Unified Modeling Language

郭大維教授
臺灣大學資訊工程系

Introduction

Unified Modeling Language
Object-Oriented Analysis and Design (OOA&D)
Specifying, Visualizing, and Documenting Computer Systems.
Important Contributors:
Grady Booch, Jim Rumbaugh, and Ivar Jacobson
Object Management Group (OMG)

Introduction

Development of Object-Oriented Methods
Simmla 67 Language (Ole-Jone Dahl in 1967), Smalltalk, C++, etc.
OO Methods, 1980’s, 1990’s
OOA, Booch
OOA/OOD, Coad and Yourdon
OMT, Rumbaugh
OOAD, Odell
OOSE, Jacobson

Introduction

UML - A Modeling Language
Notations
Modeling Methods
A modeling language
A process
What steps to take in doing a design!
Rational Unified Process (RUP)
By Booch, Rumbaugh, and Jacobson

Contents

Introduction
Development Process
Use Cases
Class Diagrams & Object Diagrams
Interaction Diagrams
Packages and Collaborations
State Diagrams, Activity Diagrams, Physical Diagrams
Misc
Introduction
- Notations
  - Graphical stuff
    - Syntax of the language, e.g., class
- Meta-Models
  - Diagrams, e.g., class diagrams

A Class-Diagram Example

Customer
Name
Address

Company
Name
Credit Status
Credit Limit
Notification

Customer Name: String
Address:

Credit-Status

Invoice
Day: Date
Payment:
  - Boolean
    - No:
      - String
    - Price:
      - Money
  - Deliver

Learning OO
Help people to do good OO.
Communicating with Domain Experts
Understanding of users’ world
Use Cases and Class Diagrams!

Why “Unified”?
- Across historical methods and notations
- Across the development lifecycle
- Across application domains
- Across implementation languages and platforms
- Across development process
- Across internal concepts

Objectives of UML
- Specifying, Visualizing, and Documenting Computer Systems.
- Elements – Vocabularies
- Design Guidelines and Experience Rules – Grammar

OO Method vs Formal Specification/Design Language
- Less rigorous, easy understanding and manipulation
  \[ i, \exists (i, i) \ ? \ P \]
- Standard vs Nonstandard Methods
  Flexibility, Automatic Analysis, and Info/Code Exchanging

Why do analysis and design?
- Communication
  - Code is precise but too detailed.
  - E.g., package diagrams to show the major system components.
- Learning OO
  - Help people to do good OO.
- Communicating with Domain Experts
  - Understanding of users’ world
  - Use Cases and Class Diagrams!
Introduction

Five UML View-Points

- Use-Case View
  - Specify system functionality for users, designers, and test engineers.
  - Diagrams: use cases, sequence, collaboration, state, activity diagrams.

- Design View
  - Specify detailed design of the system’s internal functionality, including use cases and actors.
  - Diagrams: Class, Object, State, Sequence, collaboration, activity diagrams.

- Implementation View
  - Specify how to split the system into software components and do implementation.
  - Diagrams: state, sequence, collaboration, activity diagrams.

- Process View
  - Specify the operation of the entire system
  - Diagrams: component, state, sequence, collaboration, activity diagrams.

- Deployment View
  - Specify the architecture of the system hardware and the deployment of software processes.
  - Diagrams: deployment, state, sequence, collaboration, activity diagrams.

UML Vocabularies

- Things
  - Structural things, e.g., classes, components, use cases.
  - Behavioral things, e.g., Interaction and state machine.
  - Grouping of things, e.g., package.
  - Annotational things, e.g., notes.

- Relationships
  - Dependency
  - Association
  - Generalization
  - Realization

- Diagrams
  - Use case, class, object, sequence, collaboration, state, activity, component, deployment.
Contents

Introduction
Development Process
Use Cases
Class Diagrams & Object Diagrams
Interaction Diagrams
Packages and Collaborations
State Diagrams, Activity Diagrams, Physical Diagrams
Misc

Development Process

UML - A Modeling Language
Notations
Modeling Methods
A modeling language
A process
What steps to take in doing a design!
Rational Unified Process (RUP)
By Booch, Rumbaugh, and Jacobson

Overview

An iterative and incremental development process:
Each construction iteration
Analysis, design, implementation, testing, and integration.

