Fundamentals of Object-Oriented Analysis and Design

Advanced Software Engineering
Ali Kamandi
Sharif University of Technology
March 2007
kamandi@ce.sharif.edu
Introduction

Object-Oriented analysis is based upon concepts that we first learned in kindergarten: objects and attributes, classes and members, wholes and parts.

[Coad & Yourdon 1991]
Introduction

- To define all classes, the operations, the attributes, the relationships and the behavior the following tasks must be done:
  - Basic user requirements must be communicated between the customer and the software engineer.
  - Classes must be identified.
  - A class hierarchy must be specified.
  - Object to object relationships should be represented.
  - Object behavior must be modeled.
  - Above Tasks must be reapplied iteratively until the model is complete.
OO analysis and design methods

- Coad and Yourdon
- Booch method OOSE
- OMT
- RDD
- The Wirfs-Brock method (do not make clear distinction between design and analysis tasks)
Coad and Yourdon

- **Analysis process: five layers (1989)**
  - Finding classes
  - Identifying structures (Gen-Spec, Whole-Part)
  - Identifying subjects (subsystems)
  - Defining attributes
  - Defining services (operations)

- **Design process: four components (1991)**
  - Problem domain Component
  - Human Interaction Component
  - Task Management Component
  - Data Management Component

- Simplicity of notation
Booch method

- Design process: Incremental design
- Macro Process & Micro Process
- Notation: rich in symbols.
- Discussion: complicated notation
OSE

- Jacobson 1992 (Objectory 1987)
- Full lifecycle
- Analysis
  - requirement analysis, robustness analysis
  - Use Case Model, Domain Object Model, Interface Description
- Construction: design & implementation
- Testing
OMT (Rumbaugh 1991)

- The process: three phase (analysis, system design and object design)
- Object Model (Object classes & their relationships)
- Dynamic Model: Event-Trace diagram & state-chart diagram)
- Functional Model: Data Flow Diagram (DFD)
- Pragmatic: OMTool. Published text.
- Discussion: place more emphasis on specifying what an object is rather than how it is used.
OOG- A Generic View

- define use cases
- extract candidate classes
- establish basic class relationships
- define a class hierarchy
- identify attributes for each class
- specify methods that service the attributes
- indicate how classes/objects are related
- build a behavioral model
- iterate on the first five steps
Unified Modeling Language (UML)

User model view. This view represents the system (product) from the user’s (called “actors” in UML) perspective.

Structural model view. Data and functionality is viewed from inside the system. That is, static structure (classes, objects, and relationships) is modeled.

Behavioral model view. This part of the analysis model represents the dynamic or behavioral aspects of the system.

Implementation model view. The structural and behavioral aspects of the system are represented as they are to be built.

Environment model view. The structural and behavioral aspects of the environment in which the system is to be implemented are represented.

UML analysis modeling focuses on the first two views of the system. UML design modeling addresses the three other views.
Domain Analysis

- OOA at the **business area level** called Domain Analysis.
- Domain Analysis is performed to create a library of reusable classes applicable to an entire category of applications.
- Using a robust class library produces the system faster, cheaper and more reliable.
- But where did such a library come from? By applying domain analysis.
Domain Analysis Process

• The goal: to find or create those classes that are broadly applicable, so that they may be reused.

• It can be viewed as an umbrella activity for the software process.

• The role of domain analyst is to design and build reusable components that maybe used by many people working on similar but not necessarily the same applications.

• Key inputs and outputs for the domain analysis process:
The OO Process Model: RUP
The purposes of business modeling

- To understand the structure and the dynamics of the organization in which a system is to be deployed (the target organization).
- To understand current problems in the target organization and identify improvement potentials.
- To ensure that customers, end users, and developers have a common understanding of the target organization.
- To derive the system requirements needed to support the target organization.
Artifacts of Business Modeling

- Business Use Case Model
- Business Object Model
- Business Glossary
- Business Rules
- Business Vision
- Target Organization Assessment
- ...

