A Tutorial on OO Modeling using UML

Robert B. France
Dept. of Computer Science
Colorado State University
USA
france@cs.colostate.edu

If software developers built pyramids ...
Modeling and software engineering

- SE is “the discipline of resolving problems with software solutions”. (B. Blum).
- Focus is on solving complex problems.
- Problem solving involves analyzing problems and synthesizing solutions.
- Rigorous software modeling is concerned with imposing structure on problems and solutions to facilitate rigorous analysis and synthesis.

What software engineering concern does modeling address?

The problem-implementation gap

- When the problem-implementation gap is wide significant effort is required to transform problem-level abstractions to programs
  - Manually bridging the gap introduces significant accidental complexities
- Conventional wisdom: raise the level of abstraction at which programs are written
The nature of the software development crisis is evolving

Model-Driven Engineering: A software engineering perspective

- MDE is concerned with reducing accidental complexities associated with bridging wide abstraction gaps
- through use of technologies that support rigorous transformation of abstractions to software implementations

MDE is concerned with developing software tools to support the work of software developers

Use of MDE techniques helps make a software developer a software engineer
What is a model?

A description of an aspect of a software-based system that may or may not exist. A model is created to serve one or more purposes.

Modeling should be purpose-driven

- Modeling to explore problem or solution spaces
  - Models as (informal) sketches
  - Models as formally analyzable artifacts
- Modeling to communicate aspects of problems or solutions
  - Models as documentation artifacts
- Modeling to generate code
  - Models as abstract programs
Modeling aptitude

- Why Johnny can’t model and Jane can
- Hypothesis: A good modeler is a good programmer; a good programmer is not necessarily a good modeler
- Modeling requires programming and abstraction skills
  - Abstraction skills refine programming language skills
  - Programs produced by developers with good abstraction skills should be of significantly better quality

Early focus of MDE

- Models as documentation/communication artifacts
  - Early work on CASE technologies laid the foundation
- Models as artifacts for generating implementation and deployment artifacts
  - OMG’s work on UML and MDA
  - Separation of Platform Independent and Platform Specific abstractions
  - Software development viewed as a transformative process
    - Model transformations are key enablers of MDA
Evolution of MDE

- Breathing life into models
  - Models as analyzable artifacts
  - Foundation provided by work on formal specification techniques
- Supporting exploratory development through compositional modeling
  - Making agile development rigorous
- Models as artifacts for managing and evolving runtime behavior
  - models@run.time
- Supporting domain-specific software development

The Unified Process

Validation vs. Verification
The Unified Process
Validation vs. Verification

- Validation is concerned with establishing that a design or an implementation satisfies users stated and unstated requirements
  - Are we building the right thing?
- Verification is concerned with establishing that a development artifact (e.g., design or code) satisfies a specification (e.g., a requirements or design formal specifications)
  - Did we build it right?
The Unified Process (UP)

- The Unified Process is an industry standard software engineering process
  - It is the generic process for the UML
  - It is free - described in "The Unified Software Development Process", ISBN:0201571692"
- UP is:
  - Use case (requirements) driven
  - Iterative and incremental
- UP is a generic software engineering process. It has to be customised (instantiated) for your project
  - In house standards, document templates, tools, databases, lifecycle modifications, …
- Rational Unified Process (RUP) is an instantiation of UP
  - RUP is a product marketed and owned by IBM
  - RUP also has to be instantiated for your project

UP basics

- Small steps, feedback and evolution
- Iterative, incremental, time-boxed
- Risk-driven

![Iteration Diagram](image-url)
Key Practices

- Deliver product in increments developed in iterations
- Iterations are carried out in a fixed time
  - Developers can choose to drop features but should not extend iteration
- High risk and high value aspects tackled in early iterations
  - Cohesive architecture implemented in early iterations
- Customers continuously engaged in evaluation, feedback and requirements elicitation
- Continuously verify quality; test-driven code development
- Model software

Motivating time-boxing

- “Work expands so as to fill the time available for completion” (Parkinson’s Law)
- Forces prioritization of tasks and risks
- Gain confidence of customers
- Build team confidence/satisfaction
Iteration structure

- **Requirements phase**
  - Conceive, Plan, Requirements, etc.

- **Analytic phase**
  - Analyze

- **Design phase**
  - Design

- **Implement phase**
  - Test

- **Iterative phase**

- **Development cycles**

Baselines and increments

- Each iteration generates a baseline
- A baseline is a set of reviewed and approved artefacts that:
  - Provide an agreed basis for further review and development
  - Can be changed only through formal procedures such as configuration and change management
- An *increment* is the difference between the baseline generated by one iteration and the baseline generated by the next iteration
  - This is why the UP is called “iterative and incremental”
UP structure

- Each phase can include several iterations
  - The exact number of iterations per phase depends on the size of the project! e.g. one iteration per phase for small projects
- Each phase concludes with a major milestone

The UP Phases

Note: The phases are not iterations

Core Process Workflows
- Business Modeling
- Requirements
- Analysis & Design
- Implementation
- Test
- Deployment

Core Supporting Workflows
- Configuration & Change Mgmt
- Project Management
- Environment
Phases

- Inception
  - Early exploration of problem to determine project feasibility
  - What's the perceived business value?
  - What are the risks?

- Elaboration
  - Requirements detailing (major requirements identified)
  - Iterative implementation of "core" architecture

- Construction
  - Iterative development of remaining low-risk elements
  - Prepare for deployment

- Transition
  - Beta tests
  - Deployment

Modeling Requirements

Requirements Engineering
Requirements Class Modeling
Use Case Modeling
Problem vs. Solution views of software

- A problem targeted by software can be expressed as:
  - A difficulty the users or customers are facing,
  - Or as an opportunity that will result in some benefit such as improved productivity or sales.
- Requirements documents describe problems

- The solution to the problem entails developing software
  - Software designs and their implementations in source code describe solutions

What is a software requirement?

- A requirement is a statement about the proposed software that all stakeholders agree must be made true in order for the customer’s problem to be adequately solved.
  - Says something about what the software does and what data it maintains
  - All the stakeholders have agreed that it is valid
- A collection of requirements is a requirements document (e.g., the SRS in UP)
Overview

- Requirements engineering
- Class modeling
- Basic use case modeling
- Advanced use case modeling
Requirements Engineering activities

- **Elicitation**: extracting requirements from customers and users
- **Analysis**: analyzing and organizing requirements to gain deeper understanding
  - Results of analysis reflected in models: Use Cases, Requirements Class Model
- **Specification**: detailed documentation of required behavior
- **Validation**: requirements are validated against customer/user needs
- **Requirements management**: activities related to documenting, controlling and tracking requirements and changes to requirements

Broad categories of requirements

Requirements can be broadly categorized as

- **Functional requirements**
  - Describe what the system should do
- **Non-functional requirements** (aka **system attributes**)
  - *Constraints* that must be adhered to during development
  - Examples: the software must restrict access to sensitive information; the software must deliver a response within a time interval after a particular event has occurred; the software must be usable by persons who are not computer literate.
Analyzing functional requirements

- Requirements models used to analyze and structure requirements
  - Models are used to impose structure on requirements so that they can better understood and to uncover problems in requirements
    - The act of building models can reveal problems related to inconsistent, incomplete and ambiguous requirements
  - Class model: requirements expressed in terms of problem (not solution!) concepts and their relationships
  - Use cases: required behavior expressed in terms of users interacting with the system (treated as a black-box) to accomplish goals

Documenting functional requirements

- Describing functional requirements as list of functions (aka, features)
  - Preferred way of presenting requirements to users and customers
  - Features in list are prioritized (e.g., necessary, desirable, cosmetic)
  - Feature list used to focus activities in UP iterations
Functional requirements categories

- System functions: Services provided by software system
- Function categories
  - Evident: user-observable services that must be provided
  - Hidden: services that must be provided that are not visible to users (e.g., logging transactions in a banking system)
  - Frill: Optional services that enhance use of system

Function documentation example

<table>
<thead>
<tr>
<th>Ref #</th>
<th>Function</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1.1</td>
<td>Handle cash payments, capturing amount tendered and calculating balance due</td>
<td>evident</td>
</tr>
<tr>
<td>R1.2</td>
<td>Log credit payments</td>
<td>hidden</td>
</tr>
</tbody>
</table>
System Attributes (Non-Functional Requirements - NFRs)

Three major types

- Categories reflecting: usability, efficiency, reliability, maintainability and reusability:
  - Response time, Throughput, Resource usage, Reliability, Availability, Recovery from failure, Allowances for maintainability and enhancement, Allowances for reusability

- Categories constraining the environment and technology of the software.
  - Platform
  - Technology to be used

- Categories constraining the project plan and development methods
  - Development process (methodology) to be used
  - Cost and delivery date
    - Often put in contract or project plan instead

Documenting attributes of system functions

<table>
<thead>
<tr>
<th>Ref #</th>
<th>Function</th>
<th>Cat.</th>
<th>Attribute</th>
<th>Details</th>
<th>Cat.</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0.9</td>
<td>Display desc. and item price</td>
<td>evident</td>
<td>Response time</td>
<td>5 secs. max</td>
<td>must</td>
</tr>
<tr>
<td></td>
<td></td>
<td>interface</td>
<td>Forms based graphical</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R1.2</td>
<td>Log credit payments</td>
<td>hidden</td>
<td>Fault tolerance</td>
<td>Must log within 24 hrs even in power failure</td>
<td>must</td>
</tr>
</tbody>
</table>
Modeling requirements

- Describe required *functionality* (behavior) using *use cases*
  - Use case diagrams simply identify the use cases and the actors that interact with them
  - Use cases can be described textually or using *activity diagrams*
- Describe imposed *structure* on problem concepts using *class models*

Basic requirements specification structure

- Textual requirements description
  - Requirements glossary/dictionary
  - Functional requirements description
  - NFR description
- Requirements models
  - Requirements class model
  - Requirements use cases
Models vs. Diagrams

- A diagram is a view on a model
  - A diagram does not necessarily display all information in a model
  - A class diagram may only show some of the classes, associations, attributes and operations included in a model
  - A diagram is the same as a model if it displays all the information in the model.

Requirements Class Models

Modeling problem concepts and their relationships
What is a class? Ans: It depends …

- A class is a description of a set of objects that share the same properties (expressed as attributes and relationships)
  - At the requirements level a class describes a concept in the problem domain
  - At the design level a class describes a concept in the solution domain
  - At the programming level a class defines objects that will perform computations
- An object is a concept, abstraction, or thing with identity that has meaning for an application.
  - An object is an instance of a class
  - Each object “knows” its class

What is a requirements class model?

