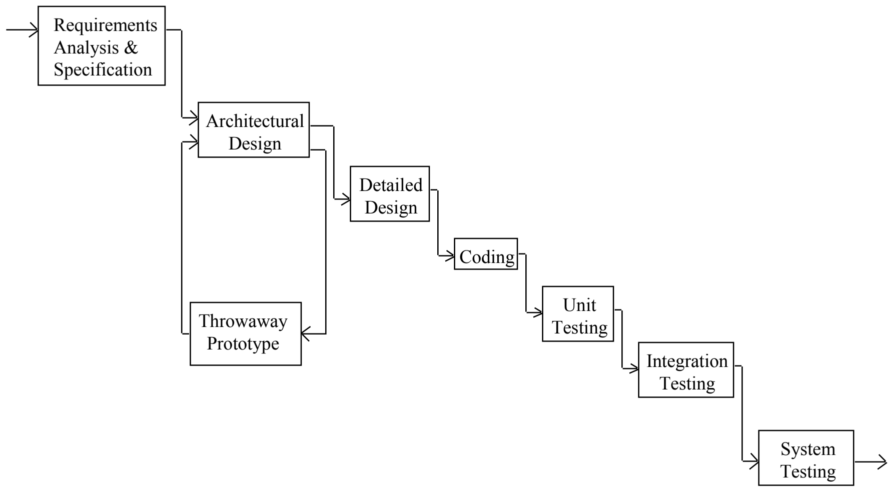


Throw-away Prototyping in Design

Objectives
- Test design early
- Experiment with alternative design decisions

Examples of prototyping in design
- Algorithm design
  - Experiment with - speed, accuracy
- Early performance analysis
  - Measure timing parameters
- User interface

Impact of Throwaway Prototyping on Architectural Design Phase



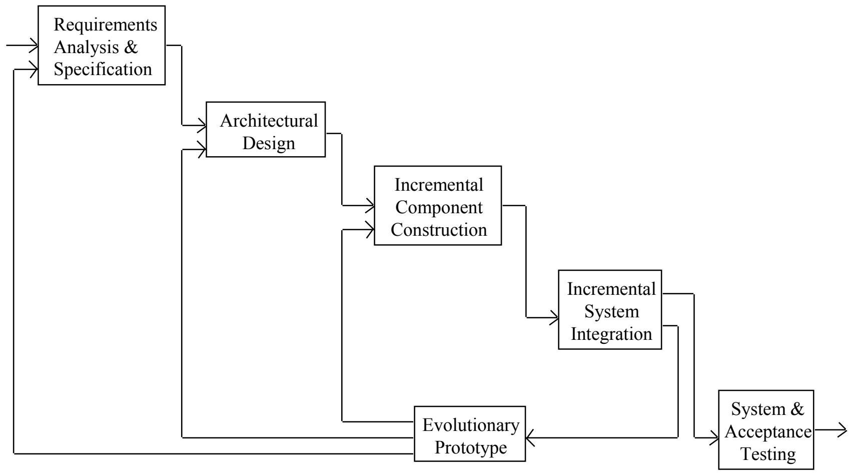
Incremental Development

Problem
- Operational system available late in life cycle

Solution
- Use incremental development
- Also known as evolutionary prototyping

Objective
- Subset of system working early
- Gradually build on
- Prototype evolves into production system

Incremental Development Software Life Cycle


Requirements Analysis & Specification
Architectural Design
Incremental Component Construction
Incremental System Integration
Evolutionary Prototype
System & Acceptance Testing

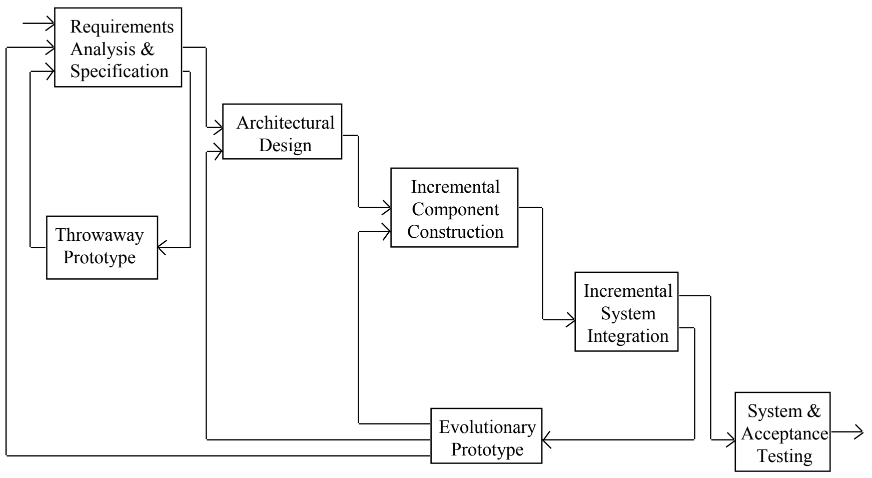
Should Prototype Evolve into Production System?

Tradeoff
- Rapid development
- Quality of product

Throw-away prototype
- Speed, not quality is goal
- Must not evolve into production system

Evolutionary prototype
- Must emphasize quality
- Maintainability is key issue

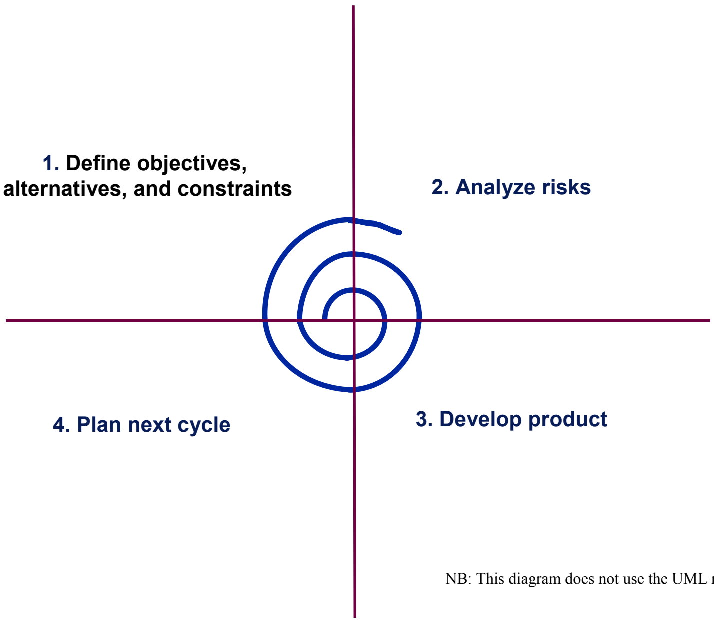
Combined Throwaway Prototyping / Incremental Development Software Life Cycle Model


Requirements Analysis & Specification
Throwaway Prototype
Architectural Design
Incremental Component Construction
Incremental System Integration
Evolutionary Prototype
System & Acceptance Testing

Spiral Process Model (SPM)

- SPM consists of four main activities that are repeated for each cycle (Fig. 5.6):
  - Defining objectives, alternatives and constraints
  - Analyzing risks
  - Developing and verifying product
  - Spiral planning
- Number of cycles is project specific
- Risk driven process
  - Analyze risks in second quadrant

Figure 5.6 The spiral process model



NB: This diagram does not use the UML notation

Design Concepts

Reference: Gomaa text, Chapter 3

- Allow us to Manage and Reduce Complexity
  - Abstraction
  - Objects and classes
  - Information Hiding
  - Concurrency
  - Finite State Machines

Abstraction

- Concentrate on problem at:
  - Some level of generalization
  - Ignore (for now) lower level details
- Abstraction in Software Development
  - Requirements Definition
    - Define “What” system will do before “How”
  - Architectural Design
    - Define system overall structure before module details
  - Detailed Design
    - Design module before coding

Objects and Classes

- Objects represent “things” in real world
  - Provide understanding of real world
  - Form basis for a computer solution
- An Object (object instance) is a single “thing”
  - E.g., John’s car
  - Mary’s account
- A Class (object class) is a collection of objects with the same characteristics
  - E.g., account, employee, car, customer
- **Figure 2.2 UML notation for objects & classes**
- **Figure 3.1 Example of classes and objects**

Attributes

- Attribute
  - Data value held by object in class
- Example of Attributes
  - E.g., account number, balance
- Each object instance has specific value of attribute
  - John's account number is 1234
  - Mary's account number is 5678
- Attribute name is unique within class
- **Figure 3.2 Example of class with attributes**

Classes and Operations

- Operation
  - Is function or procedure that may be applied to objects in a class
  - All objects in class have same operations
- Class has one or more operations
  - Operations manipulate values of attributes maintained by object
- Operations may have
  - Input parameters
  - Output parameters
  - Return value
- Signature of operation
  - Operation's name
  - Operation's parameters
  - Operation's return value
- Interface of class
  - Set of operations provided by class
- **Figure 3.3 Class with attributes and operations**

Information Hiding

Each object hides design decision

    E.g., data structure

        interface to I/O device

Information hiding object

    Hides (encapsulates) information

    Accessed by operations

Basis for Object-Oriented Design

Advantage

    Objects are more self-contained

    Results in more modifiable -> maintainable system

Example of Information Hiding

- Example of Stack
- Conventional approach
  - Stack data structure is global
  - Stack accessed by modules
  - Module corresponds to procedure / function / subroutine
- Problem
  - Change to stack data structure has global impact
- Consider
  - Array implementation (Fig. 3.4) changed to
  - Linked list implementation (Fig. 3.6)
- Every module is impacted by change

Example of Information Hiding

- Example of Stack
- Information hiding solution
  - Hide stack data structure and internal linkage
  - Specify operations on stack data structure
  - Access to stack only via operations
- Consider
  - Array implementation (Fig. 3.5) changed to
  - Linked list implementation (Fig. 3.7)
- Change to stack only impacts Stack object

Inheritance in Design

- Subclass inherits generalized properties from superclass
- Inheritance
  - Allows sharing of properties between classes
    - Property is Attribute or Operation
  - Allows adaptation of parent class (superclass) to form child class (subclass)
- Subclass inherits attributes & operations from superclass
  - May add attributes
  - May add operations
  - May redefine operations

Sequential & Concurrent Problems

Sequential problems
- Activities happen in strict sequence
  - E.g., compiler, payroll
- Sequential solution = program

Concurrent problems
- Many activities happen in parallel
  - E.g., multi-user interactive system, air traffic control system
- Sequential solution to concurrent problem increases design complexity

Concurrent and Real-Time Systems

- Concurrent System
  - Consists of many activities (tasks) that execute in parallel
- Real-Time system
  - Concurrent system with timing deadlines
- Distributed application
  - Concurrent system executing on geographically distributed nodes

Concurrent Processing

- Characteristics of concurrent task (process, active object)
  - One sequential thread of execution
  - Represents execution of
    - Sequential program
    - Sequential component of concurrent program
- Concurrent system
  - Many tasks execute in parallel
- Tasks need to interact with each other (Fig. 3.10)

Finite State Machines

- Finite number of states
  - Only in one state at a time
- Transition
  - Change of state
  - Caused by event
  - Transition to same or different state
  - Action may result from state transition
- Notation
  - State transition diagram
  - State transition table
  - Statechart
  - Examples of statecharts (Figures 10.1 - 10.3)

Software Design Terminology

Design concept or principle
- Fundamental idea that can be applied to designing a system, e.g., information hiding

Design notation or representation
- A means of describing a software design
  - Textual and Graphical, e.g., UML

Design strategy
- Overall plan and direction for performing design

Design structuring criteria
- Guidelines for decomposing a system into its parts

Software Design Method

Systematic approach for creating a design
- Design decisions to be made
- Order in which to make them

Describes sequence of steps for producing a design
- Based on set of design concepts
- Employs design strategy(ies)
- Provides design structuring criteria
- Documents resulting design using design notation(s)

Example of Software Design Method COMET

Design concepts
- Finite state machine, concurrent task, information hiding

Design structuring criteria
- Object, subsystem and task structuring criteria

Design strategy
- Develop analysis model, then map to design model

Design notation
- UML (Unified Modeling Language)

Boundary between Requirements & Design

- Black Box Requirements Methods
  - Distinct split between Requirements and Design
- Software Analysis methods
  - Problem-oriented perspective
  - Define problem-oriented components (functions, objects) & interfaces
- Most software design methods
  - Have analysis phase, e.g, SA/SD, OOA/OOD
- Decisions made in Analysis
  - Have major impact on Design
    - Scope of component
    - How it interfaces to other components
- Problem-Oriented Analysis
  - Considered part of Design Process
  - Usually first phase of design

Survey of Software Design Methods

Hassan Gomaa
Dept of Information & Software Engineering
George Mason University

Survey of Software Design Methods

- Structured Design
- DARTS
- Jackson System Development
- "Conventional" Object-Oriented Design
- ADARTS and CODARTS
- Comparison of Software Design Methods

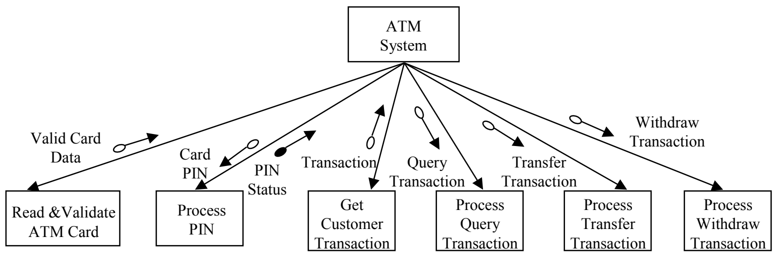
Structured Design

- Program Design Method
- Data flow diagrams mapped to structure charts
- Leaf-level transformations mapped to functional modules
- Module structuring criteria
  - Module cohesion
    - To identify strength of module
  - Module coupling
    - To determine connectivity between modules
- Strategies for mapping data flow diagram to Structure chart
  - Transform Analysis
  - Transaction Analysis