Scope of Business Modeling

- Organization Chart
- Domain Modeling
- One Business Many Systems
- Generic Business Model
- New Business
- Revamp Business
Organization Chart

- Build a simple map of the organization and its processes to get a better understanding of requirements.
- Business modeling is part of software engineering project (Inception phase)
- No change in organization (no BPI/BPR)
Domain Modeling

- Build a model of information at a business level (without considering workflows)
- As part of software engineering project
- Inception and Elaboration
One business- Many systems

- Building a large system or a family of applications
- One business model will serve as input to several SE projects.
- Help to Find functional requirements and architecture of the application family
Generic Business Model

- One application- several organizations
- Avoid complex requirements
- BPI
New Business/ Revamp

- Find requirements
- Determine the feasibility of the new line of business
- Revamp: BPR
- The business-modeling effort is, in these cases, often treated as a project on its own.
Business Process Modeling

[Diagram showing relationships between entities like Passenger, Individual Check-In, Group Check-In, Tour Guide, and Baggage Handling.]
Business Process Modeling

Activity state

Verify reservation

Alternative threads

Send to airport travel agency

Guard condition

[correct]

Concurrent threads

Get preferences

Transition

[incorrect]

Synchronization bar (fork)

Decision (branch)

[no baggage]

Receive baggage and print receipt

[baggage]

Print boarding card

Decision (merge)

Synchronization bar (join)

Give travel documentation to passenger
Business Object Model

Ticket
Passenger name: Name

Flight
Departure: Time
Arrival: Time

Airline
Requirement Management

- To establish and maintain agreement with the customers and other stakeholders on what the system should do.
- To provide system developers with a better understanding of the system requirements.
- To define the boundaries of the system.
- To provide a basis for planning iterations.
- To provide a basis for estimating cost and time to develop the system.
- To define a user-interface for the system, focusing on the needs and goals of the users.
Requirements: Artifacts

- Use Case Model
- SRS
- Vision
- Supplementary Specification
- Glossary
- ...
Analysis: Purpose

- To transform the requirements into a design of the system.
- To evolve a robust architecture for the system.
Analysis: Artifacts

- Analysis Model
- Software Architecture Document
A First Look
Analysis: Finding Classes

- CRC Card
- Responsibility Driven Method
- Use Case Driven Method
Analysis Phase: What’s Next?

- Find the objects needed for the use cases
- Determine the responsibility for each object
- Determine what objects work together to complete a use case
- Verify all use cases are covered
Identifying Objects

- Work use case by use case
- Extract noun phrases and build a list
- Identify candidate objects
  - Physical objects
  - Conceptual entities
  - Categories of objects
  - External interfaces
CRC Cards (1)

*Class, Responsibility, Collaborator*

- Helps to find object interactions
- Gives a physical entity (the card) to work with
  - 4" x 6” note cards recommended
- 3 pieces of data:
  - Class – name and purpose of the class
  - Responsibility – what the class knows and the behaviors it performs
  - Collaborator – Other classes that need to help with a responsibility
CRC Cards(2)

MessageBuilder
Builds message from selections
Presents guesses to user
Controls the pacing

Purpose: The MessageBuilder is a hub of activity in the application. It coordinates the timing, the presentation of guesses, the message construction. It centralizes control and is a core element of the control architecture.
CRC Card Format(3)

<table>
<thead>
<tr>
<th>Class:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsibility</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

- Write class description on the back
CRC Cards (4)

- Make 1 CRC card for each object in your list
  - Fill in the class name
    - Naming is important – try to name classes so their role in the problem domain is obvious
    - Use standard terminology where possible (i.e. from domain glossary or dictionary)
  - Write description on the back of the card
    - Helps clarify why the card exists
    - Documents your thinking at this point in time
  - Leave other sections blank for now
Analysis Phase: What’s Next?

- Find the objects needed for the use cases
- Determine the responsibility for each object
- Determine what objects work together to complete a use case
- Verify all use cases are covered
Responsibilities

- The knowledge an object maintains
- Actions an object can perform
- Only the publicly available services at this stage
- Still only **what** gets done, not **how**
Identifying Responsibilities

- Work use case by use case
- Identify verbs that represent an action an object must perform
  - What does the class have to be able to do?
- Identify information that some object must maintain
  - What does the class need to know?
CRC Cards (5)

- Add responsibilities to CRC card as a phrase
  - E.g. “Know machine name of master controller”

- If card fills up:
  - May have gone into too much detail
  - May be good indication to split class into several classes
  - Leads to more flexible design
Analysis Phase: What’s Next?

- Find the objects needed for the use cases
- Determine the responsibility for each object
- Determine what objects work together to complete a use case
- Verify all use cases are covered
Identifying Collaborators

- For each responsibility ask:
  - Is the class capable of handling this responsibility itself?
  - If not, what does it need (that it can’t do by itself)?
  - From what other class can it acquire what it needs?
Identifying Collaborations to Offer

For each class ask:
- What does this class do or know?
- What other classes need the information I have?

If a class has no interactions, discard it

Write the name of the class providing the collaboration to the right of the responsibility it helps fulfill
Analysis Phase: What’s Next?