- Syntactically, a class model is a structure of classes.
- A requirements class model describes problem concepts and their relationships
  - It describes the structure that requirements engineers impose on a problem/domain to better understand the problem
Requirements vs. Design vs. Program Class Models

- Syntactically, a class model is a structure of classes.
- What a class model represents depends on whether it is used to describe requirements, logical designs or programs.
  - A **requirements class model** describes problem concepts and their relationships.
  - A **design class model** describes solution concepts and their relationships.
  - A **program (or implementation) class model** describes program-level classes (e.g., Java classes) and their programming language specific relationships.
- It is good practice to qualify the term “class model” with a descriptor that determines the level of abstraction at which the concepts it captures are described.
  - e.g., a class model at the requirements level should be referred to as a “requirements class model” rather than just a “class model.”

An example of a requirements class diagram

**Diagram vs. model:** A diagram can be a partial view of a model; large models can be described using multiple diagrams.
Structure of a class

- In general, a class has the following structure:
  - Name compartment (mandatory)
  - Attributes compartment (optional)
  - Operations compartment (optional; not included in requirements class models – see below)

- Every class must have a unique name.
- An object of a class must have values associated with each attribute of the class

**Prof. France’s requirements class restriction**

- A requirements class must not contain operations
  - Assigning operations to classes is often a design decision
  - i.e., there is often a number of alternative ways in which functionality can be distributed across classes; determining what aspects of required behavior a class is responsible for is an important design activity
  - Required behavior is described by use cases, not by class operations at the requirements level
  - No visibility qualifiers (e.g., public, private) are associated with attributes (best left to design)

Style guidelines for requirements classes

- Center class name.
- Capitalize the first letter of class names.
- Left justify attributes in plain face.
- Begin attribute names with a lowercase letter.
- Put the class name in italics if the class is *abstract*.
  - An abstract class is one whose instances must be instances of a specialized class
  - At the implementation level, this translates to a class that cannot be instantiated
- Show full attribute information when needed and suppress them in other contexts or when merely referring to a class.
Depicting requirements classes: Do’s and Don’t’s

Remember - No Operations!

Attributes

- An attribute is a named property.
  - Each class instance associates value(s) with each attribute of a class.
- What should be modeled as an attribute in a requirements class?
  - Properties of problem concepts that we want to treat as primitive (i.e., not decomposable further into an entity with other attributes)
Attributes: Do’s & Don’t’s

<table>
<thead>
<tr>
<th>Person</th>
<th>Person addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bad due to a plural attribute</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Person</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>0..1</td>
</tr>
<tr>
<td></td>
<td>street</td>
</tr>
<tr>
<td></td>
<td>municipality</td>
</tr>
<tr>
<td></td>
<td>provOrState</td>
</tr>
<tr>
<td></td>
<td>country</td>
</tr>
<tr>
<td></td>
<td>postalCode</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address type</th>
</tr>
</thead>
<tbody>
<tr>
<td>street</td>
</tr>
<tr>
<td>municipality</td>
</tr>
<tr>
<td>provOrState</td>
</tr>
<tr>
<td>country</td>
</tr>
<tr>
<td>postalCode</td>
</tr>
</tbody>
</table>

Good solution. The type indicates whether it is a home address, business address etc.

Another Prof. France rule: Do not use attributes to relate concepts! Use relationships instead

Bad practice: Customer rentedVideos: List of Video

What information is lost in the above diagram that is more evident in the diagram below?

Better: Customer rents 1..* Video
Modeling static class relationships

- Two kinds of static relationships:
  - **Associations**
    - Represent structural relationships among problem concepts
    - An association specifies a collection of links, where a link is a physical or conceptual connection among objects (an object is an instance of a class).
  - **Generalizations**
    - Represent generalization/specialization class structures
  - The two kinds of relationships are orthogonal

Associations & Multiplicities

- A customer can have 0 or more rented videos.
- A video can be rented by at most one customer
Association Roles

- When a class is part of an association its objects that are linked via the association play a role in the relationship.
- You can name the role that an object plays in an association by placing the name at the class’s association end.
- Formally, a class role is the set of objects that are linked via the association.

```
association roles

Person

project leader

1

project member

1..* 0..1

Project

managed project

assigned project

Bob Bill Keisha Jane

proj1 proj2 proj3

Bob Bill Keisha Jane

proj1 proj2 proj3

project leader = {Bob, Keisha, Jane}
managed project = {proj1, proj2, proj3}
project member= {Bob, Bill, Keisha}
assigned project = {proj1, proj2, proj3}
```
Prof. France’s Requirements Class Modeling Rules

- All association ends must have multiplicities
  - There is a default, but most people do not remember it; better to be explicit to avoid confusing the reader

- All associations must be named (either an association name or a role name at each end; role names at each end are preferred because they can be more informative)

More Modeling Rules

- Do not put navigation arrows on associations in requirements class diagrams
  - Determining which class is visible to another class is a design decision (also see below)

- Do not put operations in requirements class diagrams
  - There can be more than one way to assign responsibilities to a class; this is a design decision
  - To avoid making design decisions too early, novice modelers should not include operations in requirements class models
An initial Requirements Class Diagram of a Video Rental System. Can you spot any problems?

Association classes

- Sometimes, an attribute that concerns two associated classes cannot be placed in either of the classes

Challenge: How do you model the above without an association class?
Reflexive associations

- It is possible for an association to connect a class to itself

UML forms of Aggregation

- Composition (strong aggregation)
  - parts are existent-dependent on the whole
  - parts are generated at the same time, before, or after the whole is created (depending on cardinality at whole end) and parts are deleted before or at the same time the whole dies
  - multiplicity at whole end must be 1 or 0..1
- (weak) Aggregation
Guidelines to Identifying Associations

- Focus on associations for which knowledge of the relationship must be preserved for some duration
  - An association should exist if a class
    - possesses
    - controls
    - is connected to
    - is related to
    - is a part of
    - has as parts
    - is a member of, or
    - has as members
  - another class
- More important to identify concepts than associations
  - Too many associations can lead to confusing models
- Avoid showing redundant/derivable associations
Actions versus associations