Inception Elaboration Construction Transition

Inception

Goal:
Establish the business rationale for the project.
Decide the scope of the project.
Forms:
Informal chatting
Full-pledged feasibility study
What should be done?
Work out the business case – cost and income!

Elaboration

Want to get a better understanding of the problem:
What is it you are actually going to build?
How are you going to build it?
Contents:
Collect more detailed requirements.
Do high-level analysis and design to establish a baseline architecture.
Create the plan for construction.

Informal chatting
Full-pledged feasibility study
What should be done?
Work out the business case – cost and income!
Elaboration

Risks:
- Requirement Risks
- Technical Risks
- Skill Risks
- Political Risks

Elaboration - Requirement Risks

Q:
- Will we build a wrong system?
- Starting points:
  - Use cases
    - A typical interaction that a user has with the system in order to achieve a goal!

Elaboration - Requirement Risks

The usage of use cases
- Indicate a function that users can understand and that has a value for users.
- No too much detailed!
- Domain Model
  - A model whose primary subject is the world the computer system is supporting!

Elaboration - Requirement Risks

Important tasks for elaboration
- Get all potential use cases, especially the most important and riskiest ones.
- Come out the skeleton of the conceptual model of the domain:
  - How the business operates?
  - Lays a foundation for the object model that will represent objects supported by the system.

Elaboration - Requirement Risks

UML Techniques for Conceptual Domain Model:
- Class Diagram
  - Definitions of vigorous vocabulary about the domain.
- Activity Diagram
  - Encouraging the finding of parallel processes.
- Interaction Diagram
  - Exploring different roles interact in the business

Elaboration - Requirement Risks

A Class-Diagram Example

Company

Credit Status

Credit Limit

Notification

Payment

Invoice

Day: Date

Payment: Boolean

No: String

Price: Money

Deliver

Customer

Name: String

Address:

Person

Credit-Status

Credit Card

Person

Invoice

Company

Credit Status

Credit Limit

Notification

Payment

Invoice

Day: Date

Payment: Boolean

No: String

Price: Money

Deliver

Customer

Name: String

Address:

Person

Credit-Status

Credit Card
Elaboration - Requirement Risks

**Remark**
- Use minimum notation
- Focus on important issues and risky areas
- A starting point for building classes in the construction phase
- Use package diagrams if needed
- A skeleton – concentrate on important details, instead of all.

Elaboration - Technological Risks

**Q:**
- Will the selecting technology actually do the job for us?
- Will the various pieces fit together?
- Possible solution:
  - Build prototypes to try out technology!
Elaboration - Technological Risks

- Biggest Challenge:
  - How the components of a design fit together?
  - E.g., Java + database + session + ...
- Must:
  - Address any architecture design decisions!
  - Especially for distributed systems!

- Questions: How can we change the elements of the design relatively easy?
- What will happen if a piece of technology doesn't work?
- What if we can't connect two pieces of the puzzle?
- What is the likelihood of something going wrong?
- Look at use cases to do assessment!
  - Class diagrams, interaction diagrams, package diagrams, deployment diagrams.

Elaboration - Skills Risks

- Skill Risks
  - Can you get the staff and expertise you need?
  - Always little experience and thought
- Solutions
  - Short training
  - Mentoring
  - Project reviewing every specific period of time
  - Reading
  - Pattern learning

Elaboration - Political Risks

- Political Risks
  - Are the political forces that get in the way and seriously affect your project?
  - Internal
  - External
- Solutions
  - Find good ones to do it if you cannot!

Elaboration - Duration

- A fifth of the total length of the project.