Structure Chart

- Used in Software Architectural Design
- Shows decomposition of program into modules
- Module corresponds to
  - Procedure
  - Function
  - Information Hiding Module
- Shows module calling structure
- Defines interfaces between modules
  - Input parameters
  - Output parameters

Example of Structure Chart

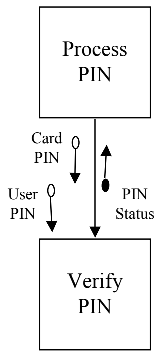


Structured Design Modularization Criteria

- Coupling
  - Measure of connectivity between modules
    - Data coupling
    - Common coupling
- Cohesion
  - Measure of strength or unity within module
    - Functional cohesion
    - Informational cohesion

Module Coupling

- Data Coupling
  - Parameter passing
    - Data element
    - Array

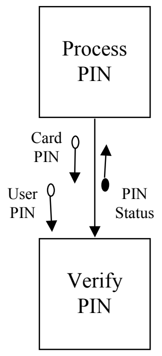


Example of Data Coupling

- Common Coupling
  - Two or more modules reference externally declared data structure
    - E.g., FORTRAN common block
  - Highly error prone
    - Leads to "non-modular" tightly coupled systems

Functional Cohesion

- Module performs one "problem-related" function
- Low level module
  - Read Card
  - Validate Card
- Supervisory module
  - Process PIN
  - Process Query Transaction
- Advantage
  - Contents of modules highly unified
- Disadvantage
  - Could lead to several small modules

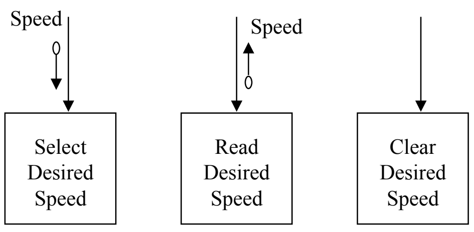


Informational Cohesion

- Provides Information Hiding
  - Module consists of group of procedures and / or functions
  - Shared hidden data structure
  - One function or procedure executed for each call
  - Data structure accessed indirectly via access procedures / functions
  - E.g, stack
    - Push
    - Pop

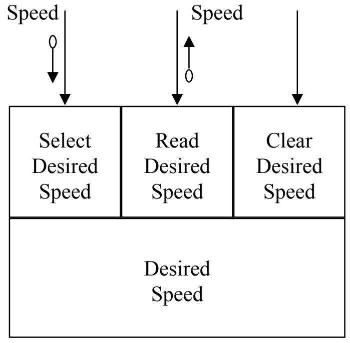
Typical Structured Design Solution Modules for Desired Speed

- Functional cohesion
  - One function / module
    - Select Desired Speed
    - Read Desired Speed
    - Clear Desired Speed
- Common coupling
  - Global data
    - Desired Speed



Structured Design Notation
Informational Cohesion Module

- Informational cohesion
  - Each function is operation of information hiding module (IHM)
    - Select Desired Speed
    - Read Desired Speed
    - Clear Desired Speed
- Data coupling
  - No global data
    - Desired Speed maintained by IHM

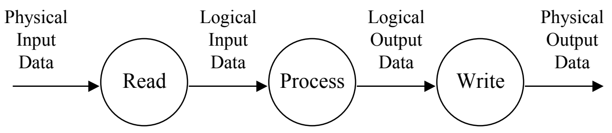


Initialization and Termination Modules

- Temporal cohesion
- Encourages use of global memory
- Solution
  - Initialize variable in module where it is used
  - Use information hiding module (informational cohesion)
    - Data structure hidden
    - Access procedures include
      - Initialization procedure
      - E.g., select desired speed

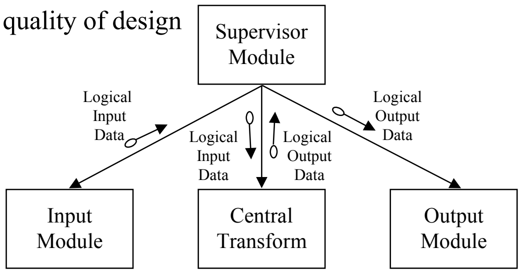
Design Strategies Transform Analysis

- Draw DFD
  - Input, process, output
- Analyze DFD and determine
  - Input branches
  - Output branches
  - Central transform



Transform Analysis

- Develop top level structure chart
  - Level 1
    - Master (supervisor) module
  - Level 2 - one module for each
    - Input branch
    - Output branch
    - Central transform
- Decompose each Level 2 module into its components
- Analyze quality of design



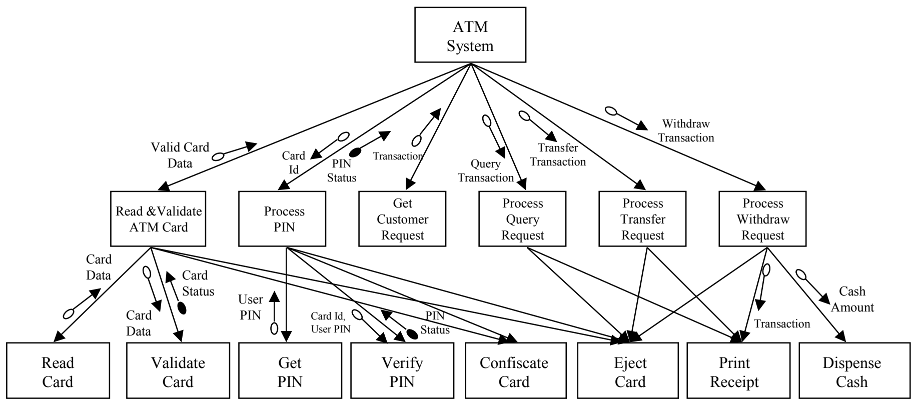
Design Strategies Transaction Analysis

- Transaction
  - Element of data that triggers action
- Transaction center
  - Receives all transactions
  - Separates individual transactions by type
  - Calls transaction module to process transaction
- Transaction module
  - One module for each transaction type
- Action modules
  - Perform individual actions
  - Called by transaction modules
  - May be shared by different transaction modules

Example of Transaction Analysis ATM System

- Transactions
  - Withdraw
  - Query
  - Transfer
- Withdraw
  - Cash dispensed
  - Account decremented
  - Card ejected
  - Receipt printed
- Query
  - Account read
  - Receipt printed
- Transfer
  - “From” Account decremented
  - “To” Account incremented
  - Receipt printed

ATM Client Subsystem Structure Chart



Design Approach for Real-Time Systems (DARTS)

- Extends Real-Time Structured Analysis and Design
  - Structure system into tasks
  - Define interfaces between tasks
- Steps in DARTS
  - Develop Real-Time Structured Analysis specification
  - Structure system into tasks
  - Define task interfaces
  - Structure each task into modules using Structured Design

Design Approach for Real-Time Systems (DARTS)

- Extends Real-Time Structured Analysis and Design
  - Structure system into tasks
  - Define interfaces between tasks
- Steps in DARTS
  - Develop Real-Time Structured Analysis specification
  - Structure system into tasks
  - Define task interfaces
  - Structure each task into modules using Structured Design

Step 2: Structure System into Tasks

- Characteristics of concurrent task (process)
  - One sequential thread of execution
    - Sequential program
    - Sequential component of concurrent program
- Structure RTSA specification into concurrent tasks
- Analyze transformations on data flow/control flow diagrams
- Apply task structuring criteria
- Develop preliminary task architecture diagrams
- Develop task behavior specifications

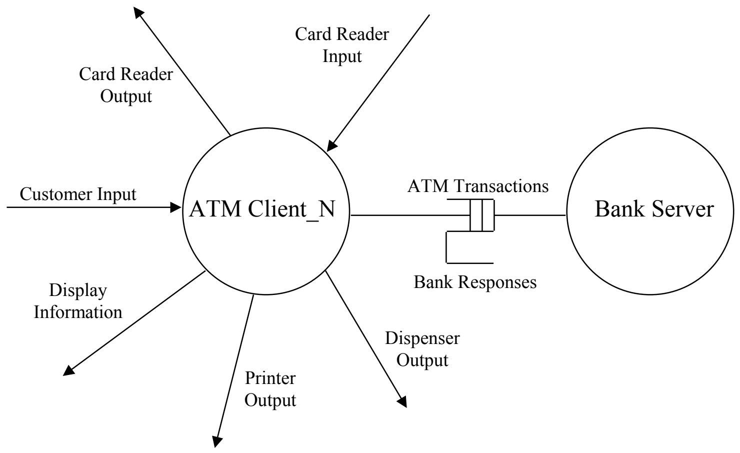
Step 3: Define Task Interfaces

- Map Data Flow Diagram interfaces to task interfaces
- Data flows
  - Message communication
- Event flows
  - Event synchronization
  - Message communication
- Data stores
  - Information hiding modules
- Update task architecture diagrams

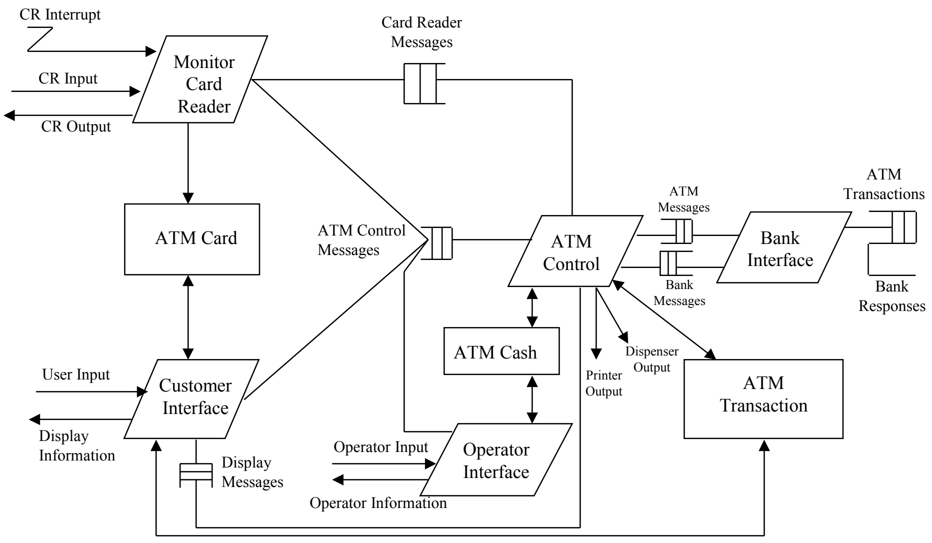
Step 4: Design Each Task

- Task is a sequential program
- Design task using program design method
- E.g., Structured Design
  - Transform Analysis
  - Transaction Analysis
- Steps
  - Develop data flow diagram for task
  - Develop structure chart showing modules
  - Define module interfaces

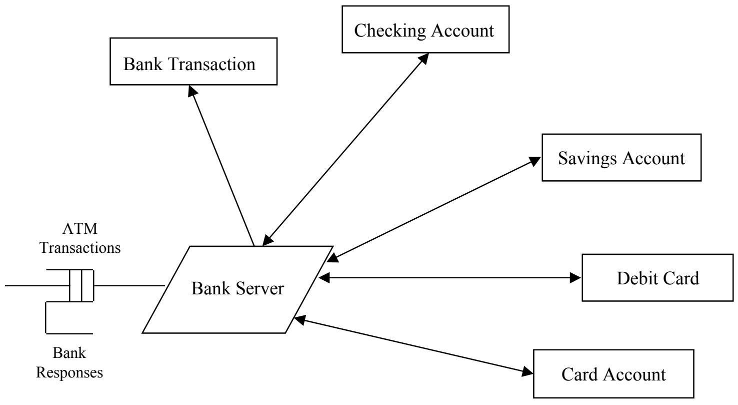
Distributed System Architecture



ATM Client Task Architecture Diagram



Task Architecture Diagram for Bank Server



Jackson System Development (JSD) Overview

- Modeling approach to software design
- JSD design
  - Models behavior of real world entities over time
  - Each entity is mapped to concurrent task
  - Tasks communicate with each other
- JSD design is mapped to an implementation
  - Single program
  - Multi-tasking implementation
  - Ada implementation
  - Use JSP notation

Jackson System Development Steps in Method

- Modeling phase
  - Model entities using concurrent tasks
- Network phase
  - Add functionality
  - Define interfaces between tasks
- Implementation phase
  - Map JSD design to implementation

JSD Modeling Phase

- Identify real-world entities
- Describe behavior of each entity
  - Sequence of events experienced by entity
  - Represented by entity structure diagram
  - May be many entities of same type
- Map each entity to software model task
  - Task has same structure as entity
  - Task receives real world events as inputs
  - Task spec built around entity structure

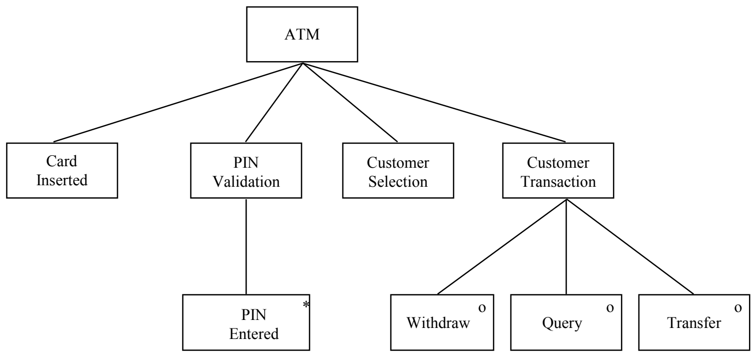
JSD Network Phase

- Define communication interfaces between model tasks
  - Message communication (Data stream)
  - State vector connection
- Add functionality to model tasks
  - If matches structure of model task
- More complex functionality requires additional tasks
  - Function tasks