- A common mistake is to represent actions as associations
  - Do not model dynamic relationships in a class model

```
<table>
<thead>
<tr>
<th>LibraryPatron</th>
<th>CollectionItem</th>
</tr>
</thead>
<tbody>
<tr>
<td>borrow</td>
<td>borrow</td>
</tr>
<tr>
<td>return</td>
<td>returned</td>
</tr>
</tbody>
</table>

Bad, due to the use of associations that are actions

Better: The **borrow** operation creates a **Loan**, and the **return** operation sets the **returnedDate** attribute.

Generalization/Specialization

- A generalization (or specialization) is a relationship between a general concept and its specializations.
  - Objects of specializations can be used anywhere an object of a generalization is expected (but not vice versa).

- Example: *Mechanical Engineer* and *Aeronautical Engineer* are specializations of *Engineer*
Generalization

- Shape
  - Polygon
  - Ellipse
  - Spline

  Separate Target Style

- Shape
  - Polygon
  - Ellipse
  - Spline

  Shared Target Style

Generalization

- Vehicle
  - WindPowered Vehicle
  - MotorPowered Vehicle
  - Land Vehicle
  - Water Vehicle

  - Truck
  - Sailboat

  {overlapping}

  power

  venue

  {overlapping}
Avoiding having instances change class

- An instance should never need to change class

![Class Diagram]

Identifying generalizations and interfaces

- There are two ways to identify generalizations:
  - bottom-up
    - Group together similar classes creating a new superclass
  - top-down
    - Look for more general classes first, specialize them if needed
Handling large requirements models

- Use packages to provide views of large domain models.
- Developers may not have to draw package boxes around groups as in this example. Rather, a CASE tool will allow “drill down”.

Object diagrams

- An object diagram describes a configuration of objects.
- One can view a class diagram as specifying a set of system states modeled by object configurations.
  - A state (object configuration) that satisfies the constraints expressed in a class model is said to be valid.
Valid vs. Invalid states

- **Pat:** Employee
- **Wayne:** Employee
- **Ali:** Employee
- **OOCorp:** Company
- **OOCorp’s Board:**
- **Carla:** Employee
- **UML inc:** Company
- **UML inc’s Board:**
- **Terry:** Employee

Associations versus generalizations in object diagrams

- **Associations** describe the relationships (links) that will exist between objects at run time.
  - When you show an object diagram generated from a class diagram, there will be an object for each associated class joined by a link that is an instance of the association.

- **Generalizations** describe relationships between classes in class diagrams.
  - They do not appear in object diagrams at all.
  - An instance of any class should also be considered to be an instance of each of that class’s superclasses.
Use Cases

What is use case modeling?

- Use case model: a view of a system that emphasizes the behavior as it appears to outside users.
- A use case model partitions system functionality into transactions (‘use cases’) that are meaningful to users (‘actors’).
Use Cases in UML 2.x

- Use cases are associated with subjects
  - A subject can be a system, a subsystem in a system, or a class
- A use case describes interactions between users (clients) and a subject
- At the requirements level the subject is the system under development

Use Case Modeling: Core Elements

<table>
<thead>
<tr>
<th>Construct</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>use case</td>
<td>A sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system.</td>
<td><img src="image" alt="UseCase" /></td>
</tr>
<tr>
<td>actor</td>
<td>A role played by an entity that interacts with the subject (e.g., system, subsystem, class).</td>
<td><img src="image" alt="Actor" /></td>
</tr>
<tr>
<td>System/subject boundary</td>
<td>Represents the boundary between the subject and the actors who interact with the subject.</td>
<td><img src="image" alt="Boundary" /></td>
</tr>
</tbody>
</table>
Depicting actors

Some UML alternatives to illustrate external actors that are other computer systems. The class box style can be used for any actor, computer or human. Using it for computer actors provides visual distinction.

A use case diagram
Simple Use Case example

Actor Inputs
1. Customer submits identification information
2. Customer submits items to be rented
3. Customer pays.

System Response
1. If customer is authenticated, then request rental items
2. Display calculated price.
3. Inform customer that payments is authorized
### Use Cases as requirements

- Use cases can be used to capture functional requirements
- Not all requirements can be captured by use cases
  - Some information on system attributes can be included in a use case if it is directly applicable to the functionality described by the use case
  - System attributes that span use cases are documented as supplementary requirements
Actor types

- **Primary**: actor whose goal is accomplished by the use case
- **Supporting**: actor that provides services to the system
  - E.g., authorization service
- **Offstage**: an actor that has an interest in the use case but is not primary or supporting
  - E.g., regulating agency

Use Case instance (scenario)

- A **scenario** is a particular sequence of actions in a use case.
  - A use case is a related set of scenarios that yields an observable result of value to a particular actor
- A **use case instance** is an execution of a scenario.
  - Often use case instance and scenario are used synonymously in informal discussions.
Levels of rigor

- **Brief**: One paragraph summaries of functionality
- **Casual**: Multiple paragraphs that cover multiple scenarios
- **Fully-dressed** (Detailed): Structured, detailed description of scenarios

Essential vs. Concrete Use Cases

- **Essential Use Cases** describe functionality in implementation independent terms
  - Requirements level use cases must be essential
- **Concrete Use Cases** describe external functionality in system dependent terms
  - Use cases can be used during design to document externally observable behavior of subsystems
### Requirements Use Case template

<table>
<thead>
<tr>
<th><strong>Use Case Number:</strong></th>
<th>EU-xxxx : Indicates an essential use case, i.e., a use case that describes activity in system independent terms</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use Case Name:</strong></td>
<td>Enter name of Use Case.</td>
</tr>
<tr>
<td><strong>Overview:</strong></td>
<td>Describe the purpose of the Use Case and give a brief description.</td>
</tr>
<tr>
<td><strong>Type:</strong></td>
<td>Enter Use Case priority (primary, secondary, optional)</td>
</tr>
<tr>
<td><strong>Actors:</strong></td>
<td>List all actors that participate in this Use Case. Indicate the actor that initiates the use case by placing &quot;initiator&quot; in brackets after the actor name. Also, indicate primary actors by placing &quot;primary&quot; in brackets after actor name.</td>
</tr>
<tr>
<td><strong>Properties:</strong></td>
<td><strong>Performance:</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Security:</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Other:</strong></td>
</tr>
</tbody>
</table>

### Analysis Use Case template – cont’d

<table>
<thead>
<tr>
<th><strong>Precondition:</strong></th>
<th>Enter the condition that must be true when the main flow is initiated. This should reference the conceptual model.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Flow:</strong></td>
<td><strong>Main Flow:</strong> Steps should be numbered.</td>
</tr>
<tr>
<td></td>
<td><strong>Subflows:</strong> Break down of main flow steps.</td>
</tr>
<tr>
<td></td>
<td><strong>Alternate Flows:</strong> Include the post condition for each alternate flow if different from the main flow.</td>
</tr>
<tr>
<td><strong>Post Condition:</strong></td>
<td>Enter the condition that must be true when the main flow is completed. This should reference the conceptual model. Include the following information in this section:</td>
</tr>
<tr>
<td><strong>Cross References:</strong></td>
<td>References to other Use Cases or textual requirements that relate to this Use Case.</td>
</tr>
</tbody>
</table>
Goals and Use Cases

- Interactions usually take place to satisfy system goals.
- Identifying goals is important because it can lead to consideration of more effective alternatives.
- For each interaction ask “why?” - the answer should lead to a system goal.

Determining Use Case scope

- A use case should describe end-to-end functionality
  - Should describe a task carried in response to an event generated by an actor that produces a result of value to a subset of its actors and leaves the system in a stable state (one in which it is not waiting for a restricted set of inputs)
Identifying Use Cases and Actors

- Approaches to identifying use cases
  - Actor-first: Identify actors first and consider the ways they interact with the system
  - Operation-first: Identify system-level operations and then identify actors that interact with operations
  - Event-first: Identify external events and develop use cases that handle the events

Developing Use Cases

- Scope system and identify primary actors that interact with the system
- Determine goals of primary actors (can be documented in an Actor-Goal list)
- For each actor, consider the ways that the actor typically interacts with the system to accomplish goals
- Consider exceptional behaviors
Use Case Modeling tips

- Writing essential use cases: Focus on intent
  - Keep user interface terms out
  - Ask “what is the goal?”
- Write “black-box” use cases
  - Do not describe internal operations (e.g., storing to a database)
- Focus only on interactions between system and actors
  - Ignore interactions between actors
- Focus on text description
  - Use diagrams for presentation purposes only
- A use case diagram should
  - contain only use cases at the same level of abstraction
  - include only required actors

How do I know I have a good use case?

- Use case should describe an activity that yields an observable result of value to an actor.
- A use case can describe an elementary business process: a sequence of tasks performed to handle a business event
- Use cases are typically not single steps or single low-level actions.
Requirements – advanced use case modeling

Use Case Relationships

Relating Use Cases

- Specializing/generalizing use cases
- Including use cases
- Extending use cases
### Use Case Modeling: Core Relationships

<table>
<thead>
<tr>
<th>Construct</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>association</td>
<td>The participation of an actor in a use case, i.e., instance of an actor and instances of a use case communicate with each other.</td>
<td></td>
</tr>
<tr>
<td>generalization</td>
<td>A taxonomic relationship between a more general use case and a more specific use case.</td>
<td></td>
</tr>
<tr>
<td>extend</td>
<td>A relationship from an extension use case to a base use case, specifying how the behavior for the extension use case augments (subject to conditions in the extension) the behavior defined for the base use case. The base use case does not depend on the extension use case. Compare: include.</td>
<td>&lt;&lt;extend&gt;&gt;</td>
</tr>
<tr>
<td>include</td>
<td>A relationship from a base use case to an inclusion use case, specifying how the behavior for the base use case contains the behavior defined for the inclusion use case. Compare: extend.</td>
<td>&lt;&lt;include&gt;&gt;</td>
</tr>
</tbody>
</table>

### Use Case Relationships

![Use Case Relationships Diagram](image.png)

- **Place Order**: Extension points: additional requests after creation of the order
- **Request Catalog**: the salesperson asks for the catalog
- **Order Product**: «include»
- **Supply Customer Data**: «include»
- **Arrange Payment**: «Include»
- **Order Product**: «include»

---

Fig. R-54, UML Notation Guide

© Robert B. France
Actor generalization - example

- The Customer and the Sales Agent actors are very similar.
- They both interact with List products, Order products, Accept payment.
- Additionally, the Sales Agent interacts with Calculate commission.
- Our diagram is a mess – can we simplify it?

Actor generalisation

- If two actors communicate with the same set of use cases in the same way, then we can express this as a generalisation to another (possibly abstract) actor.
- The descendents actors inherit the roles and relationships to use cases held by the ancestor actor.
- We can substitute a descendental actor anywhere the ancestor actor is expected. This is the substitutability principle.
Use case generalisation

<table>
<thead>
<tr>
<th>Use case generalization semantics</th>
<th>Inherit</th>
<th>Add</th>
<th>Override</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relationship</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Extension point</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Precondition</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Postcondition</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Step in main flow</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Alternative flow</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Sales system

- FindProduct
  - FindBook
  - FindCD

Including Use Cases

- A base use case can include another use case at a specified location.
  - Used to avoid writing the same flow of events across a number of use cases.
  - The base use case is an incomplete description of functionality; it requires the included use cases to complete its description.
  - The included use case can be a complete or incomplete use case.
The base use case executes until the point of inclusion:

```java
include(InclusionUseCase)
```

- Control passes to the inclusion use case which executes
- When the inclusion use case is finished, control passes back to the base use case which finishes execution

---

**Extending Use Cases**

- A use case can extend a base use case by incorporating additional behavior at specified locations of the base use case.
  - The base use case can act as a stand-alone use case.
  - The base use case can only be extended at specified points called *extension points*.
  - Often used to separate optional behavior from mandatory behavior.
  - Also used to model a separate flow that is executed under certain conditions.
Example: Online HR System

Online HR System

- Locate Employees
- Update Employee Profile
- Update Benefits
- Access Travel System
- Access Pay Records

Employee

Manager

Healthcare Plan System

Insurance Plan System

[if currentMonth = Oct.]

(readOnly)
Online HR System: Use Case relationships

Online HR System: Update Benefits Use Case

- **Actors**: employee, healthcare plan system, insurance plan system

- **Precondition**:
  - Employee has logged on to the system and selected ‘update benefits’ option

- **Basic course**
  - System displays employee account
  - System asks employee to select medical plan type; include (Update Medical Plan).
  - System asks employee to select dental plan type; include (Update Dental Plan).
  - ...
  - System asks user to select benefits options: benefit options
    - reimbursement option selected: Elect Reimbursement for Healthcare
    - stock option selected: Elect Stock Purchase
Role of requirements use cases in development

- Functional requirements are primarily recorded in essential use cases
- The scope of an iteration is usually expressed in terms of use case scenarios covered
- A design should include realizations of each requirements use case
  - Use cases realized as collaborating objects in designs

Evolution of Use Cases

- Inception
  - Most interesting and complex or risky use cases written in brief format
  - 10-20% of core complex functions rewritten in fully-dressed format
- Elaboration
  - In each iteration, use cases are prioritized and high priority use cases are developed in fully-dressed format
  - 80-90% completed by end of elaboration
- Construction
  - Remaining (minor) use cases are developed
The benefits of basing software development on use cases

- They can help to define the scope of the system
- They are often used to plan the development process
- They can be used to both develop and validate the requirements
- They can form the basis for the definition of test cases
- They can be used to structure user manuals
- They can be used during requirements to prototype user interfaces and during design to design user interfaces

Use cases must not be seen as a panacea

- The use cases must be validated
- There are some aspects of development that are not covered by use case analysis.
  - Non-functional requirements are often not covered
  - Functionality that is not triggered by actors is not covered
    - E.g., auditing transactions in a banking system
- There is a tendency to write use cases in terms of how a system works currently; this can bias design and make it less likely that designers will consider innovative solutions
Specializing Use Cases

- Generalizing/specializing use cases
  - A specialized use case inherits the behavior (sequences of actions) of its parent(s).
  - A specialized use case can override some of the behavior of its parent(s). It can also add to the behavior.
  - A specialized use case can be used anywhere the general use case is expected.

Activity modeling

Using activity diagrams to describe use cases
Modeling behavior using activity diagrams

- Activity diagrams are used to model behavior as a structure of actions
  - Actions are performed by behavioral elements such as use cases, objects
- Use activity diagrams to model the behavior of:
  - Workflows or business processes
  - Use cases
  - Operations in classes

Activities

- Activities are networks of nodes connected by edges
- There are three categories of node:
  - Action nodes: represent discrete units of work that are atomic within the activity
  - Control nodes: control the flow through the activity
  - Object nodes: represent the flow of objects around the activity
- Edges represent flow through the activity
- There are two categories of edge:
  - Control flows: represent the flow of control through the activity
  - Object flows: represent the flow of objects through the activity
Key activity diagram symbols

- Action node
- Object node
- Decision node
- Merge node
- Initial node
- Join
- Fork
- Control node
- Flow final node
- Activity final node

Simple example

1. Receive Order
2. [order rejected]
3. Fill Order
4. [order accepted]
5. Ship Order
6. Accept Payment
7. Make Payment
8. Send Invoice
9. Invoice
10. Close Order

[Diagram showing the flow of the process described above]
**Activity diagram syntax**

- A complete activity starts in an **initial node** and terminates in a **final node**
- Activities can have **preconditions and postconditions**
- When an action node finishes, it produces a token that may traverse an edge to trigger the next action
  - This is sometimes known as a **transition**

**Modeling activities**

An activity is a structure of actions
Another example

Use case example
Advanced activity modeling
Activity diagram semantics

- The *token game*
  - Token – an object, some data or a focus of control
  - Imagine tokens flowing around the activity diagram
- Tokens traverse from a source node to a target node via an edge
  - The source node, edge and target node may all have constraints controlling the movement of tokens
  - All constraints must be satisfied before the token can make the traversal
- A node executes when:
  - It has tokens on all of its input edges AND these tokens satisfy predefined conditions (see later)
  - When a node starts to execute it takes tokens off its input edges
  - When a node has finished executing it offers tokens on its output edges
Action nodes

- Action nodes offer a token on all of their output edges when:
  - There is a token simultaneously on each input edge
  - The input tokens satisfy all preconditions specified by the node

- Action nodes:
  - Perform a logical AND on their input edges when they begin to execute
  - Perform an implicit fork on their output edges when they have finished executing

Types of action node

<table>
<thead>
<tr>
<th>action node syntax</th>
<th>action node semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Close Order</code></td>
<td>Call action - invokes an activity, a behavior or an operation. The most common type of action node. See next slide for details.</td>
</tr>
<tr>
<td><code>OrderEvent</code></td>
<td>Send signal action - sends a signal asynchronously. The sender does not wait for confirmation of signal receipt. It may accept input parameters to create the signal</td>
</tr>
<tr>
<td><code>OrderEvent</code></td>
<td>Accept event action - waits for events detected by its owning object and offers the event on its output edge. Is enabled when it gets a token on its input edge. If there is no input edge it starts when its containing activity starts and is always enabled.</td>
</tr>
<tr>
<td><code>end of month occurred</code></td>
<td>Accept time event action - waits for a set amount of time. Generates time events according to its time expression.</td>
</tr>
</tbody>
</table>

| input token | action node does not execute |
| action node | output token |
| action node | action node does not execute |
| action node | action node executes |
Call action node syntax

- The most common type of node
- Call action nodes may invoke:
  - an activity
  - a behavior
  - an operation
- They may contain code fragments in a specific programming language
  - The keyword 'self' refers to the context of the activity that owns the action

Control nodes

<table>
<thead>
<tr>
<th>control node syntax</th>
<th>control node semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial node – indicates where the flow starts when an activity is invoked</td>
<td></td>
</tr>
<tr>
<td>Activity final node – terminates an activity</td>
<td></td>
</tr>
<tr>
<td>Flow final node – terminates a specific flow within an activity. The other flows are unaffected</td>
<td></td>
</tr>
<tr>
<td>Decision node – guard conditions on the output edges select one of them for traversal May optionally have inputs defined by a «decisionInput»</td>
<td></td>
</tr>
<tr>
<td>Merge node – selects one of its input edges</td>
<td></td>
</tr>
<tr>
<td>Fork node – splits the flow into multiple concurrent flows</td>
<td></td>
</tr>
<tr>
<td>Join node – synchronizes multiple concurrent flows May optionally have a join specification to modify its semantics</td>
<td></td>
</tr>
</tbody>
</table>
Decision and merge nodes

- A decision node is a control node that has one input edge and two or more alternate output edges
  - Each edge out of the decision is protected by a guard condition
  - guard conditions must be mutually exclusive
  - The edge can be taken if and only if the guard condition evaluates to true
  - The keyword else specifies the path that is taken if none of the guard conditions are true
- A merge node accepts one of several alternate flows
  - It has two or more input edges and exactly one output edge

Fork and join nodes - concurrency

- Forks nodes model concurrent flows of work
  - Tokens on the single input edge are replicated at the multiple output edges
- Join nodes synchronize two or more concurrent flows
  - Joins have two or more incoming edges and exactly one outgoing edge
  - A token is offered on the outgoing edge when there are tokens on all the incoming edges i.e. when the concurrent flows of work have all finished
Activity Final Nodes vs. Flow Final Nodes

![Diagram showing activity final nodes vs. flow final nodes]

Activity partitions

- Each activity partition represents a high-level grouping of a set of related actions
  - Partitions can be hierarchical
  - Partitions can be vertical, horizontal or both
  - Partitions can refer to many different things e.g. business organisations, classes, components and so on
  - If partitions can’t be shown clearly using parallel lines, put their name in brackets

- (London::Marketing) Market product
- (p1, p2) SomeAction

nested partitions  multiple partitions
Partitions/Swimlanes

Partitions using annotations
Dimensional partitions

Expanding activities
Object nodes

- Object nodes indicate that instances of a particular classifier may be available
  - If no classifier is specified, then the object node can hold any type of instance
- Multiple tokens can reside in an object node at the same time
  - The upper bound defines the maximum number of tokens (infinity is the default)
- Tokens are presented to the single output edge according to an ordering:
  - FIFO – first in, first out (the default)
  - LIFI – last in, first out
  - Modeler defined – a selection criterion is specified for the object node

Object node syntax

- Object nodes have a flexible syntax. You may show:
  - upper bounds
  - ordering
  - sets of objects
  - selection criteria
  - object in state

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order</td>
<td>order objects may be available</td>
</tr>
<tr>
<td>Order (upperBound = 12)</td>
<td>zero to 12 Order objects may be available</td>
</tr>
<tr>
<td>Order (ordering = LIFO)</td>
<td>last Order object in is the first out (FiFO is the default)</td>
</tr>
<tr>
<td>Set of Order</td>
<td>sets of Order objects may be available</td>
</tr>
<tr>
<td>Order [open]</td>
<td>select Order objects in the open state</td>
</tr>
</tbody>
</table>

Order objects raised in December may be available
Activity parameters

Object nodes can provide input and output parameters to activities
- Input parameters have one or more output object flows into the activity
- Output parameters have one or more input object flows out of the activity
- Draw the object node overlapping the activity boundary

Pins

Pins are object nodes for inputs to, and outputs from, actions
- Same syntax as object nodes
- Input pins have exactly one input edge
- Output pins have exactly one output edge
- Exception pins are marked with an equilateral triangle
- Streaming pins are filled in black or marked with (stream)
**Input/Output pins**

A pin represents an input or output data node.

**Exceptions**

- Accept Payment
  - Send Rejection
  - Credit Account
- Accept Payment
  - Rejected Payment
  - Send Rejection
  - Accepted Payment
  - Credit Account
Timers

Interrupts
Summary

- Activity diagrams can be used to model behavior (e.g., use cases) as flows of activities

Key concepts

- Activities
- Connectors
- Activity partitions
- Action nodes
  - Call action node
  - Send signal/accept event action node
  - Accept time event action node
- Control nodes
  - decision and merge
  - fork and join
- Object nodes
  - input and output parameters
  - pins

The Object Constraint Language (OCL)

Specifying constraints in UML models
What is OCL?

- OCL can be used
  - to describe constraints
    - A constraint is a restriction on one or more values of a model or system.
    - A constraint is an expression that evaluates to true or false
  - as a query language
    - Queries are expressions that evaluate to a value (true, false, and other values)
    - Can be used to define new attributes and operations
- OCL expressions are always associated with a UML model
  - OCL expressions can be associated with any model element in UML

Constraints vs. Queries

- Examples of constraints:
  - Duration of a flight is the same as the difference between the arrival and departure times
  - The maximum number of passengers on a flight must be less than 1,001
  - The origin of a flight must be different than its destination
- Examples of queries:
  - Return all the departing flights from a given airport
  - Return all the flights departing from a given airport with a departure time after 4 p.m.
  - Derive the arrival time by adding the duration of the flight to the departure time.
Specifying Constraints - Invariants

Different kinds of constraints

- Class invariant
  - a constraint that must always be met by all instances of the class
- Precondition of an operation
  - a constraint that must always be true BEFORE the execution of the operation
- Postcondition of an operation
  - a constraint that must always be true AFTER the execution of the operation
**Constraint context and self**

- Every OCL expression is bound to a specific context.
  - The context is often the element that the constraint is attached to.

- The context may be denoted within the expression using the keyword ‘self’.
  - ‘self’ is implicit in all OCL expressions
  - Similar to ‘this’ in C++
Notation

- Constraints may be denoted within the UML model or in a separate document.
  - the expression:
    context Flight inv: self.duration < 4
  - is identical to:
    context Flight inv: duration < 4
  - is identical to:

```
<<invariant>>
duration < 4
```

Flight

duration: Integer

Elements of an OCL expression

- In an OCL expression these elements may be used:
  - basic types: String, Boolean, Integer, Real.
  - classifiers from the UML model and their features
    - attributes, and class attributes
    - query operations, and class query operations (i.e., those operations that do not have side effects)
  - associations from the UML model
Example: OCL basic types

context Airline inv:
name.toLower = ‘klm’

context Passenger inv:
age >= ((9.6 - 3.5)* 3.1).floor implies mature = true

Model classes and attributes

- “Normal” attributes
  context Flight inv:
  self.maxNrPassengers <= 1000

- Class attributes
  context Passenger inv:
age >= Passenger.minAge
Example: Using query operations

context Flight inv:

self.departTime.difference(self.arrivalTime) .equals(self.duration)

<table>
<thead>
<tr>
<th>Time</th>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>$midnight: Time</td>
<td>nrOfDays : Integer</td>
</tr>
<tr>
<td>month : String</td>
<td>nrOfHours : Integer</td>
</tr>
<tr>
<td>day : Integer</td>
<td>nrOfMinutes : Integer</td>
</tr>
<tr>
<td>year : Integer</td>
<td>equals(i:Interval):Boolean</td>
</tr>
<tr>
<td>hour : Integer</td>
<td>$Interval(d, h, m : Integer) :</td>
</tr>
<tr>
<td>minute : Integer</td>
<td>Interval</td>
</tr>
<tr>
<td>difference(t:Time):Interval</td>
<td></td>
</tr>
<tr>
<td>before(t: Time): Boolean</td>
<td></td>
</tr>
<tr>
<td>plus(d : Interval) : Time</td>
<td></td>
</tr>
</tbody>
</table>

Associations and navigations

- Every association in the model is a navigation path.
- The context of the expression is the starting point.
- Role names are used to identify the navigated association.
Example: navigations

context Flight
inv: origin <> destination
inv: origin.name = ‘Amsterdam’

context Flight
inv: airline.name = ‘KLM’

Association classes

context Person inv:
if employer.name = ‘Klasse Objecten’ then
  job.type = JobType::trainer
else
  job.type = JobType::programmer
endif
Significance of Collections in OCL

- Most navigations return collections rather than single elements

<table>
<thead>
<tr>
<th>Flight</th>
<th>0..*</th>
<th>Airplane</th>
</tr>
</thead>
<tbody>
<tr>
<td>type: Airtype</td>
<td>1</td>
<td>type: Airtype</td>
</tr>
<tr>
<td>flights</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Three Subtypes of Collection

- Set:
  - arrivingFlights (from the context Airport)
  - Non-ordered, unique
- Bag:
  - arrivingFlights.duration (from the context Airport)
  - Non-ordered, non-unique
- Sequence:
  - passengers (from the context Flight)
  - Ordered, non-unique
Collection operations

- OCL has a great number of predefined operations on the collection types.
- Syntax:
  - collection->operation

The collect operation

- The collect operation results in the collection of the values obtained by evaluating an expression for all elements in the collection.
The collect operation

context Airport inv:
self.arrivingFlights -> collect(airLine) -> notEmpty

The collect operation syntax

- Syntax:
  collection->collect(elem : T | expr)
collection->collect(elem | expr)
collection->collect(expr)

- Shorthand:
collection.expr

- Shorthand often trips people up. Be Careful!
The select operation

The *select* operation results in the subset of all elements for which a boolean expression is true.

context Airport inv:
self.departingFlights->select(duration<4)->notEmpty

- **f1** duration = 2
- **f2** duration = 5 (red)
- **f3** duration = 3
- **f4** duration = 5
- **f5** duration = 2

The select operation syntax

- **Syntax:**
  collection->select(elem : T | expression)
  collection->select(elem | expression)
  collection->select(expression)
The forAll operation

- The forAll operation results in true if a given expression is true for all elements of the collection

Example: forAll operation

context Airport inv:
self.departingFlights->forAll(departTime.hour>6)
The forAll operation syntax

- Syntax:
  - collection->forAll(elem : T | expr)
  - collection->forAll(elem | expr)
  - collection->forAll(expr)

The exists operation

- The exists operation results in true if there is at least one element in the collection for which a given expression is true.
Example: exists operation

context Airport inv:
self.departingFlights->exists(departTime.hour<6)

The exists operation syntax

- Syntax:
  collection->exists(elem : T | expr)
  collection->exists(elem | expr)
  collection->exists(expr)
Other collection operations

- **isNullOrEmpty**: true if collection has no elements
- **notEmpty**: true if collection has at least one element
- **size**: number of elements in collection
- **count(elem)**: number of occurrences of elem in collection
- **includes(elem)**: true if elem is in collection
- **excludes(elem)**: true if elem is not in collection
- **includesAll(coll)**: true if all elements of coll are in collection

Local variables

- The *let* construct defines variables local to one constraint:
  Let var : Type = <expression1> in <expression2>

- Example:
  context Airport inv:
  Let supportedAirlines : Set (Airline) =
  self.arrivingFlights -> collect(airLine) in
  (supportedAirlines ->notEmpty) and
  (supportedAirlines ->size < 500)
Iterate

- The *iterate* operation for collections is the most generic and complex building block.

```
collection->iterate(elem : Type;
 answer : Type = <value> |<expression-with-elem-and-answer>)
```

Iterate example

- Example iterate:
  
  ```
  context Airline inv:
  flights->select(maxNrPassengers > 150)->notEmpty
  ```

  Is identical to:
  
  ```
  context Airline inv:
  flights->iterate (f : Flight;
  answer : Set(Flight) = Set{ } | if f.maxNrPassengers > 150 then
  answer->including(f)
  else
  answer endif )->notEmpty
  ```
Specifying Constraints: Operation Specifications

Pre- and PostCondition Example

A class named Account has an attribute balance and an operation overdraft() that returns true if the balance is less than 0 and false otherwise.

context Account::overdraft():Boolean
pre : -- none
post : result = (balance < 0)
More complex operation specifications

The operation birthdayOccurs() adds 1 to the customer age.

**context** \( \text{Customer::birthdayOccurs()} \)

**pre** : -- none

**post** : \( \text{age} = \text{age}@\text{pre} + 1 \)

**context** \( \text{Account::safeWithdraw(amt:Integer)} \)

**pre** : balance > amt

**post** : balance = \( \text{balance}@\text{pre} - \text{amt} \)

---

Example model

<table>
<thead>
<tr>
<th>Airport</th>
<th>Flight</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: String</td>
<td>departTime: Time</td>
</tr>
<tr>
<td></td>
<td>/arrivalTime: Time</td>
</tr>
<tr>
<td></td>
<td>maxNrPassengers: Integer</td>
</tr>
<tr>
<td></td>
<td>passengers</td>
</tr>
<tr>
<td></td>
<td>* (ordered)</td>
</tr>
<tr>
<td></td>
<td>* (ordered)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{midnight}: \text{Time}$</td>
</tr>
<tr>
<td>month : String</td>
</tr>
<tr>
<td>day : Integer</td>
</tr>
<tr>
<td>hour : Integer</td>
</tr>
<tr>
<td>minute : Integer</td>
</tr>
<tr>
<td>difference:($\text{Time}$/Interval)</td>
</tr>
<tr>
<td>before:($\text{Time}$/Time)</td>
</tr>
<tr>
<td>equals:($\text{Interval}$/Interval)</td>
</tr>
<tr>
<td>$Interval(d,h,m) : Interval \</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>customers</td>
</tr>
<tr>
<td>(0..1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>nrOfDays : Integer</td>
</tr>
<tr>
<td>nrOfHours : Integer</td>
</tr>
<tr>
<td>nrOfMinutes : Integer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CEO</th>
</tr>
</thead>
</table>

© Robert B. France
Derived Attribute & Initial Value Example

Defining derived attributes
context Flight::arrivalTime:Time
derive:departTime.plus(duration)

Defining initial attribute value
context Flight::maxNrPassengers:Integer
init: 100

Defining initial association end value
context Flight::passengers:Set(Passenger)
init: Set()

Query operation examples

Return all the departing flights from a given airport
context Airport::departures():Set(Flight)
body: result=departingFlights

Query operation example: Return all the airports served by an airline
context Airline::served():Set(Airport)
body: result=flights.destination->asSet
Inheritance of constraints

- Guiding principle Liskov’s Substitution Principle (LSP):
  - “Whenever an instance of a class is expected, one can always substitute an instance of any of its subclasses.”

Inheritance of constraints

- Consequences of LSP for invariants:
  - An invariant is always inherited by each subclass.
  - Subclasses may strengthen the invariant.
- Consequences of LSP for preconditions and postconditions:
  - A precondition may be weakened (contravariance)
  - A postcondition may be strengthened (covariance)
An Example: Royal and Loyal Model

Taken from “The Object Constraint Language” by Warmer and Kleppe
Defining initial values & derived attributes

**context** LoyaltyAccount::points  
**init**: 0

**context** CustomerCard::valid  
**init**: true

**context** CustomerCard::printedName  
**Derive**: owner.title.concat(‘ ’).concat(owner.name)

**context** LoyaltyProgram  
**inv**: partners.deliveredServices -> size() > 0

**context** LoyaltyProgram  
**inv**: partners.deliveredServices ->  
forall(pointsEarned = 0 and pointsBurned  
implies Membership.account -> isEmpty())

A note on the collect operation  
partners -> collect(numberOfCustomers)  
can also be written as  
partners.numberOfCustomers
context Customer
inv: programs -> size() = cards -> select (valid = true) -> size()

context ProgramPartner
inv: numberOfCustomers = programs.participants -> asSet() -> size()
### Defining new attributes and operations

```java
def: turnover : Real = transactions.amount -> sum()
//Attributes introduced in this manner are always derived attributes
```

### Specifying Operations