- Events to signal the termination
  - Developers feel comfortable providing estimates to the person-week effort.
  - All significant risks have been identified, and how you intend to deal with them are known.
Planning of the Construction Phase

Goal
Be aware of progress
Signal progress through the team

Essence
Set up a series of iteration
Define the functionality to deliver in each iteration

Planning of the Construction Phase

Method
Customer vs Developer
Customer
Assess the business value of a use case.
Developer
Build the system

Planning of the Construction Phase - Steps

Steps
Categorize use cases according to the business value and development risks!
Determine your iteration length
A fixed iteration with a handful of case uses being implemented.
Project velocity
Developer-week per iteration = 
( #developers * iteration-length ) / load-factor
Iteration
( Development-time of all use cases / Developer-week per iteration ) + 1

Planning of the Construction Phase - Steps

Assign use cases to iterations
Do not put off risk until the end!

Contingency Factor
10~20 percent of the construction time.
Transition
10~35 percent of the construction time for tuning and packaging
Ready for a release plan!

Construction

Goal
Build the system in a series of iterations.
Demo and confirm the implementation.
Reduce risk!

Iterations within construction are both incremental in function and iterative in the code base!
Refactoring!
Integration!

Remark
Self-Testing Software
Testing as a continuous process!
Unit test code by the developers
Function test code developed by separate team
When the plan goes away!
Time-boxed!
Redo the plan!
**Remark**
- Refactoring – a couple of small steps
- Rewriting vs redesigning
- Never refactor a program and add functionality to it at a time.
- Have a good test in-place before refactoring
- Take short, deliberate steps
- Avoid debugging!

**Using UML in the Construction**
- Add a use case
- Check class diagrams to see how they fit the software been built!
- How classes collaborate to implement the functionality required by each use case
- Try interaction diagrams!
- If the change is serious, use the notations to discuss with colleagues!

**Using UML in the Construction**
- Use UML to help document what is built!
- Detailed documentation should be from the code! (+ additional doc)
- Use package diagrams as the logical road map of the system!
- Dependencies of logical pieces
- Use deployment diagrams to show the high-level physical picture!

**Using UML in the Construction**
- For a class with a complex behavior
- Use state diagrams to describe it!
- Use iteration diagram to describe complicated interactions among classes
- When a complex algorithm is involved
- Use activity diagram to understand the code!

**An State Diagram Example**

**Transition**
- Goal:
  - Development to fix bugs!
  - No additional of substantial functionality!
  - Beta-testing, performance tuning, user training, etc.
- Why iterative development?
  - Do the development process regularly!
  - Get used to deliver finished code!
- Tradeoff
  - Meet users' requirements
  - Optimize code!
**When To Use iterative Development**

Only on projects you want to succeed!

---

**Use Cases**

Why use cases?

- People need a way to communicate in project development and planning.
- Scenarios – those behind use cases
- A sequence of steps describing an interaction between a user and a system.

The customer browses the catalog and adds desired items to the shopping basket. When the customer wishes to pay, the customer describes the credit and shopping info and confirms the sale. The system checks the authorization on the credit card and confirms the sale both immediately and with a follow up email.

---

**Use Cases**

A use case is a set of scenarios tied together by a common user goal!

- Buy a Product – Use-Case Text:
  1. Customer browses the catalog and selects items to buy.
  2. Customer goes to check out.
  3. Customer fills up in the shipping information.
  4. System presents full pricing information, including shipping.
  5. Customer fills in credit card information.
  7. System confirms sale immediately.
  8. System sends confirming email to customer.

Alternative: Authorization Failure

- At Step 6, if the authorization fails, let customer try again!

---

**Use Cases**

How to create a use case?

- Describe the primary scenario and alternatives as variations on that sequence!
- The existence of preconditions!
- Divide up use cases
- E.g., Regular Customer – skip steps 3, 4, and 5 when info is already there.
- Need an amount of detail depending on the risk in each use case!

---

**Use Case Diagrams**

Introduced by Jacobson in 1994

- Show what needed to build in each iteration!
- Trading (Chp3)

Primary Elements:

- Actor
  - A role that a user or an external system plays with respect to the system!
  - A user can play more than one role.
  - Actors, who carry out use cases, are useful when trying to come up with the use cases.
Use Case Diagrams

- Situations worth tracking the actors later:
  - Need configuring for various of users.
  - Help in negotiating priorities among various actors.
- Remark:
  - Use cases may not have clear links to specific actors.
  - A good source for identifying use cases is external events!