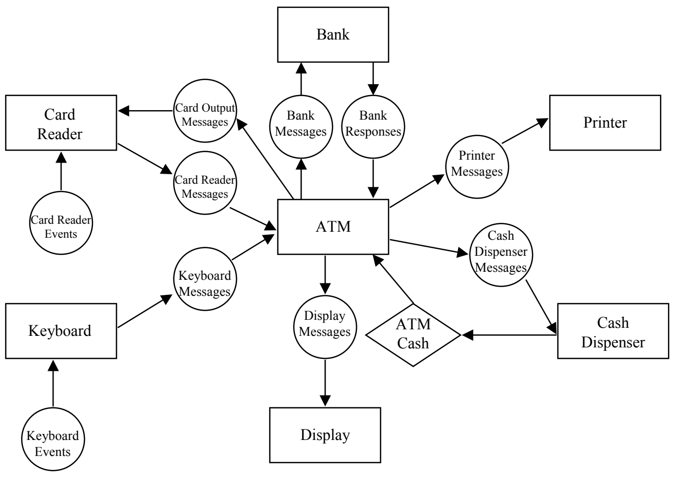
JSD Implementation Phase

- Map logical model and function tasks
  - Onto one or more physical tasks
  - Uses JSP program inversion
- Multiple logical tasks of same type
  - Mapped to one physical task
  - State vector mapped to record
- Program inversion

ATM Entity Structure Diagram



ATM System Network Diagram



Naval Research Lab Software Cost Reduction Method (NRL)

- Based on Work of David Parnas
- Information hiding
  - – Main criterion for decomposing system into modules
- Information hiding modules
  - – Identify design decisions likely to change
  - – Module for each changeable design decision
  - – Each changeable decision is "secret" of module
- Emphasis on Information Hiding
  - – -> More maintainable and reusable components

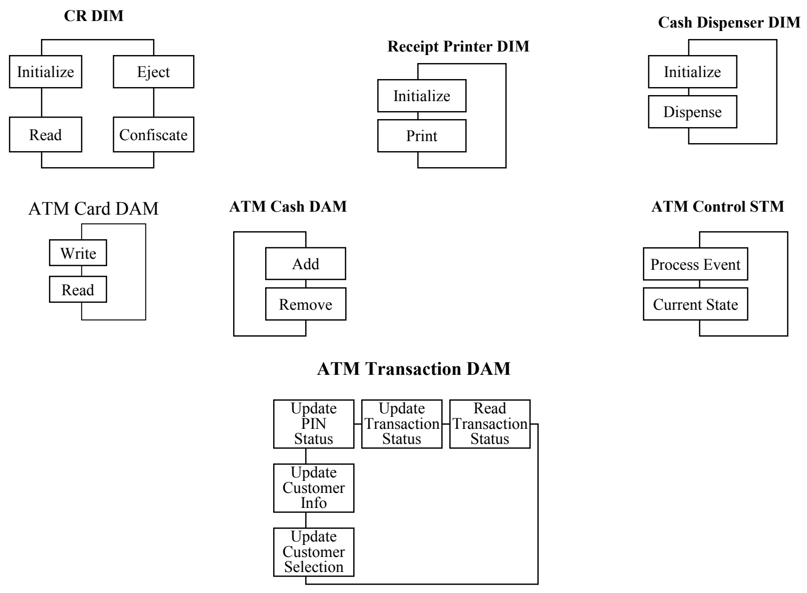
NRL Module Hierarchy

- Categorization of modules
- Three main categories needed for complex systems
  - – Hardware hiding modules
    - Hides hardware dependent details
  - – Behavior hiding modules
    - Hides system requirements
  - – Software decision modules
    - Hides design decisions

Example of Information Hiding Modules

- Device Interface Modules
  - Engine
  - Brake
  - Cruise Control Lever
  - Shaft
  - Throttle
- Behavior Hiding Modules
  - Data Abstraction Modules
    - Current Speed
    - Desired Speed
  - State Transition Modules
    - Cruise Control

ATM Client Information Hiding Modules



“Early” Object-Oriented Design

- Taxonomy of object-oriented design methods (based on Wegner)
- Object-based design method = Objects
  - Objects are information hiding modules
- Class-based design method = Objects + Classes
  - Classes are object types
- Object-oriented design method = Objects + Classes + Inheritance
  - Inheritance is specialization of a class
    - Superclass: account,
    - Subclasses: checking account, savings account

Characteristics of Object

- Has state
- Characterized by operations
  - It provides
  - Uses from other objects
- Instance of some class
- Denoted by name
- Restricted visibility of and by other objects
- Viewed either by its specification or implementation

Example of Objects

Concrete objects
- Represent real world entities
  - Card Reader
  - Cash Dispenser
  - Receipt Printer

Abstract objects
- Represent conceptual entities
  - Checking Account
  - Savings Account
  - Bank Transaction

Steps in Object-Oriented Design

- General approaches, e.g., Booch
  - Identify objects in the problem domain
    - Map real-world entities onto software objects
    - Informal strategy used for identifying objects
  - Determine software object's interfaces
    - Operations provided by object
    - Operations used by object
  - Organize objects into class hierarchies
    - Develop class and object diagrams

Object Structuring

- "Objects are ripe for the picking".... B. Meyer
- Noun / verb approach
  - Underline nouns and verbs in specification
    - Common noun: Class of objects
    - Proper noun: Instance of object
    - Verb: Operation
  - Not practical for large/medium scale systems
- Identify objects from Structured Analysis specification
- Object identification in Object-Oriented Analysis
  - Object structuring criteria

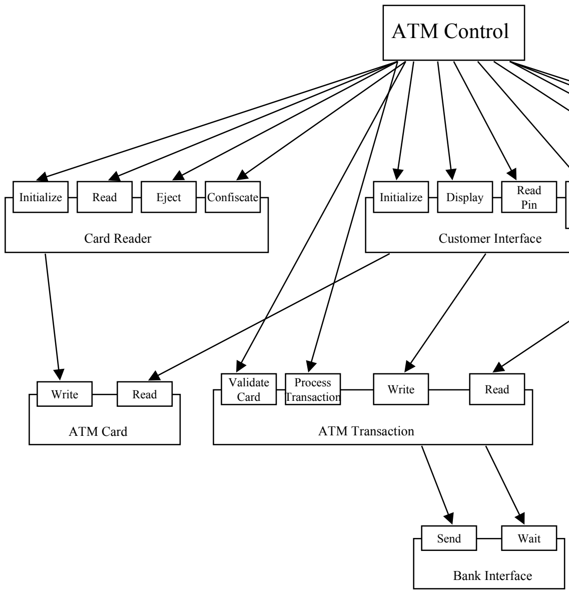
Banking System
Object Structuring

Examples of Device Interface Objects
- Card Reader
- Cash Dispenser
- Receipt Printer

Examples of Data Abstraction Objects
- Checking Account
- Savings Account

Example of Control Object
- ATM Control

Sequential Object Oriented Design for ATM Client



Assessment of "Early" Object-Oriented Design - Strengths

- OOD is based on key concepts in software design
  - Information hiding, Classes, Inheritance
- Structuring system into objects using information hiding
  - Makes system more maintainable
    - Objects are more self contained
    - Objects potentially reusable
- Inheritance
  - Allows objects to be adapted in controlled manner
- Good for design of sequential systems
- Maps well to object-oriented programming languages

Assessment of "Early" Object-Oriented Design - Limitations for Concurrent Design

- Does not distinguish between
  - Active objects (concurrent tasks) and
  - Passive objects (instances of information hiding classes)
- Assumes same criteria for task as class structuring
- Can result in large number of tasks
  - Tasking overhead can be large
- Does not address
  - Inter-task communication and timing issues

Comparison of Software Design Methods

	OOA	Task Structuring	IHM/Class Structuring	FSMs
SAD	N	Little guidance	Functional modules	Yes
DARTS	N	Criteria provided	IHMs for data stores	Yes
JSD	N	Criteria provided	Not supported	No
NRL	N	Some guidance	IHM criteria provided	Yes
OOD	Y	Little guidance	Class criteria provided Inheritance provided	Yes
COMET	Y	Criteria provided	Class criteria provided Inheritance provided	Yes

Object-Oriented Software Engineering with UML

Hassan Gomaa
Dept of Information & Software Engineering
George Mason University

Reference: H. Gomaa, “Chapter 6 - Designing Concurrent, Distributed, and Real-Time Applications with UML”, Addison Wesley Object Technology Series, July, 2000

Object-Oriented Analysis and Design

- Object-Oriented Analysis and Design Method combines:
  - Object Modeling
    - J. Rumbaugh et al, "Object-Oriented Modeling and Design", Prentice Hall, 1991
  - Use cases
    - I Jacobson et al, "Object-Oriented Software Engineering", Addison Wesley, Reading MA, 1992.
  - Statecharts (Harel)
  - Sequence Diagrams (Rumbaugh, Jacobson)
  - Object Collaboration Diagrams (Booch)
  - Concurrent, Distributed, & Real-Time Design (Gomaa)
- Unified Modeling Language (UML)
  - Standardized notation for object-oriented development

Object Modeling Technique (OMT)

- Addresses
  - Structural (static) aspects of problem
    - Object Model
      - Information model
  - Dynamic aspects of problem
    - Dynamic Model
      - Uses state transition diagrams and scenarios
  - Functional aspects of problem
    - Functional model
      - Data flow diagrams

Object-Oriented Software Engineering

- Based around *use case* (scenario) concept
- OOSE supports five models:
- Requirements model
  - Defines functional requirements in terms of use cases
- Analysis model
  - Defines objects and how they participate in use cases
- Design model
  - Maps object structure to operational environment
- Implementation model
  - Source code of system
- Test model
  - Testing of system

Unified Modeling Language (UML)

- Standardized notation for object-oriented development
  - Combines notations of OMT, Booch, and use cases
- Needs to be used with an analysis and design method
  - Notation provides more support for analysis than design
- Intended for all types of OO software development
- UML notation used for OO analysis and design method for concurrent, real-time and distributed applications
  - Concurrent Object Modeling and architectural design mEThod (COMET)
- H. Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison Wesley Object Technology Series, July, 2000.

Unified Modeling Language (UML) Diagrams

Reference: Gomaa text, Chapter 2

- Use Case Diagrams
  - Fig. 2.1
- Class Diagrams
  - Figs. 2.3-2.4
- Collaboration Diagrams
  - Fig. 2.5
- Sequence Diagrams
  - Fig. 2.6
- Statecharts
  - Figs. 2.7-2.8
- Deployment diagrams
  - Fig. 2.13

Object-Oriented Software Life Cycle

Requirements & Analysis Modeling

- Requirements Modeling (Fig. 6.1)
  - Use Case Modeling
    - Define software functional requirements in terms of use cases and actors
- Analysis Modeling (Fig. 6.1)
  - Static Modeling
    - Define structural relationships between classes
    - Depict classes and their relationships on class diagrams
  - Dynamic Modeling
    - Define statecharts for state dependent objects
    - Defines how objects participate in use cases using collaboration diagrams or sequence diagrams

Object-Oriented Software Life Cycle

Architectural Design (Fig. 6.1)

- Maps analysis model (emphasis on problem domain) to design model (emphasis on solution domain)
- Structure system into subsystems
- Design each subsystem
- Sequential Applications
  - Emphasis on OO concepts
    - Information hiding, classes, inheritance
- Concurrent, Distributed and Real-Time Applications
  - Emphasis on
    - OO concepts
    - Concurrent tasking

Object-Oriented Software Life Cycle
Incremental Development

- Complete architectural design of software
- Incremental Software Construction (Fig. 6.1)
  - Select subset of system based on use cases
    - Detailed design, code, unit test of components in subset
- Incremental Software Integration (Fig. 6.1)
  - Integration testing of each system increment
    - Integration test based on use cases
  - Develop integration test cases for each use case
  - White box testing
    - Test interfaces between components in use case

Object-Oriented Software Life Cycle
System Testing

- System Testing (Fig. 6.1)
  - Includes functional testing of system
    - Testing of functional requirements
  - Black box testing
    - Based on use cases
- Need system test for each increment released to user
- Independent test team
  - Goal is to break system
  - Thorough systematic test of system before release to users

Steps in Using COMET/UML

1 Develop Object-Oriented Requirements Model
- Develop Use Case Model (Chapter 7)

2 Develop Object-Oriented Analysis Model
- Develop static model of problem domain (Chapter 8)
- Structure system into objects (Chapter 9)
- Develop statecharts for state dependent objects (Chapter 10)
- Develop object interaction diagrams for each use case (Chapter 11)

3 Develop Object-Oriented Design Model
- Design Overall Software Architecture (Chapter 12)
- Design Distributed Component-based Subsystems (Chapter 13)
- Structure Subsystems into Concurrent Tasks (Chapter 14)
- Design Information Hiding Classes (Chapter 15)
- Develop Detailed Software Design (Chapter 16)

COMET OO Analysis and Design

- UML Notation
  - Supports both Analysis and Design concepts
- COMET/UML method
  - Separate requirements activities, analysis activities and design activities
- Requirements Modeling
  - Consider system as black box
  - Develop Use Case Model