- Find the objects needed for the use cases
- Determine the responsibility for each object
- Determine what objects work together to complete a use case
- Verify all use cases are covered
Use Case Verification

- Walk through all flows of all use cases
- When an unmet responsibility is discovered
  - Add it to an existing class; or
  - Create a new class
- When a missing class is discovered, make a CRC card for it
- Keep repeating until you can walk through each flow without discovering a new class, responsibility or collaborator
Finding Objects: Use Case Realization

1: Item inserted
2: Handle item
3: Increment item
4: Receipt requested
5: Handle receipt
6: Get item values
7: Print item values
8: Print total
9: Get paper status
10: Activate

:Customer
:Deposit Item Type
:Deposit Item Receiver
:Alarm Device
:Receipt Printer
:Customer Panel
Putting the Pieces Together

Order
- dateReceived : Date
- isPrepaid : Boolean
- number : Integer
- total : Currency
+ close() : void
+ dispatch() : void

Customer
- name : String
- address : Address
- CreditRating : String
+ GetCreditRating() : String{query,sequential}

CorporateCustomer
- ContactName : String
- CreditLimit : Currency
+ Remind() : void
+ BillForMonth() : void

PersonalCustomer
- CreditCardNumber : String

Employee

OrderLine
- isSatisfied : Boolean
- price : Currency
- quantity : Integer

{if Order.Customer.GetCreditRating() is "poor", then Order.isPrepaid must be true}

line items

Product

1


1

0..1

sales rep
Data Collection Domains

CUSTOMER
- Data Sensor
- Analysis Algorithm
- Browsing Program
- Clock/Timer
- Presentation/Display
- Central Computer (data holder)
- Location
- Environmental Parameter

Sensor Type
- Location
- Value Object

Data Links
- Phone, Satellite
- Signal - Transmitter

NOTE: (value object)
Analysis and Design: Artifacts

- Analysis Model
- Software Architecture
Architecting a dog house

Can be built by one person
Requires
  Minimal modeling
  Simple process
  Simple tools
Architecting a house

Built most efficiently and timely by a team
Requires
  - Modeling
  - Well-defined process
  - Power tools
Architecting a high rise
Early architecture

Progress
- Limited knowledge of theory
Modern architecture

Progress
- Advances in materials
- Advances in analysis
Modeling a house
We all know that ...

- Architecture and design are the same thing
- `<my favorite technology>` is the architecture
- A good architecture is the work of a single architect
- Architecture is flat, one blueprint is enough
- Architecture is just structure
- System architecture precedes software architecture
- Architecture cannot be measured and validated
- Architecture is a Science
- Architecture is an Art
Software architecture encompasses the set of significant decisions about the organization of a software system:

- selection of the structural elements and their interfaces by which a system is composed
- behavior as specified in collaborations among those elements
- composition of these structural and behavioral elements into larger subsystem
- architectural style that guides this organization
Architectural style

- An architecture style defines a family of systems in terms of a pattern of structural organization.

- An architectural style defines
  - a vocabulary of components and connector types
  - a set of constraints on how they can be combined
  - one or more semantic models that specify how a system’s overall properties can be determined from the properties of its parts
Many stakeholders, many views

- Architecture is many things to many different interested parties
  - end-user
  - customer
  - project manager
  - system engineer
  - developer
  - architect
  - maintainer
  - other developers

- Multidimensional reality

- Multiple stakeholders
  multiple views, multiple blueprints
Architectural view

- An architectural view is a simplified description (an abstraction) of a system from a particular perspective or vantage point, covering particular concerns, and omitting entities that are not relevant to this perspective.
Architecturally significant elements

- Not all design is architecture
- Main “business” classes
- Important mechanisms
- Processors and processes
- Layers and subsystems
- Architectural views = slices through models
Representing System Architecture

Logical View
- End-user
  - Functionality

Implementation View
- Programmers
  - Software management

Process View
- System integrators
  - Performance
  - Scalability
  - Throughput

Deployment View
- System engineering
  - System topology
  - Delivery, installation
  - Communication

Use Case View
Architectural patterns

- Layered
- Pipes and filters
- Distributed
- Event-driven
- Batch
- Repository-centric
- Blackboard
- Interpreter
- Rule-based
- ...

Software Architecture
Shaw and Garlan
Buschmann et al
A System of Patterns
Buschman et al
Booch
The Design Process

- Design is a messy, iterative process
- Early descriptions tend to be less precise
- Later descriptions add more precision and formality
Object Oriented Design

- OOD transforms the analysis model created using OOA into a design model that serves as a blueprint for software construction.
- OOD results in a design that achieves a number of different levels of modularity.
- Subsystems: Major system components.
- Objects: Data and the operations.
...then get more precise
Object-Oriented Design

• The subsystem layer: Representation of each of the subsystems that enable the software to achieve its customer defined requirements.