Use Case Diagrams

- From the external point of view
  - It describes what use cases are.
  - Scope and Constraints
  - What users really want!
- From the internal point of view
  - It describes how use cases operate.

Use Case Relationships

- Include
  - Occur when you want to avoid repetition.
  - E.g., Analyze-Risk and Price-Deal "include" Valuation.
- Generalization
  - Describe a variation on normal behavior (casually).
  - Override the base use case!
  - E.g., Limits-Exceeded is "generalized" into "Capture-Deal".

Use Cases Diagrams

- Extend
  - Similar to generalization but with more rules to it.
  - Extension points for adding behavior to the base use case.
  - Example – Buy-a-Product and Regular Customer
  - Generalization and Extend may cause the splitting of complicated use cases.

Use Cases

- System Use Cases
  - An interaction with the software.
  - E.g., text copying and style def. functionality
- Business Use Cases
  - How a business responds to a customer or an event.
  - E.g., unifying text formats.
- Order in Elaboration
  - Business use cases first
  - System use cases to satisfy business use cases
  - Use cases represent an external view.

Use Case Diagrams

- Use Case Boundary
  - To identify what is external or internal
  - Typical system boundaries
    - HW/SW boundary of a device or computer system
    - Dept of an organization
    - Entire organization
  - Examples
    - Wire in paychecks
Class Diagrams

Why Class Diagrams?
- Central within object-oriented methods in modeling systems and the relationship among their components.

Usages of Class Diagrams
- Types, attributes, operations of objects
- Static relationship among them
- Association
- Subtypes, etc.

Class Diagrams

Three Perspectives in Drawing Class Diagrams (Cook and Daniels, 1994):
- Conceptual – << type >>
  - Represent the concepts in the domain under study – maybe no direct mapping to classes
  - Should be language-independent
- Specification – << type >>
  - Consider software and the interfaces of the software
  - No implementation should be considered.
- Implementation – << implementation class >>
  - Lay down the implementation bare
  - Lines between perspectives are not sharp; however, it is important to separate the specification perspective and implementation perspective!!
  - Perspectives are no part of the formal UML but are useful in modeling.

Class Diagrams

Association
- Relationship between instances of classes.
- Association Ends – Roles!
- Multiplicity – *, 1..n, etc.
- Navigability
- Order – Customer
- Naming
- Associations – verbs
- Roles – nouns
- The diagram indicates only the interface – nothing more!
### Class Diagrams
- In implementation model
- Pointers in both directions between the related classes.
- class Order {
  private Customer _customer;
  private Set _OrderLines;
  ...
- Navigability!
  - Unidirectional/Bidirectional Associations
  - Inverse Constraints

### Attributes
- Attributes denote the status and characteristics of classes
- Single valued
- visibility name:type = default-value
- Optional, e.g., dateReceived[0..1]:Date
- Perspectives
  - At the conceptual level
  - A way to set values
  - At the implementation
  - A field for a attribute

### Operations
- Definition:
  - Operations are processes that a class knows to carry out.
  - Operations correspond to the methods on a class.
  - Perspectives
    - At the conceptual level,
    - The principal responsibilities of classes
    - At the specification level,
    - Methods on a class

### Attributes
- Attributes denote the status and characteristics of classes
  - Single valued
  - visibility name:type = default-value
  - Optional, e.g., dateReceived[0..1]:Date
- Perspectives
  - At the conceptual level
  - A way to set values
  - At the implementation
  - A field for a attribute

### Operations
- At the implementation level,
  - Private (-), public (+), and protected (#) operations, as well:
  - visibility name (parameter-list); return-type-expression [property-string]
  - + balanceOn (date:Date): Money
  - Parameter
    - Direction name:type = default-value
    - (direction: in, out, inout)

### Generalization
- Definition
  - “Super-type” – inverse of specialization
- Perspectives
  - At the conceptual level,
    - Everything about a “super-type” is true for a “subtype”.
  - At the specification level,
    - The interface of a “subtype” must conform to that of a “super-type”.