COMET OO Analysis and Design

- Analysis modeling
  - Consider analysis of problem domain
  - Determine problem oriented objects and classes
  - Analyze static viewpoint in Static Model
    - Classes, relationships, attributes
  - Analyze dynamic viewpoint in Dynamic Model
    - Statecharts
    - Object interaction model
      - Consider objects supporting each use case
      - Analyze sequence of interactions between objects
      - Analyze information passed between objects

COMET OO Analysis and Design

- Design Model
  - Consider solution domain
  - Make decisions about overall software architecture
  - Make decisions about distributed component-based subsystems
  - Make decisions about characteristics of objects
    - Active or Passive
  - Make decisions about characteristics of messages
    - Asynchronous or Synchronous (with/without reply)
  - Make decisions about class interfaces
    - Operations and parameters
  - Make detailed design decisions

Use Case Modeling

Hassan Gomaa
Dept of Information & Software Engineering
George Mason University

Reference: H. Gomaa, "Chapter 7 - Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison Wesley Object Technology Series, July, 2000

Use Case Modeling

- Use Case
  - Describes sequence of interactions between user (actor) and system
- **Figure 2.1 UML notation for use case diagram**
- Initially developed in Use Case model
  - Shows interaction between actor and black box system
- Use cases refined in Dynamic Model
  - Show objects participating in use case
  - Develop collaboration diagrams or sequence diagrams
- Use cases refined further in Design Model
- Use cases form basis of integration & system test cases

Actors

- Actor models external entities of system
- Actors interact directly with system
  - Human user (**Figure 7.1)**
  - External I/O device (**Figure 7.2)**
  - Timer (**Figure 7.3)**
  - External system **(Figure 7.4)**
- Actor initiates actions by system
  - May use I/O devices or external system to physically interact with system
  - Actor initiates use cases

Use Cases

- Define system functional requirements in terms of Actors and Use cases
  - Narrative description
- Identifying use cases
  - Consider requirements of each actor who interacts with system
  - Use case is a complete sequence of events initiated by an actor
    - Use case starts with input from an actor
    - Describes interactions between actor and system
    - Provides value to actor
  - Basic path
    - Most common sequence
  - Alternative branches
    - Variants of basic path
      - E.g., for error handling
- **Figure 7.5 Banking system actor & use cases**

Documenting Use Cases

- Name
- Summary
  - Short description of use case
- Dependency (on other use cases)
- Actors
- Preconditions
  - Conditions that are true at start of use case
- Description
  - Narrative description of basic path
- Alternatives
  - Narrative description of alternative paths
- Postcondition
  - Condition that is true at end of use case

Use Case Relationships

- **Include** relationship
  - Identify common patterns (sequences) in several use cases
  - Extract common pattern into **abstract use case**
  - Concrete use cases **include** abstract use case
  - **Figure 7.7 Example of abstract use case and include relationship**
- **Extend** relationship
  - Use case A is an extension of use case B
  - Under certain conditions use case B will be extended by description given in use case A
  - Same use case can be extended in different ways
  - **Figure 7.6 Example of extend relationship**

Use Case Package

- Use Case Package
  - Encompasses related group of use cases
  - Represent high level requirements
- Use Case Package Structuring
  - Group use cases into packages based on
    - Major subset of system functionality
    - Related use cases started by the same actor
- **Figure 7.8 Example of use case package**

Static Modeling

Hassan Gomaa
Dept of Information & Software Engineering
George Mason University

Reference: H. Gomaa, "Chapter 8 - Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison Wesley Object Technology Series, July, 2000

Object-Oriented Software Life Cycle

Requirements & Analysis Modeling

- Requirements Modeling
  - Use Case Modeling
    - Define software functional requirements in terms of use cases and actors
- Analysis Modeling
  - Static Modeling
    - Define structural relationships between classes
    - Depict classes and their relationships on class diagrams
  - Dynamic Modeling
    - Define statecharts for state dependent objects
    - Defines how objects participate in use cases using collaboration diagrams or sequence diagrams

Modeling Objects in the Problem Domain

- Analyze problem domain
  - Map real-world objects to software objects
  - Objects less likely to change than functions of system
- Object structuring criteria
  - Guidelines for structuring objects
- Structural view of objects
  - Static (Object) Model
- Dynamic view of objects
  - Dynamic model
    - Statecharts
    - Sequence Diagrams or Object Collaboration Diagrams

Static Modeling

- Define structural relationships between classes
  - Depict classes and their relationships on class diagrams
- Relationships between classes
  - Associations
  - Composition / Aggregation
  - Generalization / Specialization
- Static Modeling during Analysis
  - System Context Class Diagram
    - Depict external classes and system boundary
- Static Modeling of Entity classes
  - Persistent classes that store data

Static Modeling

- Class
  - Real world entity type about which information is stored
  - Represents a collection of identical objects (instances)
  - Described by means of attributes (data items)
  - Has operations to access data maintained by objects
  - Each object instance can be uniquely identified
- Relationships between classes
  - Associations
  - Composition / Aggregation
  - Generalization / Specialization
- **Figure 2.3 UML notation for relationships on class diagram**

Associations

- Association is
  - static, structural relationship between classes
  - E.g, Employee <u>works in</u> Department
- Multiplicity of Associations
  - Specifies how many instances of one class may relate to a single instance of another class
- 1-to-1 association (**Figure 8.1)**
  - Company <u>has</u> President
- 1-to-many association (**Figure 8.2)**
  - Bank <u>manages</u> Account
- Optional association (0, 1, or many) (**Figure 8.5)**
  - Customer <u>owns</u> Credit Card
- Many-to-Many association (**Figure 8.6)**
  - Course <u>has</u> Student
  - Student <u>attends</u> Course

Composition and Aggregation Hierarchies

- Whole/Part Relationships
  - Show components of more complex class
  - Composition is stronger relationship than aggregation
- Composition Hierarchy
  - Whole and part objects are created, live, die together
  - Often also has a physical association
  - Association between instances
  - **Figure 8.10 Example of composition hierarchy**
- Aggregation Hierarchy
  - Part objects of aggregate object may be created and deleted independently of aggregate object
  - Often used for more abstract whole/part relationships than composite objects
  - **Figure 8.11 Example of aggregation hierarchy**

Generalization / Specialization Hierarchy

- Some classes are similar but not identical
  - Have some attributes in common, others different
- Common attributes abstracted into generalized class (superclass)
  - E.g., Account (Account number, Balance)
- Different attributes are properties of specialized class (subclass)
  - E.g., Savings Account (Interest)
- IS A relationship between subclass and superclass
  - Savings Account IS A Account
  - **Figure 8.12 Generalization / specialization hierarchy**

Static Modeling of Problem Domain

- During Analysis Modeling
  - Conceptual static model
  - Emphasizes real-world classes in the problem domain
  - Does not initially address software classes
  - Emphasis on
    - Physical classes
      - Have physical characteristics (can see, touch)
    - Entity classes
      - Data intensive classes
- **Figure 8.16 Conceptual static model for Banking System**

System Context Class Diagram

- Defines boundary between system and external environment
  - May be depicted on System Context Class Diagram
- External classes
  - External entities that system interfaces to
- Categories of external classes
  - External I/O device
  - External user
  - External system
  - External timer
- **Figure 8.17 Banking System class context diagram**

Static Modeling of Entity Classes

- Entity classes
  - Data intensive classes
  - Store long-living (persistent) data
  - Especially important for Information Systems
    - Many are database intensive
  - Also important for many real-time and distributed applications
- During analysis modeling
  - Model entity classes in the problem domain
  - Attributes
  - Relationships
  - **Figure 8.18 Conceptual static model for Banking System - entity classes**

Object Structuring

Hassan Gomaa
Dept of Information & Software Engineering
George Mason University

Reference: H. Gomaa, "Chapter 9 - Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison Wesley Object Technology Series, July, 2000

Object-Oriented Software Life Cycle

Requirements & Analysis Modeling

- Requirements Modeling
  - Use Case Modeling
    - Define software functional requirements in terms of use cases and actors
- Analysis Modeling
  - Static Modeling
    - Define structural relationships between classes
    - Depict classes and their relationships on class diagrams
  - Dynamic Modeling
    - Define statecharts for state dependent objects
    - Defines how objects participate in use cases using collaboration diagrams or sequence diagrams

Object Structuring

- Decomposition of problem into objects
  - Based on judgement and characteristics of problem
  - No single correct representation
- Whether objects are in same class or different class depends on nature of problem
- In auto catalog
  - cars, vans, trucks may all be objects in same class
- For vehicle manufacturer
  - cars, vans, trucks may all be objects of different classes

Object Structuring Criteria

- Determine all software objects in system
  - Use Object Structuring Criteria
  - Guidelines for identifying objects
- Structuring criteria depicted using stereotypes (Fig. 9.1)
  - **Stereotype** defines a new building block that is derived from an existing UML modeling element but is tailored to the modeler's problem
  - Depicted using guillemets
    - «entity», «interface», «control»
- Objects are categorized
  - A **category** is a specifically defined division in a system of classification

Object Structuring Criteria

- Interface objects
  - Interface to external environment
  - Each software interface object interfaces to an external (real-world) object (Fig. 9.2)
  - Device interface object
    - Interfaces to I/O device
- Input device interface object
  - E.g., Sensor Interface (Fig. 9.3)
- Output device interface object
  - E.g., Actuator Interface (Fig. 9.4)
- I/O (Input/Output) device interface object
  - E.g., ATM Card Reader Interface (Fig. 9.5)

Object Structuring Criteria

- User interface object (Fig. 9.6)
  - Interfaces to a human user
    - Via standard I/O devices
      - keyboard, visual display, mouse
    - Support simple or complex user interfaces
      - Command line interface
      - Graphical user interface (GUI)
    - Graphical user interface (GUI) object
      - Often a composite object
      - Composed of simpler objects

Object Structuring Criteria

- System interface object
  - Interfaces to an external system
  - Hides details of **how** to communicate with external system
    - E.g., Robot Interface
    - Interfaces to external (real-world) robot
    - Example: Fig. 9.7

Depicting External Classes and Interface Classes

- Start from system context class diagram
  - Shows external classes
  - System (aggregate class)
- Each **external class** must interface to
  - software **interface class**
- Use UML package notation
  - System shown as package
  - External classes are outside the system package
  - Interface classes are inside the system package
  - Example: Fig. 9.8

Object Structuring Criteria

- Entity objects
  - Long lasting objects that store information
    - Same object typically accessed by many use cases
    - Information persists over access by several use cases
      - E.g., Account, Customer
  - Entity classes and relationships shown on static model
  - Entity classes often mapped to relational database during design
  - Examples: Figs. 9.9 – 9.10

Object Structuring Criteria

- Control objects
  - Provides overall coordination for execution of use case
  - Glue that unites other objects that participate in use case
  - Makes overall decisions
  - Decides when, and in what order, other objects participate in use case.
    - Entity objects
    - Interface objects
  - Simple use cases do not need control objects
  - More complex use case usually has at least one control object

Object Structuring Criteria

- Control object
  - Coordinator object
    - Provides sequencing for use case
    - Is not state dependent
    - Example: Fig. 9.11
  - State dependent control object
    - Defined by finite state machine
      - statechart or state transition table
    - Example: Fig. 9.12
  - Timer object
    - Activated periodically
    - Example: Fig. 9.13

Object Structuring Criteria

- Application Logic Objects
  - Business Logic Object
    - Defines business specific application logic (rules) for processing a client request
    - Usually accesses more than one entity object
    - Example: Fig. 9.11
  - Algorithm Object
    - Encapsulates algorithm used in problem domain
    - More usual in scientific, engineering, real-time domains
    - Example: Fig. 9.14

Subsystems

- Subsystem
  - Composite or aggregate object
  - Subsystem may be depicted as package in UML
  - Subsystem dependencies may be depicted
  - For more advanced relationship depict subsystem as aggregate or composite class
  - Examples: Figs. 9.15-9.16
- Object Structuring
  - Only easily identified subsystems are determined
  - Subsystem structuring is addressed in more detail later

Finite State Machines and Statecharts

Hassan Gomaa
Dept of Information & Software Engineering
George Mason University

Reference: H. Gomaa, "Chapter 10 - Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison Wesley Object Technology Series, July, 2000

Object-Oriented Software Life Cycle

Requirements & Analysis Modeling

- Requirements Modeling
  - Use Case Modeling
    - Define software functional requirements in terms of use cases and actors
- Analysis Modeling
  - Static Modeling
    - Define structural relationships between classes
    - Depict classes and their relationships on class diagrams
  - Dynamic Modeling
    - Define statecharts for state dependent objects
    - Defines how objects participate in use cases using collaboration diagrams or sequence diagrams

Finite State Machines

Finite number of states

Only in one state at a time

Transition

Change of state

Caused by event

Transition to same or different state

Action may result from state transition

Notation

State transition diagram

State transition table

Statechart

Examples of statecharts (Figures 10.1 - 10.3)

Finite State Machines and Statecharts

- Statechart
  - Graphical representation of finite state machine
  - States are rounded boxes
  - Transitions are arcs
- Statechart relates events and states
- Event
  - Causes change of state
    - Referred to as state transition
- State
  - A recognizable situation
  - Exists over an interval of time
  - Represents an interval between successive events
- Examples of statecharts (Figures 10.1 - 10.3)

Events

- Event
  - A discrete signal that happens at a point in time
  - Also known as a stimulus
  - Has no duration
- Two events
  - May logically depend on each other
  - E.g, ATM Card inserted before Pin # entered
- Two events
  - May be independent of each other
  - E.g., ATM card read at Alexandria ATM
  - ATM Card read at Fairfax ATM