```java
def: getServicesByLevel(levelName:String): Set(Service)
= levels -> select (name = levelName).availableServices ->asSet()
```
Inheritance of constraints

- Guiding principle Liskov’s Substitution Principle (LSP):
  - “Whenever an instance of a class is expected, one can always substitute an instance of any of its subclasses.”

- Consequences of LSP for invariants:
  - An invariant is always inherited by each subclass.
  - Subclasses may strengthen the invariant.

- Consequences of LSP for preconditions and postconditions:
  - A precondition may be weakened (contravariance)
  - A postcondition may be strengthened (covariance)
**OCL Summary**

- OCL invariants allow you to
  - model more precisely
  - remain implementation independent
- OCL pre- and post-conditions allow you to
  - specify contracts (design by contract)
  - specify interfaces of components more precisely
- OCL usage tips
  - keep constraints simple
  - always give natural language comments for OCL expressions
  - use a tool to check your OCL

---

**Design Modeling**

Modeling Software Solutions using UML
Design Overview

- Two major phases
  - Architectural design
    - Concerned with identifying subsystems and their relationships.
  - Detailed design
    - Concerned with refining architecture
      - define algorithms, threads and data structures

- Static and Behavioral design views
  - Static view: Concerned with identifying solution concepts and their relationships
  - Behavioral view: Concerned with defining interactions between objects and behavior of individual objects

Moving from Problem to Solution

Requirements

- System attributes (non-functional requirements, anticipated changes, design constraints)
- Use Cases
- Requirements class diagram

Initial Design

- Design goals
- Architecture
  - subsystem structure
  - subsystem interfaces
Architectural design

- Group objects into subsystems and describe relationships between subsystems.
  - Static view: Describe relationships between subsystems
    - Component-based style
    - Import/Export style
  - Behavioral view: Describe interactions between subsystems

Identifying Subsystems

- Use Case based identification
  - Assign objects associated with a use case into the same subsystem
  - Objects that are used in more than one use case can be put into a separate subsystem
  - Reduce subsystem coupling and improve subsystem cohesion

- Function-based identification
  - Group use cases along functional lines and put objects involved in a group into the same subsystem
Subsystem Decomposition

- Subsystems require and provide services
  - In a component-based architecture these services are declared in interfaces
  - In import/export architectures provided services are exported from while required services are imported into a subsystem
- Objects in a subsystem collaborate to carry out the provided services

Detailed design

- Detailed Design: Define how services provided by subsystems are realized
  - Develop design class diagrams
  - Develop behavioral models
  - Define methods
Component-based architecture (CBA)

CBA views

- Structural view
  - UML package represents a subsystem
  - Interfaces are classifiers
- Behavioral view
  - Sequence diagrams used to describe interactions between subsystems
  - Sequence diagrams show interfaces and calls to operations in interfaces
Subsystems in CBA

- Subsystems have interfaces that declare provided and required services
  - Provided service: a service offered by the subsystem
  - Required service: a service needed by the subsystem that must be provided by another subsystem
CBA structural view: An example

Use cases grouped into borrower services and administrative services; two subsystems for each of these groupings is created — one responsible for borrower services, the other for administrative services.

Both administrative and borrower services need copy and borrower information. This observation results in a decision to include two subsystems for managing copies and borrowers.

ReturnCopy Sequence Model
A Simple Library Example

Simple Library System requirements

- Library consists of reference and general copies
  - Only general copies can be checked out; reference copies cannot be checked out
- A borrower cannot check out a copy if they have outstanding fines.
- A borrower is charged $1 for each week after its due date that it is not returned.
Library example: required services

CheckOutCopy
ReturnCopy
AddCopy
DeleteCopy
AddBorrower
DeleteBorrower
FindCopy

CheckOutCopy UseCase (draft form)

UseCase: CheckOutCopy
Main flow
1. Clerk enters borrower and copy identifiers
2. If borrower is registered, does not have fines, and copy is a general copy then system indicates that checkout is successful; else system indicates that checkout is not successful
Requirements Class Model

Transitioning to Architectural Design

Describe how subsystems interact to accomplish use cases

Requirements use case ➔ Architectural sequence models

Identify subsystems by grouping use cases

Identify subsystems by grouping classes

Architectural static model

Requirements class model
Architectural Static Model

Use cases grouped into borrower services and administrative services; two subsystems for each of these groupings is created – one responsible for borrower services, the other for administrative services.

Both administrative and borrower services need copy and borrower information. This observation results in a decision to include two subsystems for managing copies and borrowers.

CheckOut Sequence Model

- A sequence model for the check-out process is depicted.
- The model includes interactions between different services and objects.
- Key activities include checking out a copy, updating borrower information, and record-keeping.
- The flowchart highlights conditions such as "copy not registered," "borrower has fines," and "borrower registered."
ReturnCopy Sequence Model

Import/Export Architecture
Accessing Package Elements

- To control access to elements in packages they are accessed or imported.
- **Import/Access**: A package that imports/accesses another has access to the public elements of the imported/accessed package
  - public elements are said to be *exported*

---

Import versus Access

- **Package1** contains public **Class1**; **Package2** contains public **Class2**.
- **Package1** access **Package2**
  - elements in Package1 must reference Class2 as follows: Package2::Class2
- **Package1** import **Package2**
  - elements in Package1 can reference Class2 as just Class2
Import vs. Access

An Import/Export example
Detailed Design Modeling

Subsystem design basics

- Focus on modeling how subsystems accomplish goals (deliver services)
- Models
  - Design Class Model
  - Behavioral Models
    - Sequence diagrams – describes object interactions
    - Activity diagrams – describes behavior in an operation
    - State model – describes how the state of an object affects its behavior
- In the following we will first present design class and sequence modeling notations and then give an overview of how to develop a good OO design
UML Design Class Models

Requirements vs. Design classes

A Payment in the Domain Model is a concept, but a Payment in the Design Model is a software class. They are not the same thing, but the former inspired the naming and definition of the latter. This reduces the representational gap. This is one of the big ideas in object technology.

UP Domain Model
Stakeholder’s view of the noteworthy concepts in the domain.

UP Design Model
The object-oriented developer has taken inspiration from the real world domain in creating software classes. Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software, has been lowered.
Key class modeling activities

- Identify key solution concepts that will be represented by program classes
- Assign responsibilities to classes
  - Determines class attributes and operations
  - Add new classes if none of the existing classes are not suitable for particular responsibilities
- Determine which objects need to know of other objects
  - Determines navigability

Identifying operations

- Operations are needed to realize the responsibilities assigned to each class
  - There may be several operations per responsibility
  - The main operations that implement a responsibility are normally declared `public`
  - Other methods that collaborate to perform the responsibility must be as private as possible
- Assigning responsibilities to classes will be discussed in more detail after the design models section of this tutorial
Style Guidelines for design classes

- Center class name in boldface.
- Capitalize the first letter of class names (if the character set supports uppercase).
- Left justify attributes and operations in plain face.
- Begin attribute and operation names with a lowercase letter.
- Put the class name in italics if the class is *abstract*.
  - An abstract class is one whose instances must be instances of a specialized class
  - At the implementation level, this translates to a class that cannot be instantiated
- Show full attributes and operations when needed and suppress them in other contexts or when merely referring to a class.

### Depicting Classes

Information about attributes and operations can be suppressed in a class model

```
Rectangle
  getArea
  resize

Rectangle
  height
  width

getArea(): int
resize(int,int)
```
General Guidelines

- Avoid dumb objects: objects that hold data and provide only get/set methods
- Avoid “god” controllers: a “god’ controller is one that requests state information (e.g., using a get method) and uses the information to make decisions or perform calculations
- Avoid coupling by having services above and beyond get/set services in interface of objects
- A client should request an object to do something on its behalf, not request information about an object’s state.

Navigability

- In a design model one can indicate that an object “knows about” another object it is linked to by using navigation arrows on associations
  - In UML 2.0 one can also explicitly show that one object does not know about the objects it is linked to.
The constructs in diagrams 1, 2, and 4 are new to UML 2.0 and thus are most likely not supported by UML tools as yet.

- The top pair AB shows a binary association with two navigable ends.
- The second pair CD shows a binary association with two non-navigable ends.
- The third pair EF shows a binary association with unspecified navigability.
- The fourth pair GH shows a binary association with one end navigable and the other non-navigable.
- The fifth pair IJ shows a binary association with one end navigable and the other having unspecified navigability.

Design class diagram example
UML Behavioral Models

Specifying behavior using the UML

- Class models describe objects and their relationships
  - Behavior can be specified in terms of operation pre and postconditions, but behavior is not the primary focus of a class model

- Behavioral models in the UML
  - State models: describe control aspects of a system – provides descriptions sequences of operations without regard for what the operation do.
  - Interaction models: describe interactions among objects
  - Activity models: description of a behavioral feature expressed in terms of sequences of steps.
Sequence Models

Overview

- Realizations of use cases can be expressed as interaction diagrams
  - Objects interact to accomplish use case goals.
  - Helps to identify Manager and Helper classes
- Object interactions are described in terms of
  - Collaborations: descriptions of object structures that support required behaviors
  - Interactions: descriptions of communication structures that support required behaviors
- Interaction diagrams allow one to view only the parts of a system involved in accomplishing use case goals
Sequence Diagram: Basic constructs

Different kinds of arrows

**Synchronous message**

**Asynchronous message**

**Return**
Sequence diagram control structures

Control structures are shown as combined fragments

- Alternatives (alt)
  - choice of behaviors – at most one will execute
  - depends on the value of the guard (“else” guard supported)

- Option (opt)
  - Special case of alternative

- Loop (loop)
  - Optional guard: [<min>, <max>, <Boolean-expression>]
  - No guard means no specified limit

Loop example

[Diagram of a sequence diagram with a loop example]

- loop fragment
- condition
More combined fragment types

- **Break (break)**
  - Represents an alternative that is executed instead of the remainder of the fragment (like a break in a loop)

- **Parallel (par)**
  - Concurrent (interleaved) sub-scenarios

- **Negative (neg)**
  - Identifies sequences that must not occur

- **Critical Region (region)**
  - Traces cannot be interleaved with events on any of the participating lifelines

- **Assertion (assert)**
A more complex Sequence Diagram (UML 2.0)

Referencing interaction diagrams
A state model specifies the life histories of objects in terms of the sequences of operations that can occur in response to external stimuli.

- For example, a state model can describe how an object responds to a request to invoke one of its methods.

- A state model consists of state diagrams that each describes how an object responds to external stimuli.

- A state diagram describes behavior in terms of sequences of states that an object can go through in response to events.
Simple Example: Telephone Object

- **Idle**
- **Active**

**Initial State**: When the object is created, it moves into the Idle state.

**Event**: If a “lift receiver” event is received, the object moves from the Idle to the Active state. During the transition, the “get dial tone” activity is executed.

**Event**: If a “caller hangs up” event occurs when in the Active state, the object moves to the Idle state and the “disconnect” activity is performed during the transition.

Key Concepts

- **An event** is a significant or noteworthy occurrence at a point in time.
  - Examples of events: sending a request to invoke a method, termination of an activity.
  - An event occurs instantaneously in the time scale of an application.

- **A state** is a condition of an object during its lifetime.
  - For example, a student is in the registered state after completing course registration.
  - A state is an abstraction of an object’s attribute values and links
    - For example, a bank account is in the Overdraft state when the value of its balance attribute is less than 0.
Key Concepts - 2

- A transition occurs when an event causes an object to change from its current (source) state to a target state.
  - For example, if a student is in the registered state and then drops out of the program then the student is in the “not registered” state.
  - The source and target states can be the same.
  - A transition is said to fire when the change from source to target state occurs.
- A guard condition on a transition is a boolean expression that must be true for a transition to fire.
- An activity is a behavior that is executed in response to an event.

Basic UML State Diagram
Guards

- Conditional execution of transitions
  - guards (Boolean predicates) must be side-effect free

\[
\text{bid } [\text{value} < 100] / \text{reject}
\]

\[
\text{bid } [\text{value} \geq 200] / \text{sell}
\]

\[
\text{bid } [(\text{value} \geq 100) \& (\text{value} < 200)] / \text{sell}
\]

\[
\text{Selling} \quad \text{Unhappy} \quad \text{Happy}
\]

Event Types

- \textit{ChangeEvent}: Occurs when a condition expressed as a boolean expression becomes true.
  - \textit{when condition}
  - Examples: \textit{when(temperature > 80)}; \textit{when(balance < 0)}

- \textit{SignalEvent}: Occurs when an signal is sent or received, where a signal is a one-way transmission of information from one object to a target object.
  - \textit{Event}(param1,param2,…)
  - Examples: lift receiver; hang-up telephone

- \textit{TimeEvent}: Passage of a specific period of time after a designated event, or an occurrence of a specified time instance.
  - Passage of time since entry to current state - \textit{after (time period)}, e.g., after (10 secs), after (5 secs since exit of state A)
  - Occurrence of a time instance – \textit{when(due-date=Feb 28, 2005, 11pm)}
Object Behavior - General Model

- Typical object lifecycle

  Handling depends on specific request type

  ```
  void offHook()
  {
    busy = true;
    obj.reqDialtone();
    ...
  }
  ```

Object Behavior and State Machines

- Reflecting object lifecycle in a state diagram

```
Object and Threads

- **Passive objects**: Objects that respond only to method invocation events
  - Objects execute methods only when requested (thread of execution - left diagram)
- **Active objects**: objects perform actions on their own
  - Object has its own thread of execution (right diagram)

---

Passive Objects: Dynamic Semantics

- Encapsulation of information within a passive object does not protect the object from concurrency conflicts
- Explicit synchronization is still required
  - Synchronization can be described by state diagrams (e.g., protocol state models can be used to restrict when a passive object responds to method calls)
Active Objects and State Diagrams

If object is in the created state and a start event occurs, it sends a method invocation event (request to invoke ready()) to the object named master and moves to state ready.