### Operations
- Types – constraints
  - Queries
    - Marked as { query }
  - Modifiers
  - Getting/Setting Methods (internal knowledge)
- Operations vs Methods
  - The body of a procedure – method (body)
  - Method call/declaration – operation
  - e.g., polymorphism – subtyping
Generalization

- Substitutability of code
- Polymorphism
- At the implementation level,
  inheritance in programming languages
- Subclassing is one way to implement
  subtyping but not the only way.
- Stability of Generalization

Constraints Rules

- Constraints ( ) on attributes, associations, generalization, etc.
- Format:
  - Informal English statements
  - Object Constraint Language (OCL)
  - flight.pilot.training_hour >=
    flight.plane.minimum_hours

Class Diagrams

- When to use them?
- Do not try to use all the notations available to you.
- Fit the perspective from which you are drawing the models to the stage of the project:
  - Concept model – analysis
  - Specification model – software
  - Implementation model – illustrate implementation techniques
- Concentrate on key areas

Design by Contract
(Bertrand Meyer)

- Assertions
  - A Boolean statement that should never be false.
- Types:
  - Pre-conditions – checked by callers
  - What we expect!
  - Post-conditions – checked by operations
  - What we do! (e.g., square-root)
  - Invariants – constraint rules on class diagrams
  - May be false during the execution on a method (e.g., balance == sum(entries.amount)).

Design by Contract
(Bertrand Meyer)

- Assertions
  - Subclassing
  - Strengthen the invariants or post-conditions!
  - Weakening the pre-conditions!
  - Substitution
  - Ideally as a part of the code!
  - Invariants are equivalent to constraint rules on class diagrams!

Contents

- Introduction
- Development Process
- Use Cases
- Class Diagrams & Object Diagrams
- Interaction Diagrams
- Packages and Collaborations
- State Diagrams, Activity Diagrams, Physical Diagrams
- Misc
**Interaction Diagrams**

- **Purpose:**
  - Models that describe how groups of objects collaborate in some behavior.
  - Capturing the behavior of a single use case typically.
- **Types:**
  - Sequence Diagrams
  - Collaboration Diagrams

**Sequence Diagrams**

- **A Object – A Box**
- **Lifeline – Object’s Life**
- **Message**
  - An arrow between the lifelines of two objects
  - Message Order (top to bottom)
  - Labeled with a name, arguments, control information, etc.
  - A Self-Call

**Sequence Diagrams**

- **Control Information**
  - A condition, e.g., [needsReorder]
  - An iteration marker, e.g., *[for all order lines]*
- **Return**
  - A return from a message – a dashed line
- **Asynchronous Message** – do not block the caller
- **Create a new thread or a new object**
- **Communicate with a thread that is running.**

**Collaboration Diagrams**

- **An Interaction Diagram which provide the spatial layout of objects!**
- **Notation**
  - objectName:ClassName
  - Numbering of Messages
  - Simple Numbering
  - Decimal Numbering
  - Which operation is calling which operation!

**Collaboration Diagrams**

- **Collaboration Diagrams with Simple Numbering**
- **Notation**
  - Object Deletion
  - Why Sequence Diagrams?
  - Emphasize on sequence
  - Capture the overall flow of control!
  - Show concurrent processes!
Collaboration Diagrams

Collaboration Diagrams with Decimal Numbering

Interaction Diagrams

Why Collaboration Diagrams?
- Use the layout to indicate how objects are statically connected.

When to use Interaction Diagrams?
- Behavior of several objects within a single use case typically – simplicity!
- Collaborations among the objects

When alternatives are considered?

Contents

Introduction
- Development Process
- Use Cases
- Class Diagrams & Object Diagrams
- Interaction Diagrams
- Packages and Collaborations
- State Diagrams, Activity Diagrams, Physical Diagrams
- Misc

Class Diagrams: Advanced Concepts

Class Diagrams
- Central within object-oriented methods in modeling systems and the relationship among their components.
- Many more notations:
  - Stereotypes
  - Core extension mechanism of UML
  - Subtypes of Class, Association, and Generalization

Class Diagrams

- Interface
  - An example of Stereotypes!
  - A class that has only public operations with no method bodies or attributes
  - <<interface>>
- Profile
  - Extend a part of UML with stereotypes for a particular purpose.