Events and Conditions

- State transition label
  - Event [Condition]
- Condition is a Boolean function
  - Conditions are optional on statecharts
  - Condition is true for finite period of time
- When event occurs, condition must be *true* for state transition to occur.
- If condition is *false*, state transition does not occur
- Condition may be used to indicate that event has occurred
  - E.g., Closedown Was Requested
- **Figure 10.4 Partial statechart**
- **Figure 10.5 Relationship between events and conditions**
- **Figure 10.6 Use of events and conditions in statechart**
- **Figure 10.7 Example of events and conditions**

Actions

- State transition label
  - Event / action(s)
  - Event [condition] / action(s)
- Action
  - Executed as a result of state transition
  - Executes instantaneously at state transition
  - Terminates itself
  - Is optional
- **Figure 10.8 Example of actions**
- **Figure 10.9 Detailed Cruise Control statechart with actions and conditions**
- Activity
- Entry/exit actions

Activities

- Activity
  - Executes for duration of state
    - Enable Activity on entry to state
    - Disable Activity on exit from state
  - Alternatively
    - do / Activity in state
- Examples of activities
  - Increase Speed
    - Executes for duration of Accelerating state
  - Maintain Speed
    - Executes for duration of Cruising state
  - Resume Cruising
    - Executes for duration of Resuming state
- **Figure 10.10 Cruise Control statechart with activities**

Entry and Exit Actions

- Entry action
  - Action executed on entry into state
    - Entry / action
  - E.g., Display System Down
  - E.g., Display Welcome
  - **Figure 10.11 Example of entry actions**
- Exit action
  - Action executed on exit from state
    - Exit / action
  - E.g, Select Desired Speed
  - **Figure 10.12 Example of exit action**

Hierarchical Statecharts

- Disadvantages of State Transition Diagrams and Flat Statecharts
  - Complex State Transition Diagrams get very cluttered
  - Limited capability for managing complexity
- Hierarchical Statecharts
  - Based on Harel Statecharts
  - Notation for hierarchical decomposition of state transition diagrams
    - Superstate decomposed into substates
    - Default entry states
    - Transition out of superstate corresponds to transition out of every substate

Hierarchical Statecharts

- **OR** decomposition
  - When object is in superstate
    - It is in one and only one of substates
  - Transition into superstate
    - Must be to one and only one of substates
- Aggregation of state transitions
  - If same event causes transition out of every substate
  - Then aggregate into transition out of superstate
- Examples: Fig. 10.14, 19.20-19.23

Hierarchical Statecharts

- Concurrent statecharts
  - State of an object described by more than one statechart
  - Show different aspects of object, may not be concurrent
- Orthogonal statechart
  - Used to depict states of different aspects of object
- **AND** decomposition
  - Object is in one substate on each lower level statechart
  - Object’s state is union of all substates
- Same event
  - May cause transitions on more than one statechart
- Output event on one statechart
  - May be input event on other statechart
- Substate on one statechart
  - May be condition on other statechart
- Example: Fig. 10.15

Guidelines on Statecharts

- State name must be passive not active
  - Represents time period when something
    - is happenING, e.g., Elevator Moving
    - Identifiable situation, e.g., Elevator Idle, Initial
- State names must be unique
- Must be able to exit from every state
- Flat statechart
  - Statechart is only in one state at a time
- Hierarchical statechart
  - **or** decomposition
    - Statechart is only in one substate at a time
  - **and** decomposition
    - Statechart is in one substate on each lower level concurrent statechart

Guidelines on Statecharts

- Event is the cause of the state transition
  - Event happens at a moment in time
  - Event name indicates something has just happened
    - e.g, Up Request, Door Closed
- Action is the result of the state transition
  - Action is a command, e.g., Stop, Close Door
  - Action executes instantaneously
  - Activity executes throughout a given state
- More than one action possible with a state transition
  - No sequential dependency between actions
- Condition is a Boolean value
  - Event [Condition]
  - State transition only occurs if
    - Event happens & Condition is True
  - Condition is True over some interval of time
- Actions, Activities and Conditions are optional

Developing Statechart from Use Case

- Develop state dependent use case
- Start with scenario (one path through use case)
  - Consider sequence of interactions between actor and system
- Consider sequence of external events
  - Input event from external environment
  - Causes state transition to new state
  - Action may result from state transition
  - Activity may be enabled / disabled
- Initially develop flat statechart

Example of Developing Statechart from Use Case

- Cruise Control System
- Control Speed use case
  - Scenario of external events
  - Initial state: INITIAL
    - a) Driver engages cruise control lever in ACCEL position
    - b) Driver releases lever (CRUISE)
    - c) Driver presses brake
    - d) Driver engages lever in RESUME position
- Example: Fig. 10.16, 10.17

Example of Developing Statechart from Use Case

Control Speed Use Case
Actor: Driver

Summary: This use case describes the automated cruise control of the car, given the driver inputs via the cruise control lever, brake, and engine external input devices.

Precondition: Driver has switched on the engine and is operating the car manually.

Description:

This use case is described in terms of a typical scenario consisting of the following sequence of external events:

1. Driver moves the cruise control lever to the ACCEL position and holds the lever in this position. The system initiates automated acceleration so that the car automatically accelerates.
2. Driver releases the cruise control lever in order to cruise at a constant speed. The system stops automatic acceleration and starts maintaining the speed of the car at the cruising speed. The cruising speed is stored for future reference.
3. Driver presses the brake to disable cruise control. The system disables cruise control so that the car is once more under manual operation.
4. Driver moves the cruise control lever to the RESUME position in order to resume cruising. The system initiates acceleration (or deceleration) toward the previously stored cruising speed.
5. When the system detects that the cruising speed has been reached, it stops automatic acceleration (or deceleration) and starts maintaining the speed of the car at the cruising speed.
6. Driver moves the cruise control lever to the OFF position. The system disables cruise control, so that the car is once more under manual operation.
7. The driver stops the car and switches off the engine.

Developing Statechart from Use Case
(continued)

- Consider alternative external events
  - Could result in additional states
  - Could result in additional state transitions
  - Example: Fig. 10.10
- Develop hierarchical statechart
  - States that can be aggregated to form superstate
  - Event causing transition from several states
    - Create superstate with one transition out of superstate
    - Instead of many transitions out of substates
  - Example: Fig. 10.18-10.19
- Develop orthogonal statechart
  - Model different aspects of state dependent object
  - Example: Fig. 10.20

Example of Developing Statechart from Use Case

(Control Speed use case - continued)

Alternatives:

The driver actor interacts with the system, using three external input devices: the cruise control lever, the brake, and the engine. Following are the complete set of input events initiated by the driver actor using these external devices, and the reaction of the system to them:

- The Accel, Cruise, Resume, and Off external events from the cruise control lever. The Accel event causes automated acceleration, providing the brake is not pressed. The Cruise event may only follow an Accel event. The Resume event may only occur after cruising has been disabled and the desired cruising speed has been stored. The Off event always disables cruise control.
- The Brake Pressed and Brake Released external events from the brake. The Brake Pressed event always disables cruise control. Automated vehicle control is not possible as long as the brake is pressed. After the brake is released, automated vehicle control may be enabled.
- The Engine On and Engine Off external events from the engine. The Engine Off event disables any activity in the system.

Postcondition: The car is stationary, with the engine switched off.

Dynamic Modeling

Hassan Gomaa
Dept of Information & Software Engineering
George Mason University

Reference: H. Gomaa, "Chapter 11 - Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison Wesley Object Technology Series, July, 2000

Object-Oriented Software Life Cycle

Requirements & Analysis Modeling

- Requirements Modeling
  - Use Case Modeling
    – Define software functional requirements in terms of use cases and actors
- Analysis Modeling
  - Static Modeling
    – Define structural relationships between classes
    – Depict classes and their relationships on class diagrams
  - Dynamic Modeling
    – Define statecharts for state dependent objects
    – Defines how objects participate in use cases using collaboration diagrams or sequence diagrams

Dynamic Modeling

- Use cases refined in Dynamic Model
  - Show objects participating in each use case
- Determine how objects participate in use case
  - Shows sequence of object interactions in use case
  - Develop collaboration diagram
  - Develop sequence diagram
- Message sequence description
  - Narrative description of sequence of object interactions
- State dependent objects
  - Modeled using statecharts
- Dynamic Analysis
  - Approach to determine how objects interact with each other to support use case

Collaboration Diagrams

- Graphically depicts objects participating in a use case
  - Show objects as boxes
  - Show their message interactions as arrows
  - Number sequence of messages
- Message
  - Message = Event + Attributes
    - E.g, ATM card inserted (Card id, expiration date)
- Collaboration Diagram developed for each use case
  - Some objects only appear on one Collaboration Diagram
  - Some objects appear on several Collaboration Diagrams
- Example: Fig. 11.1

Sequence Diagram

- Shows sequence of object interactions in use case
- Emphasis on messages passed between objects
  - Objects represented by vertical lines
    - Actor is on extreme left of page
  - Messages represented by labeled horizontal arrows
    - Only source and destination of arrow are relevant
    - Message is sent from sending object to receiving object
  - Time increases from top of page to bottom
  - Spacing between messages is not relevant
  - Message sequence numbering is optional
- Example: Fig. 11.2

Message Sequence Numbering

- Aa1.1a
  - [first optional letter sequence][numeric sequence] [second optional letter sequence]
- First optional letter sequence - use case id
- Numeric sequence
  - Message sequence starting with external event
  - A1, A2, A3
- Dewey Classification System
  - A1, A1.1, A1.1.1, A1.2
- Interactive System
  - Whole number for external event
    - A1
  - Decimal number for subsequent internal events
    - A1.1, A1.2
- Second optional letter sequence
  - Concurrent event sequences
    - A3, A3a

Message Sequence Description

- Describes how objects participate in use case
  - Narrative description
  - Corresponds to Collaboration Diagram & Sequence Diagram
- Description corresponds to message sequence numbering on diagrams
  - Describe what object does on receiving message
  - E.g, every time the use case references an entity object
    - Describe the object being accessed
    - Identify attributes referenced
- Example: Fig. 11.3

Dynamic Analysis

- Determine how objects interact with each other to support use case
  - Start with external event from actor
  - Determine objects needed to support use case
  - Determine sequence of internal events following external event
  - Depict on collaboration diagram
- Non-state dependent Dynamic Analysis
- State dependent Dynamic Analysis
  - Controlled by statechart
  - Executed by control object
  - Control object activates/deactivates other objects

Non-State Dependent Dynamic Analysis

- Start with black box use case
- Determine interface objects
  - Receives external events from actor
- Determine internal objects
  - Receive messages from interface objects
- Determine object collaboration
  - Sequence of messages passed
- Develop alternative branches
  - E.g, for error handling or less frequently occurring conditions
- Example: Fig. 11.4

State Dependent Dynamic Analysis

- For each black box use case
  - Determine objects participating in use case
    - Determine interface objects
    - Determine internal objects
  - Determine object collaboration
  - Develop statechart(s)
    - Control object executes statechart
  - Iterate till object collaboration diagram consistent with statechart

State Dependent Dynamic Analysis

- For each event that arrives at control object
  - Determine state transition from current state to next state
- For each state transition
  - Determine actions that result from change in state
  - Determine activities to be executed in new state
  - Determine objects required to perform actions and activities
- Develop alternative branches
- Complete analysis use case
  - Must enter every state
  - Must execute every state transition
  - Perform each action and activity

Example of Dynamic Analysis Banking System - Validate PIN Use Case

- Figs. 11.3, 11.5 – 11.11 , 19.12- 19.14
- Client Objects
  - Interface Objects
    - Card Reader Interface
    - Customer Interface
  - Entity Objects
    - ATM Card
    - ATM Transaction
  - State Dependent Control Object
    - ATM Control

Example of Dynamic Analysis
Banking System - Validate PIN Use Case

- Server Objects
  - Entity Objects
    - Debit Card
    - Card Account
  - Business Logic Objects
    - PIN Validation Transaction Manager
- Example: Fig. 19.15

Example of Dynamic Analysis
Banking System - Withdraw Funds Use Case

- Client Objects
  - Interface Objects
    - Card Reader Interface
    - Receipt Printer Interface
    - Cash Dispenser Interface
    - Customer Interface
  - Entity Objects
    - ATM Transaction
    - ATM Cash
  - State Dependent Control Object
    - ATM Control
- Figs. 19.16- 19.18

Example of Dynamic Analysis
Banking System - Withdraw Funds Use Case

- Server objects
  - Entity Objects
    - Checking Account
    - *or* Savings Account
    - Debit Card
    - Transaction Log
  - Business Logic Objects
    - Withdrawal Transaction Manager
- Fig. 19.19

Example of Dynamic Analysis
Cruise Control System - Control Speed Use Case

- Example from Cruise Control System
- Initial Object Determination
  - State Dependent Control object
    - Cruise Control
  - Input device interface objects
    - Cruise Control Lever Interface, Engine Interface, Brake Interface
  - Output device interface object
    - Throttle Interface

Example of Dynamic Analysis
Cruise Control System - Control Speed Use Case

- Scenario of external events
  - Initial state: INITIAL
    - a) Driver engages cruise control lever in ACCEL position
    - b) Driver releases lever (CRUISE)
    - c) Driver presses brake
    - d) Driver engages lever in RESUME position
- Example: Fig. 11.12 – 11.22

Object-Oriented Software Life Cycle
Requirements & Analysis Modeling

- Requirements Modeling
  - Use Case Modeling
    - Define software functional requirements in terms of use cases and actors
- Analysis Modeling
  - Static Modeling
    - Define structural relationships between classes
    - Depict classes and their relationships on class diagrams
  - Dynamic Modeling
    - Define statecharts for state dependent objects
    - Defines how objects participate in use cases using collaboration diagrams or sequence diagrams