• The class and object layer: The class hierarchies, (generalization) and representation of objects.

• The message layer: The design details of communication of each object with its collaborators. (external and internal interfaces)

• The responsibilities layer: Data Structure and algorithmic design for all attributes and operations.
Object-Oriented Design

- The design pyramid focuses exclusively on the design of a specific product or system.
- Another layer of design which forms the foundation on which the pyramid rests, exists.
- The foundation layer focuses on the design of domain objects.
- Domain objects play a key role in building the infrastructure for the OO system by providing support for:
  - Human/computer interface activities,
  - Task management,
  - Data management.
OOA and OOD

THE ANALYSIS MODEL

THE DESIGN MODEL
Generic Components for OOD

- Problem domain component—the subsystems that are responsible for implementing customer requirements directly;
- Human interaction component—the subsystems that implement the user interface (this included reusable GUI subsystems);
- Task Management Component—the subsystems that are responsible for controlling and coordinating concurrent tasks that may be packaged within a subsystem or among different subsystems;
- Data management component—the subsystem that is responsible for the storage and retrieval of objects.
Process Flow for OOD

- System design
- Object-oriented analysis
- Object design
- Task management design
- Data management design
- Human interface design
System Design Process

- Partition the analysis model into subsystems.
- Identify concurrency that is dictated by the problem.
- Allocate subsystems to processors and tasks.
- Develop a design for the user interface.
- Choose a basic strategy for implementing data management.
- Identify global resources and the control mechanisms required to access them.
- Design an appropriate control mechanism for the system, including task management.
- Consider how boundary conditions should be handled.
- Review and consider trade-offs.
System Design

client subsystem → request → server subsystem

peer subsystem → request → peer subsystem

peer subsystem → request → peer subsystem
Subsystem Design Criteria

- The subsystem should have a well-defined interface through which all communication with the rest of the system occurs.

- With the exception of a small number of “communication classes,” the classes within a subsystem should collaborate only with other classes within the subsystem.

- The number of subsystems should be kept small.

- A subsystem can be partitioned internally to help reduce complexity.
Object Design

- A *protocol description* establishes the interface of an object by defining each message that the object can receive and the related operation that the object performs.

- An *implementation description* shows implementation details for each operation implied by a message that is passed to an object:
  - information about the object's private part
  - internal details about the data structures that describe the object’s attributes
  - procedural details that describe operations
Case Study

- Weather Station
Use-cases for the weather station
Subsystems in the weather mapping system

- **Data collection**
  - Observer
  - Satellite
  - Weather station
  - Balloon

- **Data processing**
  - Data checking
  - Data integration

- **Data display**
  - User interface
  - Map display
  - Map
  - Map printer

- **Data archiving**
  - Data storage
  - Map store
  - Data store
Weather station architecture

- Weather station
  - «subsystem» Interface
  - «subsystem» Data collection
  - «subsystem» Instruments

  - Manages all external communications
  - Collects and summarises weather data
  - Package of instruments for raw data collections
Weather station object classes

**WeatherStation**
- identifier
- reportWeather()
- calibrate (instruments)
- test()
- startup (instruments)
- shutdown (instruments)

**WeatherData**
- airTemperatures
- groundTemperatures
- windSpeeds
- windDirections
- pressures
- rainfall
- collect()
- summarise()

**Ground thermometer**
- temperature
- test()
- calibrate()

**Anemometer**
- windSpeed
- windDirection
- test()

**Barometer**
- pressure
- height
- test()
- calibrate()
Weather station subsystems

- CommsController
- WeatherStation
- WeatherData
- Instrument Status
- Air thermometer
- RainGauge
- Anemometer
- Ground thermometer
- Barometer
- WindVane
Data collection sequence
Weather station state diagram
Design Patterns

... you’ll find recurring patterns of classes and communicating objects in many object-oriented systems. These patterns solve specific design problems and make object-oriented design more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience. A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them.

Gamma and his colleagues [GAM95]
Design Pattern Attributes

- The design pattern name is an abstraction that conveys significant meaning about its applicability and intent.
- The problem description indicates the environment and conditions that must exist to make the design pattern applicable.
- The pattern characteristics indicate the attributes of the design that may be adjusted to enable the pattern to accommodate into a variety of problems.
- The consequences associated with the use of a design pattern provide an indication of the ramifications of design decisions.
• Generally codify expert knowledge of design strategies, constraints & “best practices”
End.