Object Diagrams

- A snapshot of the objects in a system at a point in time.
- An instance diagram
  - instance name: class name
- Both are optional – :Person
- A collaboration diagram without messages
Class Diagrams

- Class Scope Operations and Attributes
- Class Scope vs Instance Scope

Classification
- Relationship between an object and its type
- Single vs Multiple Classification
  - Single Classification:
    - An object belongs to a single type, which may inherit from supertypes.

Multiple Classification
- An object may have any of these types assigned to it in any allowable combination.
- Discriminator
  - An indication of the basis of the subtyping – disjoint!
- Constraint (complete)
  - An instance of the superclass must be an instance of one of the subtypes of a group.

Multiple, Dynamic Classification
- Change objects’ type within the subtyping structure.
- Multiple, Dynamic Interface
  - Additional behavior

Aggregation and Composition

- Aggression
  - A part-of relationship
    - E.g., car and engine
    - A style instance may be shared by a polygon and a circle.
- Composition
  - A stronger variety of aggregation
    - The parts are usually expected to live or die with the whole.
    - E.g., a point must belong to a polygon!
**Aggregation and Composition**

- **Alternative Notation:**
  - Circle
  - Polygon
  - Point

**Derived Associations and Attributes**

- **Derived Features**
  - Derived from others on a class diagram
- **Conceptual Perspective**
- **Specification Perspective**
- **Constraints between values — age**
- **Implementation Perspective**
- **Caching for performance reasons**

**Interfaces and Abstract Classes**

- **Interface**
  - Abstract Class: A class with no implementation (fields and method bodies) but operation declarations
  - Italicize the abstract item name and label it with the \{abstract\} constraints

**Reference Objects and Value Objects**

- **Reference Objects**
  - An object with an identity and can be referenced — no copies (change synchronization)!
  - E.g., Customer

**Multivalued Association Ends**

- **A Multivalued End**
  - Whose multiplicity’s upper bound is greater than 1, e.g., \(*\)
  - **Constraint**
  - **Sets** – basic type
  - **ordered**
  - **bag** – multi-set
  - **hierarchy**
  - **dag**
**Frozen Constraint**

- **Frozen Constraint**
  - Attribute – no change on values, value set at the object creation time.
  - { frozen } vs { read only }
  - Association End – the association end on a class could not be changed.

**Associations**

- **Classification**
  - The object Sheep is an instance of the type Border Collie
- **Generalization** – being transitive
  - The type Border Collie is a subtype of the type Dog
- **Quantified Associations**
  - Specification Perspective – imply an interface!
  - Implementation Perspective – The use of a data structure to hold data

**Associations**

- **A Qualified Association**

```
class Order {
    public OrderLine getLineItem(Product aProduct);
    ...
    class Order {
        private Map _lineItems;
```

**Associations**

- **Association Classes**
  - Add attributes, operations, and other features to associations – one instance from each side!

```
Person employer
  0..1
company
```

**Associations**

- **An Illegal Association Class (Although it seems fine)**

```
Person
  0..1
company
```

**Associations**

- **Use Stereotypes**
  - <<history>>

```
class Person {
    Company getEmployer();
    Company getEmployer(Date)
    void changeEmployer(Company newEmployer, Date changeDate);
    void leaveEmployer (Date changeDate);
```
**Parameterized Class**

```cpp
template <class numtype>
class number {
    private:
        numtype n;
    public:
        number () { n = 0; }
        void get_number() { cin >> n; }
};
```

**Parameterized Classes**

```cpp
Set<Employee> or
```

**Visibility**

Different languages have different rules in "private", "public", and "protected"!!

- **C++**
  - Public Member: visible to anywhere.
  - Private Member: used only by the class that defines it.
  - Protected Member: (a) used only by the class that defines it or (b) a subclass of that class.

- **C#**
  - Public: visible to anywhere.
  - Private: visible only within the class.
  - Protected: visible within the class and its subclasses.