Concurrent Object-Oriented Design with UML

Hassan Gomaa
Dept of Information & Software Engineering
George Mason University

Reference: H. Gomaa, "Chapter 6 - Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison Wesley Object Technology Series, July, 2000

Concurrent Object-Oriented Design Method

- Introduction to Method
- Software Architecture Design
- Task Structuring
- Information Hiding Class Design
- Detailed Software Design
- Relational Database Design

Overview

- Concurrent Object Modeling Technology (COMET)
  - Object Oriented Analysis and Design Method
  - Uses UML notation
- Provide criteria for structuring concurrent, real-time, distributed applications
- Provides steps and procedures for mapping
  - From object-oriented analysis model
  - To a concurrent object-oriented design
- See Fig. 6.1

Object-Oriented Software Life Cycle

Requirements & Analysis Modeling

- Requirements Modeling
  - Use Case Modeling
    - Define software functional requirements in terms of use cases and actors
- Analysis Modeling
  - Static Modeling
    - Define structural relationships between classes
    - Depict classes and their relationships on class diagrams
  - Dynamic Modeling
    - Define statecharts for state dependent objects
    - Defines how objects participate in use cases using collaboration diagrams or sequence diagrams

Object-Oriented Software Life Cycle

Architectural Design

- Maps analysis model (emphasis on problem domain) to design model (emphasis on solution domain)
- Structure system into subsystems
- Design each subsystem
- Sequential Applications
  - Emphasis on OO concepts
    - Information hiding, classes, inheritance
- Concurrent, Distributed and Real-Time Applications
  - Emphasis on
    - OO concepts
    - Concurrent tasking

COMET OO Analysis and Design

- UML Notation
  - Supports both Analysis and Design concepts
- COMET/UML method
  - Separate requirements activities, analysis activities and design activities
- Requirements Modeling
  - Consider system as black box
  - Develop Use Case Model

COMET OO Analysis and Design

- Analysis modeling
  - Consider analysis of problem domain
  - Determine problem oriented objects and classes
  - Analyze static viewpoint in Static Model
    - Classes, relationships, attributes
  - Analyze dynamic viewpoint in Dynamic Model
    - Statecharts
    - Object interaction model
      - Consider objects supporting each use case
      - Analyze sequence of interactions between objects
      - Analyze information passed between objects

COMET OO Analysis and Design

- Design Model
  - Consider solution domain
  - Make decisions about overall software architecture
  - Make decisions about distributed component-based subsystems
  - Make decisions about characteristics of objects
    - Active or Passive
  - Make decisions about characteristics of messages
    - Asynchronous or Synchronous (with/without reply)
  - Make decisions about class interfaces
    - Operations and parameters
  - Make detailed design decisions

Steps in Using COMET/UML

1 Develop Object-Oriented Requirements Model
   – Develop Use Case Model (Chapter 7)
2 Develop Object-Oriented Analysis Model
   – Develop static model of problem domain (Chapter 8)
   – Structure system into objects (Chapter 9)
   – Develop statecharts for state dependent objects (Chapter 10)
   – Develop object interaction diagrams for each use case (Chapter 11)
3 Develop Object-Oriented Design Model
   – Design Overall Software Architecture (Chapter 12)
   – Design Distributed Component-based Subsystems (Chapter 13)
   – Structure Subsystems into Concurrent Tasks (Chapter 14)
   – Design Information Hiding Classes (Chapter 15)
   – Develop Detailed Software Design (Chapter 16)

Software Architecture Design

Hassan Gomaa
Dept of Information & Software Engineering
George Mason University

Reference: H. Gomaa, "Chapter 12 - Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison Wesley Object Technology Series, July, 2000

Transition from Analysis to Design:
Consolidation of Collaboration Diagrams

- Used to determine overall structure of system
- Merger of Collaboration Diagrams
  - Start with first Collaboration Diagram
  - Superimpose other Collaboration Diagrams
    - Add new objects and new message interactions from each subsequent diagram
    - Objects and interactions that appear on multiple diagrams are only shown once
    - Consider alternative scenarios for each use case
- Consolidated Collaboration Diagram
  - Shows all objects and their interactions
  - Example: Fig. 12.5

Consolidation of Collaboration Diagrams

- Subsystem collaboration diagram
  - High-level collaboration diagram
  - Shows subsystems and their interactions
  - Example: Fig. 12.6
- Consolidated collaboration diagram
  - If there are too many objects for one consolidated collaboration diagram
  - Develop subsystem collaboration diagram
  - Develop consolidated collaboration diagram for each subsystem

Design of Software Architecture

- Software Architecture
  - Define overall structure of system
    - Component interfaces and interconnections
  - Separately from component internals
- Each subsystem performs major service
  - Contains highly coupled objects
  - Relatively independent of other subsystems
  - May be decomposed further into smaller subsystems
  - Subsystem is aggregate or composite object
- Candidates for subsystem
  - Objects that participate in same use case

Separation of Subsystem Concerns

- **Aggregate/composite object**.
  - Objects that are part of aggregate/composite object
  - Structure in same subsystem (e.g., Fig. 12.7)
- **Interface to external objects**
  - External real-world object should interface to 1 subsystem (e.g., Fig. 12.4)
- **Scope of Control**
  - Control object & objects it controls are in same subsystem (e.g., Fig. 12.8)
- **Geographical location**
  - Objects at different locations are in separate subsystems (e.g., Fig. 12.4)
- **Clients and Servers**
  - Place in separate subsystems (e.g., Fig. 12.1)
- **User Interface**
  - Separate client subsystem (e.g., Fig. 12.9)

Subsystem Structuring Criteria

- Control
  - Subsystem controls given aspect of system (e.g., Fig. 12.8)
- Coordination
  - Coordinates several control subsystems (e.g., Fig. 12.4)
- Data Collection
  - Collects data from external environment (e.g., Fig. 12.9)
- Data analysis
  - Provides reports and/or displays (e.g., Fig. 12.9)
- Server
  - Provides service for client subsystems (e.g., Fig. 12.9, 12.10)
- User Interface
  - Collection of objects supporting needs of user (e.g., Fig. 12.9)

Static Modeling at Design Level

- Design Modeling
  - More detailed static model is developed
  - Start from conceptual static model if there is sufficient detail (e.g., Fig. 12.11)
  - Otherwise, start from consolidated collaboration diagrams (e.g., Fig. 12.13)
    - Design class for each object
    - Design relationship for each link between objects
    - Show direction of navigability on class diagram
    - Generalization/specialization relationships are only shown on class diagram
  - Can combine the two approaches
  - Example: Fig. 12.11 – 12.12

Architectural Design of Distributed Applications

Hassan Gomaa
Dept of Information & Software Engineering
George Mason University

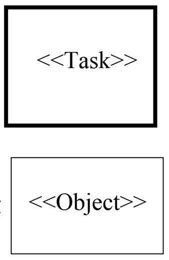
Reference: H. Gomaa, “Chapter 13 - Designing Concurrent, Distributed, and Real-Time Applications with UML”, Addison Wesley Object Technology Series, July, 2000

Architectural Design of Distributed Applications

- Distributed processing environment
  – Multiple computers communicating over network
- Typical applications
  – Distributed real-time data collection
  – Distributed real-time control
  – Client / server applications
- COMET/UML for Distributed Applications
  – Addresses structuring application into distributed subsystems
  – Examples: Figs. 4.6 – 4.8

Active and Passive Objects

- Objects may be **active** or **passive**
- **Active object**
  - **Concurrent Task**
  - Has thread of control
- **Passive object**
  - a.ka. **Information Hiding Object**
  - Has no thread of control
  - Operations of passive object are executed by task
  - Operations execute in task's thread of control
    - Directly or indirectly
- Software Design terminology
  - **Task** refers to active object
  - **Object** refers to passive object



Sequential and Concurrent Systems

- Sequential Systems
  - Sequential program
  - One thread of control (task)
  - Several passive objects
  - Synchronous communication
    - Operation (procedure or function) call
- Concurrent Systems
  - Several active objects (tasks)
  - Each task has its own thread of control
    - Asynchronous communication
    - Synchronous communication

Characteristics of Distributed Applications

- Structure of distributed application
  - Consists of one or more subsystems
  - Execute on multiple nodes in distributed configuration
  - Subsystems determined using subsystem structuring criteria
- Structure of Subsystem
  - Consists of one or more objects
  - Objects all execute on same node
- Communication between subsystems
  - Message communication
  - Example: Fig. 4.14

Steps in Designing Distributed Applications

- System Decomposition
  - Decompose system into distributed subsystems
    - Design as configurable components
  - Define message communication interfaces
  - Examples: Figs. 13.1 – 13.3
- Subsystem Decomposition
  - Structure subsystem into active objects (tasks) and passive objects
- System Configuration
  - Define component subsystem instances of target system
  - Map to hardware configuration
  - Example: Fig. 13.14

Define Subsystem Interfaces

- Message Communication between distributed subsystems
  - Loosely coupled (asynchronous) message communication
  - Multiple Client / Server message communication
    - Tightly coupled (synchronous) message communication
    - Remote Procedure Call
  - Group Message Communication
    - Broadcast message communication
    - Multicast message communication
  - Transaction management
  - Object Broker Architecture

Loosely Coupled Message Communication (Asynchronous)

Producer

    Sends message and continues

    Message queue may build up

Consumer receives message

    Suspended if no message present

    Activated when message arrives

        Processes message

    Not suspended if message present

    Example: Fig. 13.2

Client / Server Message Communication

- Server
  - Often encapsulates data store(s)
  - Does not initiate any message communication
  - Responds to message requests from clients
  - Usually services many clients
- Client
  - Sends message to server
  - Usually has to wait for response
- Tightly Coupled Message Communication
  - Remote Procedure Call
- Examples: Figs. 13.1, 4.15, 4.16

Group Message Communication

- One-to-many message communication
  - Same message sent to several recipients
- Broadcast message communication
  - Message sent to all recipients
- Multicast message communication
  - Same message sent to all members of group
- Subscription/Notification communication
  - Client subscribes to group
  - Receives messages sent to all members of group
  - Sender sends message to group
    - Does not need to know recipients
  - Example: Figs. 13.4

Distributed Objects and Object broker

- Clients and Servers designed as distributed objects
- Object Broker
  - Mediates interactions between clients and servers
  - Frees client from having to maintain information
    - Where particular service provided
    - How to obtain service
- Servers register Services & Location with Broker
- Clients request information from Broker about Servers
- Broker provides different services
- Example: CORBA, Fig. 4.17

Object Broker Architecture

- White pages
  - Client knows name of service but not location
  - Forwarding design (Fig. 13.5)
    - Broker forwards client request to Server
    - Broker forwards Server response to Client
  - Handle-driven design (Fig. 13.6)
    - Broker returns handle (remote reference) to Client
    - Client uses handle to communicate with Server
- Yellow pages
  - Client knows service type but not specific server
  - Client makes yellow pages query (Fig. 13.7)
    - Request all services of a given type
  - Client selects service, then makes white pages query

Task Structuring

Hassan Gomaa
Dept of Information & Software Engineering
George Mason University

Reference: H. Gomaa, “Chapter 14 - Designing Concurrent, Distributed, and Real-Time Applications with UML”, Addison Wesley Object Technology Series, July, 2000

Structure System into Tasks

- Concurrent Design with UML
- Concurrent task structuring criteria
  - Structure analysis model into concurrent tasks
    - Task is an active object
    - Task has thread of control
  - Consider concurrent nature of system activities
  - Determine concurrent tasks
- Define task interfaces
- Develop concurrent collaboration diagrams
- Develop task behavior specifications

Operating System Support for Concurrent Tasks

Multi-tasking Kernel (nucleus, executive)

Services

- Task creation and deletion
- Priority pre-emption task scheduling
- Inter-task synchronization using events
- Mutual exclusion using semaphores
- Inter-task communication using messages

Examples

- Unix, VRTX, Windows/NT, Linux

Language Support for Concurrent Tasks

Multi-tasking capabilities

- Concurrent tasking constructs
- Task creation and deletion
- Support for inter-task communication and synchronization
- Run time system handles scheduling and dispatching of tasks

Examples

- Ada
- Java

Task Structuring - Task Structuring Categories

- I/O task structuring criteria
  - How device interface objects are mapped to I/O tasks
- Internal task structuring criteria
  - How internal objects are mapped to internal tasks
- Task priority criteria
  - Importance of executing task relative to others
- Task clustering criteria
  - Whether and how objects should be combined into concurrent tasks

I/O Task Structuring Criteria

- Asynchronous I/O device interface task
  - Task for each asynchronous I/O device
  - Asynchronous device generates interrupt
  - Example: Fig. 14.1
- Periodic I/O device interface task
  - Task for each polled I/O device
  - I/O device (usually input) sampled at regular intervals
  - Example: Fig. 14.2
- Passive I/O device interface tasks
  - Task for each passive I/O device (usually output)
  - Computation overlapped with output
  - Example: Fig. 14.3
- Resource Monitor task
  - Task for each I/O device that receives requests from multiple sources
  - Example: Fig. 14.4

Internal Task Structuring Criteria

- Periodic task
  - Task for each periodic activity
  - Example: Fig. 14.5
- Asynchronous task
  - Task for each asynchronous internal activity
  - Example: Fig. 14.6
- Control task
  - Task executes statechart
  - Example: Fig. 14.7, 14.8
- User interface task
  - Task for each sequential user activity
  - Example: Fig. 14.9

Task Priority Criteria

- Important consideration
  - Performance Analysis
  - Real-Time Scheduling
    - Rate Monotonic Analysis
- Time critical
  - Activity that has hard deadline
  - Map to time critical task
  - Example: Fig. 14.1
- Non-time-critical computationally intensive
  - Low priority activity
  - Example: Fig. 14.3