Active Objects: Dynamic Semantics

**Run-to-completion model:**
- Events are queued
- An object processes events in a queue one at a time.
- An object cannot suspend handling of an event to handle another event
The Run-to-Completion Model

- A high priority event for (another) active object will preempt an active object that is handling a low-priority event on a uniprocessor.

State Representations

- StateName
  - activity executed when state is entered
  - activity executed when state is exited
  - activity executed while event occurs
  - activity executed while in state
  - entry/ action
  - exit/ action
  - event/ action
  - do/ action
State Syntax

A state symbol can have one or more compartments (all optional):
- Name compartment
- Internal transition compartment
  - contains internal actions or activities
  - Format: `event(args) [condition] / action`
  - `entry / action` (invoked implicitly)
  - `exit / action` (invoked implicitly)
  - `do / machine name` (invokes a nested state machine)

Entry and exit activities cannot have arguments and cannot have guard conditions.
Activity expressions can use object attributes and links, and parameters of events on incoming transitions.
State Entry and Exit Actions

LampOn

entry/lamp.on();
exit/lamp.off();

Resulting activity sequence:
printf("exiting");
printf("to off");
lamp.off();

Order of Actions: Simple Case

LampOn
entry/lamp.on();
exit/printf("exiting");

LampOff
entry/lamp.off();
exit/printf("exiting");

off/printf("to off");

off/printf("needless");

printf("exiting");
printf("needless");
lamp.off();
Internal Transitions

- Self-transitions that bypass entry and exit actions

**LampOff**
- entry/lamp.off();
- exit/printf("exiting");
- off/null;

Internal transition triggered by an "off" event

State ("Do") Activities

- Forks a concurrent thread that executes until:
  - the action completes or
  - the state is exited through an outgoing transition

**Error**
- entry/printf("error!")
- do/while (true) alarm.ring();

"do" activity
Static Conditional Branching
- Merely a graphical shortcut for convenient rendering of decision trees

Dynamic Conditional Branching
- Choice pseudostate: guards are evaluated at the instant when the decision point is reached
Hierarchical State Machines

- Graduated attack on complexity
  - states decomposed into state machines

```
<table>
<thead>
<tr>
<th>LampOff</th>
<th>entry/lamp.off()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>off/</td>
</tr>
<tr>
<td></td>
<td>on/</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LampOn</th>
<th>entry/lamp.on()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>on/</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LampFlashing</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlashOn</td>
</tr>
<tr>
<td>entry/lamp.on()</td>
</tr>
<tr>
<td>on/</td>
</tr>
<tr>
<td>1sec/</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FlashOff</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry/lamp.off()</td>
</tr>
<tr>
<td>1sec/</td>
</tr>
</tbody>
</table>
```

“Stub” Notation

- Notational shortcut: no semantic significance

```
<table>
<thead>
<tr>
<th>LampOff</th>
<th>entry/lamp.off()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>off/</td>
</tr>
<tr>
<td></td>
<td>on/</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LampOn</th>
<th>entry/lamp.on()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>on/</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>LampFlashing</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlashOn</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FlashOff</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
```
Group Transitions

- Higher-level transitions

**LampOff**
entry/lamp.off()  
off/  
on/  

**LampOn**
entry/lamp.on()  

**LampFlashing**
FlashOn
entry/lamp.on()  
1sec/  
FlashOff
entry/lamp.off()  
1sec/  

**Default transition to the initial state in LampFlashing**

Group transition: when on occurs in any LampFlashing state a transition is made to LampOn

Completion Transitions

- Triggered by a completion event
  - generated automatically when an immediately nested state machine terminates

**Committing**

<table>
<thead>
<tr>
<th>Phase1</th>
<th>Phase2</th>
</tr>
</thead>
</table>

**CommitDone**

completion transition (no trigger)
### Triggering Rules

- Two or more transitions may have the same event trigger
  - innermost transition takes precedence
  - event is discarded whether or not it triggers a transition

![Lamp Flashing Diagram](image)

### State Syntax - Submachines

- **State name: submachine**
  - machine is the name of a (nested) state machine that has an initial and final state.
  - Execution of the nested machine begins in the initial state.
  - When final state of the nested machine is reached then the exit action of the parent state is executed (if it exists).
UML 2.0: Entry/Exit Points

- Encapsulation of submachines

Entry/Exit Points: Usage
Deferred Events

- Events can be retained if they do not trigger a transition
  - The event is put to the top of the queue; it must be handled in the next event handling cycle

Order of Actions: Complex Case

- Same approach as for the simple case

Actions execution sequence:

exS11 ⇔ exS1 ⇔ actE ⇔ enS2 ⇔ initS2 ⇔ enS21
Composite States

- A state can be decomposed in two ways:
  - AND decomposition: object state decomposed into two parallel components.
  - OR decomposition: object state decomposed into a network of states.

Orthogonality

- Multiple simultaneous perspectives on the same entity
Orthogonal Regions

- Combine multiple simultaneous descriptions

Orthogonal Regions - Semantics

- All mutually orthogonal regions detect the same events and respond to them “simultaneously”
  - usually reduces to interleaving of some kind
Interactions Between Regions

- Typically through shared variables or awareness of other regions’ state changes

- **sane**: Boolean
- **flying**: Boolean

**Catch22**

**Sanity Status**

- **Crazy**
  - entry/sane := false;
- **Sane**
  - entry/sane := true;
- request
- **Grounding/**
- **Flying**
  - entry/flying := true;
- **Grounded**
  - entry/flying := false;

**Flight Status**

Transition Forks and Joins

- For transitions into/out of orthogonal regions:

**Staff**

- **Child**
- **Adult**
- **Retiree**

**Employee**

- **Manager**
- **Staff Member**
Common Misuse of Orthogonality

- Using regions to model independent objects

An Example
Developing good OO designs

Key design activities

- Assign responsibilities to classes
  - Determines the operations that a class will have
- Determine which objects need to know of other objects (determine class navigability)
  - Determines the references that a class will have
  - A reference contains a pointer to an object
Class responsibilities

- A *responsibility* is something that the class is required to do (contract or obligation of a class)
  - Each functional requirement must be attributed to one of the classes
    - All the responsibilities of a given class should be *clearly related*.
    - If a class has too many responsibilities, consider *splitting* it into distinct classes
    - If a class has no responsibilities attached to it, then it is probably useless
    - When a responsibility cannot be attributed to any of the existing classes, then a *new class* should be created

Distributing responsibilities across classes

- Perform use case analysis
  - For each use case determine the class that is responsible for coordinating the behavior that accomplishes the use case goals. This class are referred to as the **Manager** class for the use case
    - The Manager class is a coordinator; try to write it so that it does mostly coordination tasks (e.g., calling methods in other classes, or transforming results it receives into a presentable form).
    - More importantly, the Manager class should not just call getters and setters and do all the work using results received from getters.
  - After the Manager class is identified, identify the classes that have information that will be used to accomplish the use case goals and assign to these classes responsibilities that are specific to the information those classes have access to. These classes are referred to as **Helper** classes for the use case.
    - The responsibilities should not just be simple get and set! Have the classes manipulate the information that they have access to to produce a result of value to the use case.
Responsibility types

- Action-oriented responsibilities
  - Carrying out an activity
  - Controlling and coordinating activities in other objects
  - Delegating activities to other objects
- Data-oriented responsibilities
  - Maintaining private information
  - Maintaining relationships to other objects
  - Maintaining derived information
- Interaction models reflect decisions pertaining to assignment of responsibilities
  - Message passing between objects reflect responsibility distribution
  - Use interaction diagrams to explore possible distribution of responsibilities

Patterns of Responsibility Assignment Principles

- Cohesion
- Expert
- Creator
- Low Coupling
- Controller
High Cohesion

- Cohesion is a measure of how diverse an entity’s features are.
  - A highly cohesive class has features that pertain to a single concept
  - A highly cohesive class has one general responsibility
    - Guideline: Should be able to describe responsibility of a highly cohesive class in one sentence
    - Use sentence as comment in code
  - Guideline: Assign a responsibility so that parts of the class are strongly related and the class responsibility is tightly focused
    - Class easier to understand
    - Easier to maintain and reuse

Levels of Cohesion

- Very low cohesion: class is responsible for many things in different functional contexts
- Low cohesion: class is solely responsible for a complex group of tasks in a single functional area
- Moderate cohesion: class is responsible for relatively simple tasks in different but related functional areas
- High cohesion: class is responsible for a group of tasks in a single functional area and discharges its responsibility by delegating some of its responsibilities to other classes.
  - Example: Machine-Machine Schedule association
When to ignore high cohesion guidelines

- A class that provides a single point of entry into a system may sometimes be desirable
  - Such a class is called a Façade and provides external clients with a single point of access to services offered by a system
- For efficiency reasons it may be more appropriate to place two diverse classes in the same class
  - Rather than an object delegating responsibility for a service to another object it may carry it out itself to avoid delegation performance overhead

Expert

- Assign responsibility to the class that has the information necessary to discharge the responsibility.
- Naïve use can lead to undesirable coupling and low cohesion.
  - Giving a class the responsibility for storing its objects in a database leads to low cohesion and undesirable coupling
    - Low cohesion: class contains code related to database handling between the
    - Undesirable coupling: class is tightly coupled to database services provided by another system
  - Example: Elevator Control System
Creator

- Class B can be responsible for creating objects of A in the following situations:
  - A is a part class of B
  - B is a container of A objects
  - B records A objects
  - B has the data needed to initialize A objects

Low Coupling

- Assign responsibilities to reduce high coupling to unstable classes (i.e., classes with high probability of significant changes)
  - Reduces impact of change
  - Classes can be understood in relative isolation

- Forms of coupling in OO designs
  - Class X contains a reference to Class Y objects
  - Class X operation includes calls to Class Y operations
  - Class X operation has a Class Y object as a parameter or declares a Class Y object as a local variable
  - Class X is a direct or indirect subclass of Class Y
  - Class X implements an interface Y

- Classes designed for reuse should have low coupling. Why?
Controller

- Assign responsibility for handling a system event to a class representing the system or a class that is responsible for handling the events in a group of related use cases.
  - A system event is an event generated by an actor. A system event results in the execution of a system operation.
  - A controller is a non-user interface class responsible for receiving and handling system events. A controller defines the method for the system operation.
- A good controller delegates the work needed to handle a system event to other objects.
  - A controller controls and coordinates the collaborating objects.
  - A controller does not do much of the actual work.