**Contents**

- Introduction
- Development Process
- Use Cases
- Class Diagrams & Object Diagrams
- Interaction Diagrams
- Packages and Collaborations
- State Diagrams, Activity Diagrams, Physical Diagrams
- Misc

**Package Diagrams**

- How do you break down a large system into smaller systems?
- Functional Decomposition
  - Separation of functions and data
- UML Package Diagrams
  - A OO Grouping Mechanism
**Package Diagrams**

- Why Changes Propagate?
  - One class sends messages to another.
  - One class has another as part of its data.
  - One mentions another as a parameter to an operation.

  Interface changes!

**Package Diagrams**

- How to minimize dependencies?
  - UML Dependency vs Compilation Dependency?
  - Why no transitivity?
    - Similar to a layered architecture!
  - Example: Package Diagram

**Package Diagrams**

- Class Types inside a Package:
  - Private, Public, or Protected
  - Sharing/Dependency of the Public Methods of Public Classes!
  - Reducing of the Interface of a Package
  - Facades

- Another Objective for Package Diagrams:
  - Help to see what dependencies are!

**Package Diagrams**

- Containment of Packages or Classes
  - Key Classes
  - Dependency of Packages That Contain Sub-packages
  - Summaries of Low-Level Dependency

**Package Diagrams**

- Collaboration
  - The interaction among two or more classes
  - Show the implementation of an operation or the realization of a use case.
  - May include class diagrams and interaction diagrams
  - Use for classes inside a package or common behavior across packages
Package Diagrams

- Parameterizing of Collaboration
- Same collaboration for different classes
- Roles (Collaboration) vs Classes
- Pattern
- Example: Collaboration for Sale

Package Diagrams

Parameterized Collaboration for Sale
- Buyer
- Seller
- Offer
- Lot
- Sale
- buyer, seller, lot, offer

When to Use Package Diagrams
- Whenever a class diagram that encompasses the whole system is no longer legible on A4 sheet of paper!
- Useful for testing as well!
- When to use Collaboration?
- Refer to a particular interaction
- Parameterized collaboration

Contents

- Introduction
- Development Process
- Use Cases
- Class Diagrams & Object Diagrams
- Interaction Diagrams
- Packages and Collaborations
- State Diagrams, Activity Diagrams, Physical Diagrams
- Misc

State Diagrams

- Why State Diagrams?
- Describe the behavior of a system.
- Describe all of the possible states that a particular object can get into and how the object’s state changes as a result of events that reach the object.
- Based on statechart by David Harel (1987)
State Diagrams

Syntax
- Event [Guard] / Action
- do / activity
- Start point

Action
- Be associated with a transition
- Occur quickly and is not interruptible.

Activity
- Be associated with a state
- May be interrupted by some event.

Guard
- A logical condition
- Guards from the same state should be mutually exclusive
- A transition should occur as soon as the corresponding event happens.

A state could have no activity!

Superstate

An State Diagram Example

Examine
- do / check
- item

Send
- do / initiate
- delivery

Wait
- Deliver

Begin
- / get first item
  - Not all items checked
  - all items available
  - some items not in stock

/ get next item
  - [All items checked &&
    - all items available]
  - [All items checked &&
    - some items not in stock]

Item Received
- [some items not
  - in stock]
- [all items
  - available]
  - deliver

Activity
  - cancelled

An State Diagram Example - Superstate

Examine
- do / check
- item

Send
- do / initiate
- delivery

Wait
- Deliver

Begin
- / get first item
  - Not all items checked
  - all items available
  - some items not in stock

/ get next item
  - [All items checked &&
    - all items available]
  - [All items checked &&
    - some items not in stock]

Item Received
- [some items not
  - in stock]
- [all items
  - available]
  - deliver

Activity
  - cancelled

Other Event Types
- No-state transition event
  - eventName / actionName
- After Event
  - after (20 minutes)
- When Event
  - when (temperature > 100 degrees)
- Entry/Exit Event
**State Diagrams**

Example: Payment authorization

- Authorizing
- Checking
- Waiting
- Dispatching
- Authorized
- Delivered
- Rejected
- [payment not ok]
- [payment ok]

Concurrent State Diagrams

- More than one state at any point
- Good when a given object has sets of independent behaviors

When to Use State Diagrams?
- Describe the behavior of an object across several use cases.
- Use interaction diagrams or activity diagrams if needed.
- Only classed exhibiting interesting behavior!
- Do not draw state diagrams for every class!