Task Clustering Criteria

- Temporal clustering
  - Activities activated by same event
  - Example: Fig. 14.10
- Sequential clustering
  - Activities must be executed sequentially
  - Example: Fig. 14.11
- Control clustering
  - Control object grouped with objects it activates
  - Example: Fig. 14.12
- Mutually exclusive clustering
  - Activities cannot execute concurrently
  - Example: Fig. 14.13
- Task Inversion
  - Map all objects of same type to one task
  - Example: Fig. 14.14

Define Task Interfaces

- Map Analysis Model Interaction Diagram interfaces to task interfaces
- Simple messages with data transfer between concurrent tasks
  - Need to determine type of message communication
- Simple messages without data transfer between concurrent tasks (synchronization only)
  - Event synchronization
  - Message communication
- Passive objects
  - Information hiding objects
- Update task architecture on concurrent collaboration diagrams

Task Interfaces

- Producer task sends data to consumer task
  - Loosely coupled message communication
  - Tightly coupled message communication
    - With reply
    - Without reply
- Event synchronization
- Task interface to information hiding object

Loosely Coupled Message Communication (Asynchronous)

- Producer
  - Sends message and continues
  - Message queue may build up
- Consumer receives message
  - Suspended if no message present
  - Activated when message arrives
    - Processes message
  - Not suspended if message present

  Example: Fig. 14.18

Tightly Coupled (Synchronous) Message Communication With Reply

- Producer
  - Sends message
  - Waits for reply
- Consumer
  - Suspended if no message present
  - Activated when message arrives
    - Accepts message
    - Generates and sends reply
- Producer and Consumer continue
- Example: Fig. 14.19

Tightly Coupled (Synchronous) Message Communication Without Reply

- Producer
  - Sends message
  - Waits for acceptance by Consumer
- Consumer
  - Suspended if no message present
  - Activated when message arrives
    - Accepts message
    - Releases producer
- Producer and Consumer continue
- Example: Fig. 14.20

Information Hiding Object

- Passive object
- Encapsulates data store
  - Hides contents of data store
  - Data store accessed indirectly via operations
    - Access procedures
    - Access functions
- Data store
  - Accessed by two or more tasks
  - Access procedures/functions
    - Must synchronize access to data
  - Example: Fig. 14.24

Event Synchronization

Types of events

External event (interrupt) – Fig. 14.21

Timer event – Fig. 14.22

Internal event – Fig. 14.23

Two tasks may need to synchronize their operations

If message contains no data, can use internal event

Source task signals event

Signal (Event)

Destination task waits for event

Wait (Event)

Suspended until event signaled

Task Behavior Specifications (TBS)

TBS evolves as task design is refined

First developed during Task Structuring

Refined during Detailed Software Design

Describes concurrent task's

- Interface
- Structure
- Timing characteristics
- Relative priority
- Event sequencing logic

Task Behavior Specification

Task interface

- Message communication
  - Type of interface
  - Message names and parameters
- Events signaled
  - Name and Type of event
- External inputs or outputs

Task structure information

- Task structuring criterion used to design task

Task's event sequencing logic

- Response to each message or event input
- Described informally in Pseudocode

Class Design

Hassan Gomaa
Dept of Information & Software Engineering
George Mason University

Reference: H. Gomaa, “Chapter 15 - Designing Concurrent, Distributed, and Real-Time Applications with UML”, Addison Wesley Object Technology Series, July, 2000

Design Information Hiding Classes

- Design information hiding classes
  - Design of passive classes
- Information Hiding Classes
  - Initially determined from Analysis Model
- Design class operations
- Develop class hierarchies using inheritance

Information Hiding Class Structuring

- Classes derived from analysis model
  - Interface classes
    - User interface classes
    - Device interface classes
    - System interface classes
  - Entity classes are categorized further
    - Data abstraction classes
    - Database wrapper classes
  - Control classes
  - Application Logic classes
- Software decision classes

Design Class Operations

- Design Class Operations from Collaboration Model
  - Shows direction of message from sender object to receiver object
- Design Class Operations from Finite State Machine Model
  - Statechart actions are mapped to operations
- Design Class Operations from Static Model
  - May be used for entity classes
  - Standard operations
    - Create, Read, Update, Delete
  - Specific operations
    - Based on services provided by class

Information Hiding Class Categorization

- Categorize Information Hiding Classes
  - By stereotype
- Information Hiding Classes Categories
  - Determined from analysis model
    - Interface classes
    - Entity classes
    - Control classes
    - Application Logic classes
  - Determined in later design stages
    - Software decision classes

Class Design

– Data Abstraction Class
  - Hide internal structure and content of data structure
  - Examples: Fig. 15.1 – 15.2
- Device Interface Class
  - Hides actual interface to real world device
  - Supports virtual interface via operations
  - Examples: Fig. 15.3 - 15.4

– State Dependent Control Class
  - Hides contents of statechart / state transition table
  - Example: Fig. 15.5

Class Design

- Algorithm Hiding Class
  - Hides algorithm used in application domain
  - Example: Fig. 15.6
- User Interface Class
  - Hides interface to user
  - Example: Fig. 15.7
- Business Logic Class
  - Hides business application logic (rules)
  - Example: Fig. 15.8
- Database Wrapper Class
  - Hides interface to database
  - Example: Fig. 15.9

Class Design

- Software Decision Classes
  - Determined in later design stages
  - Hide design decisions that are likely to change
  - Examples (see Chapter 16)
    - Encapsulated data structures
      - Stacks
      - Queues
      - Data Tables

Inheritance in Design

- Subclass inherits generalized properties from superclass
    - Property is Attribute or Operation
- Inheritance
    - Allows sharing of properties between classes
    - Allows adaptation of parent class (superclass) to form child class (subclass)
- Subclass inherits attributes & operations from superclass
    - May add attributes
    - May add operations
    - May redefine operations

Example of Inheritance

- Attributes of *Account* Superclass
    - *accountNumber, balance*
- Operations of *Account* Superclass
    - *open (accountNumber : Integer)*
    - *close ()*
    - *readBalance () : Real*
    - *credit (amount : Real)*
    - *debit (amount : Real)*
- Example: Fig. 15.10

Example of Inheritance

- Attributes of *Checking Account* Subclass
  - Inherits *accountNumber, balance*
  - Adds *lastDepositAmount*
- Operations of *Checking Account* Subclass
  - Inherits specification and implementation of *open, readBalance, debit, close*
  - Inherits specification of *credit* but redefines implementation
  - Adds *readLastDepositAmount () : Real*

Example of Inheritance

- Attributes of *Savings Account* Subclass
  - Inherits *accountNumber, balance*
  - Adds instance attributes *cumulativeInterest, debitCount*
  - Adds static class attributes *maxFreeDebits, bankCharge*
- Operations of *Savings Account* Subclass
  - Inherits *open, readBalance , credit, close*
  - Modifies *debit*
    - Debit balance and deduct *bank Charge* if *debit Count* > *max Free Debits*
  - Adds Operations
    - *addInterest (interestRate)* Add daily interest
    - *readCumulativeInterest () : Real*
    - *clearDebitCount ()* Reinitialize debit Count to zero

Polymorphism

- Polymorphism
  - Greek for “many forms”
- Different classes may have same operation name
- Name of operation is shared among several classes
  - Specification of operation is identical for each class
  - Each class can implement operation differently
  - E.g. operation *debit* is implemented differently for checking accounts and savings accounts

Dynamic Binding

- Run-time binding between variable and objects it references
- Variable may reference objects of different classes at different times, e.g. (page 391):

```
Prompt customer for account number & account type
If customer responds checking
  Then anAccount := customerCheckingAccount
  Elseif customer responds savings
     Then anAccount := customerSavingsAccount
     …
Endif….
anAccount.debit (amount)
```

Abstract Class

- Abstract Class
  - Template for creating subclasses
  - Has no instances
  - Only used as superclass
  - Defines common interface for subclasses
- Abstract operation
  - Operation declared in abstract class but not implemented
- Abstract Class defers implementation of some or all of its operations to subclasses
- Different subclasses can define different implementations of same abstract operation

Example of Inheritance with Abstract Class (Fig. 15.11)

Abstract Superclass

  Maintenance

Encapsulated data type

  Initial Mileage

Operations

  Reset

    Interface and implementation defined

    Sets Initial Mileage

  Check

    Interface defined

    Implementation deferred to subclasses

Example of Inheritance with Abstract Class

(Fig. 15.11)

Subclasses

- Oil Change Maintenance
- Air Filter Maintenance
- Major Service Maintenance

Each subclass inherits from Maintenance Superclass

- Type definition of Initial Mileage
- Full definition (interface & implementation) of Reset
- Specification of Check

Each subclass adds

- Its own implementation of Check

Class Interface Specification

- Information hidden by class
- Class structuring criterion
- Assumptions made in specifying class
- Anticipated changes
- Superclass (if applicable)
- Inherited Operations (if applicable)
- Operations provided by class
  - Function performed
  - Precondition
  - Postcondition
  - Invariant
  - Input parameters
  - Output parameters
  - Operations used by class (provided by other classes)
- Example: Fig. 15.12, pages 395-396

Detailed Software Design

Hassan Gomaa
Dept of Information & Software Engineering
George Mason University

Reference: H. Gomaa, “Chapter 16 - Designing Concurrent, Distributed, and Real-Time Applications with UML”, Addison Wesley Object Technology Series, July, 2000

Detailed Software Design

- Detailed Design of composite tasks
    - Active objects that contain nested passive objects
- Design details of task synchronization
    - Passive objects accessed by more than one task
- Design connector classes
    - Address details of inter-task communication
- Define each task’s internal event sequencing logic
    - Pseudocode description

Example of Design of Composite Task Containing Passive Objects

- Temporal clustering and device interface objects
  - Temporal clustering task
    - Polled I/O
    - Activated periodically
    - Two passive devices
  - Information hiding objects
    - Device Interface Objects
    - Hide details of how to read from device
  - Operations executed in thread of control of task
  - Example: Fig. 16.1

Example of Design of Composite Task Containing Passive Objects

- Control clustering task and passive objects
  - Control clustering task
  - Information hiding objects
    - State dependent object
    - Device Interface Objects
  - Operations executed in thread of control of control task
  - Example: Fig. 16.2
- Concurrent access to classes
  - Classes inside task
  - Classes outside task

Synchronization of Tasks Interacting via Passive Objects

Task interaction via shared data
- Needs synchronization

Task interaction via passive data abstraction object
- Hides structure of data repository
- Hides synchronization from tasks
  - Mutual exclusion
  - Multiple readers / multiple writers

Information Hiding Objects Synchronization of Access

- Each information hiding object
  - Designed for application
  - Example: Fig. 16.3
- Mutually exclusive access to data repository
  - Use binary semaphore
  - Example: Page 408
- Access by multiple readers / writers
  - Allows access to data repository
    - By many readers concurrently
    - Only one writer
  - Example: Page 408 - 410

Interaction Between Concurrent Tasks

- Mutual exclusion
  - Two or more tasks need to access shared data
  - Access must be mutually exclusive
- Binary semaphore
  - Boolean variable that is only accessed by means of two atomic (indivisible) operations
  - **acquire (semaphore)**
    - if the resource is available, then get the resource
    - if resource is unavailable, wait for resource to become available
  - **release (semaphore)**
    - signals that resource is now available
    - if another task is waiting for the resource, it will now acquire the resource

Connector Classes

- Classes designed to provide inter-task communication and synchronization
- Message buffering monitor classes
  - Synchronized (mutually exclusive) operations
- Loosely coupled message communication
  - Use message queue monitor class
  - Encapsulates message queue
  - Example: Fig. 16.4, Pages 414 - 416
- Tightly coupled message communication without reply
  - Encapsulates a message buffer
  - Holds at most one message
  - Example: Fig. 16.5, Pages 416 - 417
- Tightly coupled message communication with reply
  - Encapsulates a message buffer - Holds one message
  - Encapsulates a response buffer - Holds one response
  - Example: Fig. 16.6, Pages 417 – 418
- Design of cooperating tasks using connectors
  - Example: Fig. 16.7, Pages 418 - 419

Banking System Case Study - Task Structuring Criteria

- Asynchronous I/O Device Interface Task
  - Card Reader Interface
- User Interface Task
  - Customer Interface
  - Operator Interface
- Control Clustering Task
  - ATM Controller
- Sequential Clustering
  - Bank Server
- Reference: Chapter 19, Figs. 19.28 – 19.30

ATM Client Subsystem - Information Hiding Class Categorization

- Device Interface Classes
  - Card Reader DI
  - Receipt Printer Interface
  - Cash Dispenser Interface
- User Interface Classes
  - Customer UI
  - Operator UI
- Data Abstraction Classes
  - ATM Card
  - ATM Transaction
  - ATM Cash
- State Dependent Class
  - ATM Control
- Connector classes
- Reference: Chapter 19, Figs. 19.31 – 19.32, 19.33

Bank Server Subsystem - Information Hiding Class Categorization

- Business Logic Classes
  - PIN Validation Transaction Manager
  - Query Transaction Manager
  - Transfer Transaction Manager
  - Withdrawal Transaction Manager
- Database Wrapper Classes
  - Checking Account
  - Savings Account
  - Debit Card
  - Card Account
  - Transaction Log
- Reference: Chapter 19, Figs. 19.35 – 19.36, 19.37, 19.38

Introduction to Architecture and Design Patterns

Hassan Gomaa
Dept of Information & Software Engineering
George Mason University

Reference: H. Gomaa, "Chapters 3, 12, 13 - Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison Wesley Object Technology Series, July, 2000

What is a Pattern?