Controller Options

- Presentation objects (UI objects) should not be responsible for handling events
  - Decouple presentation layer from application processing layer. Why?
- System as controller
  - Referred to as a façade controller
  - Use when number of system events is not large
    - Large number of events can lead to a controller with low cohesion and high coupling
- Use case handlers
  - For each use case design a controller that handles the use case events
  - Use when number of system events is large
Bloated Controllers

- Signs of problematic design
  - Interface objects handle system events directly
  - Controller object handles many events
  - Controller object performs bulk of work needed to handle event.
  - Controller class has many attributes because of its many responsibilities.

A small example illustrating assignment of responsibility to classes

This example illustrates the iterative nature of design modeling – you seldom get a good design on a first attempt!
The Problem

- [Variation of a program example used in the book “Refactoring” by Martin Fowler]
- Program calculates and prints a statement of a customer’s charges at a video store. There are three types of movies: Regular, NewRelease, Children.
- Rates
  - Regular: 2.00 for 2 days; late 1.50/late day
  - Children: 1.50 for 3 days; late 1.50/late day
  - NewRelease: 3.00/day

Naïve Design: Class Diagram

```
Customer
- name: String
+ addRental()
+ getName()
+ Statement()

Rental
- daysRented:int
+ getDaysRented()
+ getMovie()

Movie
- title: String
- priceCode:int
+ getPriceCode()
+ setPriceCode()
+ getTitle()
```

```
1
- rentals

1
- movie
```

```
```
Design 1: Sequence Model

Naïve Design: Code for Movie

```java
public class Movie{
    public static final int CHILDREN = 2;
    public static final int REGULAR = 0;
    public static final int NEW = 1;

    private string title;
    private int priceCode;

    public Movie (String mtitle, int mpriceCode)
    {
        title = mtitle; priceCode=mpriceCode;
    }

    public int getPriceCode(){
        return priceCode;
    }

    public String getTitle(){
        return title;
    }

    public void setPriceCode (int arg){
        priceCode=arg;
    }
}
```
Code for Rental

class Rental{
    private Movie movie;
    private int daysRented;

    public Rental (Movie rmovie, int rdaysRented){
        movie = rmovie;
        daysRented = rdaysRented;
    }

    public int getDaysRented(){
        return daysRented;
    }

    public Movie getMovie(){
        return movie;
    }
}

Code for Customer

class Customer{
    private string name;
    private List<Rental> rentals = new ArrayList<Rental>();

    public Customer (String cname){
        name = cname;
    }

    public void addRental (Rental arg){
        rentals.addElement(arg);
    }

    public String getName(){
        return name;
    }

    public String statement(){...
}
Code Skeleton from Sequence Model

```java
public void statement(){
    ...
    Iterator<Rental> iter = rentals.iterator();
    ...
    while(iter.hasNext()){
        Rental aRental=iter.next();
        Movie aMovie=aRental.getMovie();
        int pc=aMovie.getPriceCode();
        int days=aRental.getDaysRented();
        ...
        String t=aMovie.getTitle();
        ...
    }
    ...
}
```
public void statement(){
    double totalAmt = 0;
    Iterator<Rental> iter = rentals.iterator();
    System.out.printf("Record for %s \n", getName());
    while(iter.hasNext()){
        Rental aRental=iter.next();
        Movie aMovie=aRental.getMovie();
        int pc=aMovie.getPriceCode();
        int days=aRental.getDaysRented();
        double thisAmt = 0;
        switch (pc){
            case Movie.REGULAR:
                thisAmt = 2;
                if (days > 2)
                    thisAmt += (days-2)*1.5;
                break;
            case Movie.NEW:
                thisAmt = days*3;
                break;
            case Movie.CHILDREN:
                thisAmt = 1.5;
                if (days > 3)
                    thisAmt += (days-3)*1.5;
                break;
        }
        String t=aMovie.getTitle();
        System.out.printf("%s", t);
        System.out.printf("%.2f\n", thisAmt);
        totalAmt=totalAmt+thisAmt;
    }
    System.out.printf("Amount owed is %.2f\n",totalAmt);
}

Modified Sequence Model
Code for getAmount()

Private double getAmount (Rental aRental){
    Movie aMovie=aRental.getMovie();
    int pc=aMovie.getPriceCode();
    int days=aRental.getDaysRented();
    double thisAmt = 0;
    switch (pc){
        case Movie.REGULAR:
            thisAmt = 2;
            if (days > 2)
                thisAmt += (days-2)*1.5;
            break;
        case Movie.NEW:
            thisAmt = days*3;
            break;
        case Movie.CHILDREN:
            thisAmt = 1.5;
            if (days > 3)
                thisAmt += (days-3)*1.5;
            break;
    }
    return thisAmt;
}

statement()

public String Statement (){double totalAmt = 0;
    Iterator<Rental> iter = rentals.iterator();
    System.out.printf("Record for %s \n",getName());
    while(iter.hasNext()){Rental aRental=iter.next();
        double thisAmt=getAmount(aRental);
        String t=aRental.getMovie().getTitle();
        System.out.printf("\t %s ", t);
        System.out.printf("$%.2f\n", thisAmt);
        totalAmt=totalAmt+thisAmt;
    }
    System.out.printf("Amount owed is $%.2f\n",totalAmt);
Why is Design 1 not good enough?

- A large portion of the functionality is within the Customer class.
  - Redistribute responsibility to classes that contain the relevant information.
- If a new type of movie is added to the system (e.g., FAVORITES) then the switch statement in the Customer class must be extended and recompiled.
  - This is an undesirable coupling between the Customer and Movie class.

Design 2 Overview

- Redistribute functionality
  - Rental class: responsible for calculating rental charge
- Extend Customer interface
  - Customer class: interface extended to include a getTotalCharge() method that returns the sum of the charges for the customer.
Design 2 Class Diagram

- Customer
  - name: String
  + addRental()
  + getName()
  + statement()
  + getTotalCharge()

- Rental
  - daysRented: int
  + getDaysRented()
  + getMovie()
  + getCharge()

- Movie
  - title: String
  - priceCode: int
  + getPriceCode()
  + getPriceCode()
  + getTicket()

Design 2 Sequence Model

C: Customer

R: Rental

M: Movie

Loop for all rentals

statement()

ch := getCharge()

pc := getPriceCode()

t := getTitle

tc := getTotalCharge()

c := getCharge()
Design 2 Drawbacks

- Solution requires 2 loops through a customer’s rentals; one to record the charges for each rental, the other to calculate total charge.
  - Here is a compromise in which reuse is given greater weight than efficiency: by restricting getTotalCharge() to calculating only the sum of charges it can be used by other functions that require just that and nothing more; but this means that two loops are needed when printing out statements.
  - In design 3 I show the other approach in which efficiency outweighs reuse.

- Case statement problem not solved.

Design 2: Code for Rental

```java
class Rental{
    ...  
    double getCharge(){
        double thisAmt = 0;
        switch (movie.getPriceCode()){
            case Movie.REGULAR:
                thisAmt = 2;
                if (days > 2)
                    thisAmt += (days-2)*1.5;
                break;
            case Movie.NEW:
                thisAmt = days*3;
                break;
            case Movie.CHILDREN:
                thisAmt = 1.5;
                if (days > 3)
                    thisAmt += (days-3)*1.5;
                break;
        }
        return thisAmt;
    }
}```
Design 2: statement()

```java
public String statement (){
    Iterator<Rental> iter = rentals.iterator();
    System.out.printf("Record for %s \n",getName());
    while(iter.hasNext()){
        Rental aRental=iter.next();
        double thisAmt = aRental.getCharge();
        String t=aRental.getMovie().getTitle();
        System.out.printf("\t %s ", t);
        System.out.printf("$%.2f\n", thisAmt);
    }
    System.out.printf("Amount owed is $%.2f\n",getTotalCharge());
}
```

Design 2: getTotalCharge()

```java
private double getTotalCharge (){  
    double result = 0;
    Iterator<Rental> iter = rentals.iterator();
    while(iter.hasNext()){
        Rental aRental=iter.next();
        result += aRental.getCharge();
    }
    return result;
}
```
Design 3: Notes

- One may be tempted to subclass the Movie class but note that a Movie object can be a New Release, then a Regular movie over a period of time (thus the specialization is not static).

- Better to have the price determinant part of the Movie as a separate abstract class (*Price*) that is subclassed; a Movie object can then have a reference to the specific Price subclass object (New, Regular, Children).
  - This is an application of the State pattern to be discussed later in this course.

- In this design, *getTotalCharge* calculates more than just the total charge, it also documents the charges for each rental in the String object *result*.
  - Good aspect: program accesses rentals in one loop, unlike design 2 in which rentals are traversed twice.
  - Bad aspect: If only a total is needed the program does unnecessary work w.r.t. documenting individual rentals.

Design 3: Class Diagram
Design 3: Sequence Model

Price and Subclasses Code

class Price {
   \--
   abstract double getCharge(int daysRented);
}

class RegularPrice extends Price {
   double getCharge(int daysRented) {
      double result = 2;
      if (daysRented > 2)
         result += (daysRented-2)*1.5;
      return result;
   }
}

class NewRelPrice extends Price {
   double getCharge(int daysRented) {
      return daysRented*3;
   }
}

class ChildrenPrice extends Price {
   double getCharge(int daysRented) {
      result = 1.5;
      if (daysRented > 3)
         result += (daysRented-3)*1.5;
      return result;
   }
}
class Movie{
  \...
  double getCharge(int daysRented)
  {
    return priceCode.getCharge(daysRented);
  }
}

class Rental{
  \...
  double getCharge()
  {
    return movie.getCharge(daysRented);
  }
}

public String Statement (){
  System.out.printf("Record for %s \n",getName());
  System.out.printf("Amount owed is %.2f\n", getTotalCharge());
}
**getTotalCharge() Code**

```java
public double getTotalCharge (){
    Iterator<Rental> iter = rentals.iterator();
    double totalAmt = 0;
    while(iter.hasNext()){
        Rental aRental=iter.next();
        double thisAmt = aRental.getCharge();
        t=aRental.getMovie().getTitle();
        System.out.printf("t %s ", t);
        System.out.printf("$%.2f\n", thisAmt);
        totalAmt += thisAmt
    }
    return totalAmt;
}
```

**Conclusion**

"I knew we didn't have enough blocks for this thing."