**Contents**

- Introduction
- Development Process
- Use Cases
- Class Diagrams & Object Diagrams
- Interaction Diagrams
- Packages and Collaborations
- State Diagrams, Activity Diagrams, Physical Diagrams
- Misc

**Activity Diagrams**

Why Activity Diagrams?
- Useful in connection with workflow and in describing behavior that has a lot of parallel processing.
- Describing the sequencing of activities with support for both conditional and parallel behavior.
- Origin:
  - Event diagrams (Jim Odell), state modeling techniques, workflow modeling, Petri nets.

Core Symbols
- Activity State or Activity
- Conditional Behavior
- Branch
  - Exclusive on “Transitions”
  - Guard [condition], e.g., [else]
- Merge
  - Marks the end of conditional behavior started by a branch.
Activity Diagrams

Parallel Behavior
Fork
Interleaving semantics
Sequence of "parallel" activities is irrelevant!
Difference from flowchart
Limited to sequential processes in flowcharts
Join
The outgoing transition is taken only if all of the states on the incoming transitions have completed their activities.

Why Parallelism?
Improving the efficiency and responsiveness of business processes
Remove unnecessary sequence and spot opportunities for parallelism.
Forks and joins must match except:
1. Threads fork threads
2. Notational shorthand to remove clutter from the diagram.
3. Sync state and conditional threads

Decomposing an Activity
Break an activity down into subactivities.
The explicit start and end states are good for the usage of the subactivity in other contexts.
Dynamic Concurrency
Multiplicity Marker *

When to use activity diagrams?
Analyze a use case
Action and behavior dependency
Understand workflow
Describe a complicated sequential algorithm
Deal with multi-threaded applications
Good for considering parallel behavior or multi-threaded programming.
### Activity Diagrams

- **When not to use them?**
- **Try to see how objects collaborate**
- **Interaction diagrams**
- **Try to see how an object behaves over its lifetime**
- **State diagrams**
- **Represent complex conditional logic**
- **Truth tables**

**Disadvantage:**

- No link among actions and objects

### Contents

- **Introduction**
- **Development Process**
- **Use Cases**
- **Class Diagrams & Object Diagrams**
- **Interaction Diagrams**
- **Packages and Collaborations**
- **State Diagrams, Activity Diagrams, Physical Diagrams**
- **Misc**

### Physical Diagrams

- **Physical Diagrams**
- **Deployment Diagrams**
  - Show the physical relationships among software and hardware components in the delivered system.
- **Component Diagrams**
  - Show the various components in a system and their dependencies.

### Deployment Diagrams

- **Purpose:**
  - Show how components and objects are routed and move around a distributed system.
- **Node**
  - Some kind of computational unit, e.g., a piece of hardware
- **Connection**
  - Communication paths over which the system will interact.

### Component Diagrams

- **Purpose:**
  - Show the various components in a system and their dependencies.
- **Component**
  - A physical module of code
  - A class might appear in several components
- **Dependency**
  - Communication and compilation
**Physical Diagrams**

- Combining of Component and Deployment Diagrams
- Show which components run on which nodes!
- When to use them?
- Show the physical information of the system!

**Contents**

- Introduction
- Development Process
- Use Cases
- Class Diagrams & Object Diagrams
- Interaction Diagrams
- Packages and Collaborations
- State Diagrams, Activity Diagrams, Physical Diagrams

**A Case Study – Buy a Product**

**Buy a Product – Use-Case Text:**
1. Customer browses the catalog and selects items to buy.
2. Customer goes to check out.
3. Customer fills up in the shipping information.
4. System presents full pricing information, including shipping.
5. Customer fills in credit card information.
7. System confirms sale immediately.
8. System sends confirming email to customer.

**Alternative: Regular Customer**

- 3.a system display current shipping information, pricing information, and last digits of credit card information
- 3.b Customer accept or override these defaults
- Return to Step 6!

**A Use Case Diagram**

- Web-based On-Line Store

**A Class-Diagram**

**A Sequence Diagram**