- Pattern
  - Describes a recurring design problem
  - Arises in specific design contexts (I.e., situations)
  - Presents a well proven approach for its solution
- Micro-architecture (Gamma et al.)
  - Small number of collaborating objects that may be reused
- Design New Software Architectures using existing patterns

Pattern Categories

- Design Patterns
  - Small group of collaborating objects
  - Gang of Four (Gamma, Helms, Johnson, Vlissides)
- Architecture Patterns
  - Address the structure of major subsystems of a system
  - Buschmann, etc. at Siemens
- Analysis Patterns
  - Recurring patterns found in Analysis
  - Fowler
- Domain Specific Patterns
  - Used in a specific application area (e.g., factory automation, Internet terminal)

Software Architecture Patterns

- Also called Software Architectural Styles
  - Recurring architectures used in various software applications
- Client/Server Architecture pattern (Fig. 12.1)
  - **Client** requests services
  - **Server** is provider of services
- Layers of Abstraction pattern (Figs. 12.2, 2.3)
  - Hierarchical architecture
  - Each layer provides services for layers above it
  - Operating systems, network communications software
- Communicating tasks pattern (Fig. 12.4)
  - Each task has its own thread of control
  - Tasks communicate with each other
  - May be centralized or distributed task design

Documenting a Design Pattern

- What a pattern must include (Buschmann)
  - Context
    - Situation leading to problem
  - Problem
    - Problem that often occurs in this context
  - Solution
    - Proven resolution to Problem

What Does a Pattern Include?

- Pattern describes
  - Pattern Name
  - Alias
  - Context
    - When should pattern be used
  - Problem
  - Summary of Solution
  - Strengths of solution
  - Weaknesses of solution
  - Applicability
    - When can you use the pattern
  - Related Patterns

Patterns for Concurrent and Distributed Applications

- Architecture Patterns
  - Based on COMET Method
  - Relate to interactions between distributed subsystems
- Design Patterns
  - Based on COMET Method
  - Task Structuring
  - Task Communication
  - Information Hiding Class Structuring
  - Task / Information Hiding Class Integration
- Examples given using UML notation

Task Communication Design Patterns from COMET

- Task Communication
  - Different ways tasks interact
- List of Communication Patterns
  - FIFO Queue
  - Priority Queue
  - Bi-directional Queue
  - Synchronous with Reply
  - Synchronous without Reply
  - Event Synchronization
  - Bi-directional events

FIFO Queue Pattern -

Alias: Loosely Coupled Message Communication
Alias: Asynchronous Communication
(Fig. 3.12)

Producer

Sends message and continues

Message queue may build up

Consumer receives message

Suspended if no message present

Activated when message arrives

Processes message

Not suspended if message present

Loosely Coupled Message Communication Pattern

- **Pattern Name**: Loosely coupled message communication.
- **Alias**: Asynchronous Communication, FIFO Queue.
- **Context**: Concurrent systems.
- **Problem**: Concurrent application with concurrent tasks that need to communicate with each other. Producer does not need to wait for consumer. Producer does not need reply.
- **Summary of solution**: Use message queue between producer task and consumer task. Producer sends message to Consumer and continues. Consumer receives message. Messages may be queued FIFO (first-in-first-out) if Consumer is busy. Consumer is suspended if no message is available.
- **Strengths**: Consumer does not hold up Producer.
- **Weaknesses**: If Producer produces messages more quickly than Consumer can consume them, the message queue will eventually overflow.
- **Applicability**: Centralized and distributed environments: Real-time systems, client/server and distribution applications.
- **Related Patterns**: Tightly coupled message communication with/without reply.

Architecture Patterns

- Architecture Patterns
  - Based on COMET Method
  - Relate to interactions between distributed subsystems
- List of Patterns
  - Distributed Queue
  - Remote Procedure Call
  - Broker Forwarding
  - Broker Handle
  - Broadcast
  - Multicast

Broker Forwarding Pattern (Fig. 13.5)

- Object Broker Architecture
  - Client can query Object Broker for services provided
- Client sends message to Server via Object Broker
  - Identifies Server name and service required
  - Object Broker
    - Receives client request
    - Determines location of Server
    - Forwards message to Server at specific location
      - Invokes Service at Server
    - Receives Server response
    - Forwards response back to Client

Relational Database Design

Hassan Gomaa
Dept of Information & Software Engineering
George Mason University

Relational Database Design

- Objective: Map static model to relational database
- Each entity class from static model that needs to be stored in relational database
  - Maps to one (or more) relations (table)
  - Each object instance maps to a row of table
- Relational Database Design
  - Primary keys
  - Foreign keys for associations
  - Association classes
  - Aggregation/Composition Hierarchy
  - Generalization/Specialization hierarchy
- Reference: J. Rumbaugh et al, "Object-Oriented Modeling and Design", Prentice Hall, 1991

Primary Keys

- Each relation must have a primary key
- Primary Key
  - Combination of one or more attributes
  - Uniquely locates a row in relation
    - E.g., Account Number is primary key of Account relation
    - Account (<u>Account number</u>, Balance)
      - (underline = <u>primary key</u>)

Relational Database Design
Foreign Keys

- Associations in relational databases
  - Many-to-many association in static model
    - Maps to a relation
  - One-to-one and one-to-many associations
    - Use Foreign keys
- Foreign key
  - Primary key of one table that is embedded in another table
  - Represents mapping of association between relations into a table
  - Allows navigation between tables

One-to-one or Zero-or-one Association

- One-to-one association maps to
  - Foreign key in one of relations
- Zero-or-one association maps to
  - Foreign key in optional relation
- E.g, Customer <u>Owns</u> Debit Card
- Static model
  - Customer (Customer Name, Customer SSN, Customer Address)
  - Debit Card (Card Id, PIN, Expiration date, Status, Limit, Total)

One-to-one or Zero-or-one Association

- Relational Database Design
  - Card id chosen as primary key of **Debit Card** relation

- Relational Database Design
  - <u>Customer SSN</u> chosen as primary key of **Customer**
    - Customer (Customer Name, <u>Customer SSN</u>, Customer Address)
  - <u>Card id</u> chosen as primary key of **Debit Card** relation
- *Customer SSN* chosen as foreign key in **Debit Card**
- Represents association between **Customer** and **Debit Card** relations
    - Debit Card (<u>Card Id</u>, PIN, Expiration date, Status, *Customer SSN*)
  - (underline = <u>primary key</u>, italic = *foreign key*)

One-to-Many Association

- One-to-many association maps to
  - Foreign key in “many” relation
  - E.g., Customer <u>Has</u> Account
- Static Model
  - Customer (Customer Name, Customer SSN, Customer Address)
  - Account (Account number, Balance)
- Relational Database Design
  - Primary key of “one” relation (Customer) is chosen as foreign key in “many” relation (Account)

One-to-Many Association

- Relational Database Design
  - Customer SSN is chosen as primary key of Customer relation
    - Customer (Customer Name, <u>Customer SSN</u>, Customer Address)
  - Account Number is chosen as primary key of Account relation
  - *Customer SSN* is foreign key in Account relation
    - Account (<u>Account Number</u>, Balance, *Customer SSN*)

Static Model Association Class

- Class models association between two or more classes
  - Usually for many-to-many associations
- Association class is mapped to associative relation
- Associative relation
  - Relation to represent association between two or more relations
  - Primary key of associative relation
    - Concatenated key
    - Formed from primary key of each relation that participates in association

Static Model
Association Class

- E.g., Hours association class
  - Represents association between Project and Employee classes
  - Mapped to Associative relation Hours
- Static Model
  - Project (Project id, Project name)
  - Employee (Employee id, Employee name, Employee address)
  - Hours (Hours Worked)
    - Hours Worked is attribute of association

Relational Database Design
Associative Relation

- Relational Database Design
  - Project (<u>Project id</u>, Project name)
  - Employee (<u>Employee id</u>, Employee name, Employee address)
- Project id and Employee id
  - Form concatenated primary key of Hours relation
  - Also foreign keys
    - Hours (*<u>Project id</u>*, *<u>Employee id</u>*, Hours worked)

Static Model
Aggregation/Composition Hierarchy

- Whole/part relationship
  - Aggregate/Composite (whole) class is mapped to relation
  - Each part class is mapped to relation
  - Primary key of composite/aggregate relation
    - All of primary key of component relation
      - 1-1 aggregation
    - Part of primary key of component relation
      - 1-n aggregation
    - Foreign key
      - If not needed to uniquely identify component relation

Relational Database Design
Aggregation/Composition Hierarchy

- E.g., Static Model
  - Department IS PART OF College
  - Admin Office IS PART OF College
  - College (College name)
  - Admin Office (Location)
  - Department (Department name, Location)
- Relational Database Design
  - Primary key of aggregate relation = College name
  - College (<u>College name</u>)
  - Admin Office (<u>College name,</u> Location)
  - Department (<u>Department name</u>, *College name*, Location)

Static Model
Generalization / Specialization Hierarchy

- Three alternative mappings from Generalization / Specialization Hierarchy to relational database
  - Superclass & subclasses mapped to relations
  - Subclasses only mapped to relations
  - Superclass only mapped to relation

Relational Database Design
Generalization / Specialization Hierarchy

- Superclass & subclasses mapped to relations
  - Superclass mapped to table
    - Discriminator is attribute of superclass table
  - Each subclass mapped to table
  - Shared id for primary key
    - Same primary key in superclass and subclass tables
  - Clean and extensible
  - However, superclass / subclass navigation may be slow

Relational Database Design
Generalization / Specialization Hierarchy

- Superclass & subclasses mapped to relations
- E.g.: Account Generalization / Specialization Hierarchy
- Static Model
  - Superclass: Account (Account number, Balance)
  - Subclass: Checking Account (Last Deposit Amount)
  - Subclass: Savings Account (Interest)
- Relational Database Design
  - Account (<u>Account number</u>, Account Type, Balance)
  - Checking Account (<u>Account Number</u>, Last Deposit Amount)
  - Savings Account (<u>Account Number</u>, Interest)

Relational Database Design
Generalization / Specialization Hierarchy

- Subclasses only mapped to relations
  - Map each subclass to relation
  - No superclass relation
  - Superclass attributes replicated for each subclass table
- Can use if
  - Subclass has many attributes
  - Superclass has few attributes
  - Application knows what subclass to search

Relational Database Design Generalization / Specialization Hierarchy

- Subclasses only mapped to relations
- Example of Account Generalization / Specialization Hierarchy
- Static Model
  - Superclass: Account (Account number, Balance)
  - Subclass: Checking Account (Last Deposit Amount)
  - Subclass: Savings Account (Interest)
- Relational Database Design
  - Checking Account (<u>Account Number</u>, Balance, Last Deposit Amount)
  - Savings Account (<u>Account Number</u>, Balance, Interest)

Relational Database Design Generalization / Specialization Hierarchy

- Superclass only mapped to relation
- All subclass attributes brought up to superclass table
  - Discriminator is attribute of superclass table
  - Each record in superclass table uses attributes relevant to one subclass
  - Other attribute values are null
- Can use if
  - Superclass has many attributes
  - Subclass has few attributes
  - Only two or three subclasses

Relational Database Design Generalization / Specialization Hierarchy

- Superclass only mapped to relation
- Example of Account Generalization / Specialization Hierarchy
- Static Model
  - Superclass: Account (Account number, Balance)
  - Subclass: Checking Account (Last Deposit Amount)
  - Subclass: Savings Account (Interest)
- Relational Database Design
  - Account (<u>Account Number</u>, Account Type, Balance, Last Deposit Amount, Interest)

Example of Relational Database Design

- Bank Information (underline = <u>primary key</u>, italic = *foreign key*):

Bank (<u>Bank Name</u>, Bank Address)
ATM Info (ATM Id, ATM Location, ATM Address, *Bank Name*)
Customer (Customer Name, <u>Customer Id</u>, Customer Address)
Debit Card (<u>Card Id</u>, PIN, Start Date, Expiration date, Status, Limit, Total, *Customer Id*)
Checking Account (<u>Account Number</u>, Balance, Last Deposit Amount)
Savings Account (<u>Account Number</u>, Balance, Interest)
Card Account (<u>*Card id*</u>, <u>*Account Number*</u>)
Customer Account (<u>*Customer Id*</u>, <u>*Account Number*</u>)

Assumption: Account type is determined from Account Number

Example of Relational Database Design

- Withdrawal Transaction (<u>Transaction Id</u>, Date, Time, Status, *Card Id*, Account number, Amount, Balance)
- Query Transaction (<u>Transaction Id</u>, Date, Time, Status, *Card Id*, Account Number, Balance, Last Deposit Amount)
- Transfer Transaction (<u>Transaction Id</u>, Date, Time, Status, *Card Id*, From Account Number, To Account Number, Amount)
- PIN Validation Transaction (<u>Transaction Id</u>, Date, Time, Status, *Card Id*, Start Date, Expiration Date)

Steps in Using COMET/UML

1 Develop Object-Oriented Requirements Model
  - Develop Use Case Model (Chapter 7)

2 Develop Object-Oriented Analysis Model
  - Develop static model of problem domain (Chapter 8)
  - Structure system into objects (Chapter 9)
  - Develop statecharts for state dependent objects (Chapter 10)
  - Develop object interaction diagrams for each use case (Chapter 11)

3 Develop Object-Oriented Design Model
  - Design Overall Software Architecture (Chapter 12)
  - Design Distributed Component-based Subsystems (Chapter 13)
  - Structure Subsystems into Concurrent Tasks (Chapter 14)
  - Design Information Hiding Classes (Chapter 15)
  - Develop Detailed Software Design (Chapter 16)