

Unified Modeling Language

- Let's look at each of the words in the name
- Unified
 - Two important methodologists Rumbaugh and Booch decided to merge their approaches in 1994.
 - They worked together at the Rational Software Corporation
 - In 1995, another methodologist, Jacobson, joined the team
 - His work focused on use cases
 - In 1997 the Object Management Group (OMG) started the process of UML standardization

Week 7
Lecture Slide 1

What constitutes a good model?

- A model should
 - Provide abstraction
 - Render the problem in a format amenable to reasoning
 - use a standard notation
 - be understandable by clients and users
 - lead software engineers to have insights about the system
 - make the problem solvable computationally
 - Be good enough

Week 7
Lecture Slide 4

Models

- Models are abstract representations
 - Contain essential characteristics and omit non-essential details
 - "Essential" depends on the problem domain
 - There are no perfect representations
- Models can be representations of the world
 - Domain models
 - Requirements
- Models can be representations of software
 - Specifications
 - Design
 - Systems

Week 7
Lecture Slide 2

Remember: It's only a model

- There will always be:
 - Phenomena in the application domain that are not in the model
 - Details in the application that are not in the model
- A model is never perfect
 - "If the map and the terrain disagree, believe the terrain"
 - Perfecting the model is not always a good use of your time...

Week 7
Lecture Slide 5

Why make models?

- Systems are complex and hard to understand
 - The world, organizations, relationships, software
- Models can make certain aspects more clearly visible than in the real system
- What can you do with models?
 - Express your ideas and communicate with other engineers
 - Reason about the system: detect errors, predict qualities
 - Generate parts of the real system: code, schemas
 - Reverse engineer the real system to make a model

Week 7
Lecture Slide 3

Modeling Languages

- Natural language
 - Extremely expressive and flexible
 - Very poor at capturing the semantics of the model
 - Better used for elicitation, and to annotate models for communication
- Semi-formal notation
 - Captures structure and some semantics
 - Can perform (some) reasoning, consistency checking, animation, etc.
 - Examples: diagrams, tables, structured English, etc.
 - Mostly visual – for rapid communication with a variety of stakeholders
- Formal notation
 - very precise semantics, extensive reasoning possible
 - Every detailed models (may be more detailed than we need)

Week 7
Lecture Slide 6

Visual Languages

- Words = symbols
- Syntax = rules for combining symbols, drawing and layout of language
 - Examples: Sheet music, wiring diagram,



Week 7
Lecture Slide 7

UML and RUP in Context

- Tool
 - Rational Rose
- Method
 - RUP
- Notation
 - UML
- Process
 - Waterfall, incremental, evolutionary, not XP

Week 7
Lecture Slide 10

UML

- UML is a semi-formal visual modeling language
 - Semantics are not completely specified by standard
 - It has *extension* mechanisms
 - It has an associated textual language
 - *Object Constraint Language* (OCL)
 - Well suited for object-oriented designs

Week 7
Lecture Slide 8

Types of UML Diagrams

Structure

- Class diagrams
- Object diagram
- Package diagram
- Composite structure diagram
- Component diagram
- Deployment Diagram

Behavior

- Activity diagram
- Use case diagram
- State machine diagram
- Interaction diagrams
 - Sequence diagram
 - Communication diagram
 - Interaction overview diagram
 - Timing diagram

Week 7
Lecture Slide 11

Rational Unified Process

- RUP is the method that OMG advocates for performing analysis and design of systems
 - In other words, it's the method behind UML
 - Other methods are possible, but this one is best known
- We'll cover UML syntax first and then come back to process

Week 7
Lecture Slide 9

Types of UML Diagrams

Structure

- Class diagram
- Object diagram
- Package diagram
- Composite structure diagram
- Component diagram
- Deployment diagram

Behavior

- Activity diagram
- Use case diagram
- State machine diagram
- Interaction diagrams
 - Sequence diagram
 - Communication diagram
 - Interaction overview diagram
 - Timing diagram

Week 7
Lecture Slide 12

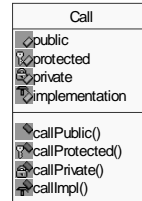
Ways of Categorizing UML Diagrams

- Structural vs. Behavioral
- Abstract vs. Concrete
- Modeling level (M0–M3)
 - M0: The concrete, non-abstract things actually in the world
 - M1: The model, a representation of some phenomena in the real world
 - M2: Ontology (vocabulary) and grammar of entities, relationships, processes from which a model can be built

Week 7
Lecture Slide 13

Classes, Attributes, and Operations

- Public (nothing or +)
- Protected (#)
- Private (-)



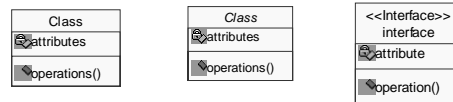
Week 7
Lecture Slide 16

Structure or Behavior First?

- When modeling a system, should you start with structure or behavior?
- Some authors strongly advocate one or the other
 - Example: XP
 - Example: CRC cards
- It's good to have a method that you're comfortable with, but it depends on the problem
 - Some applications are data-centric, others are algorithm-centric.
- We'll go through the notations quickly and spend the rest of the time on methods and design case studies.

Week 7
Lecture Slide 14

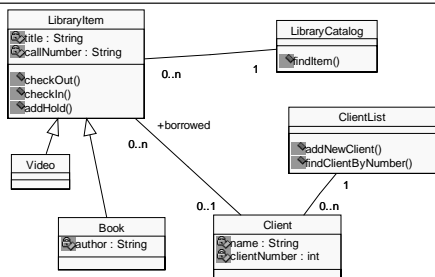
Class Diagrams



- Name
 - Name: type
- Attributes
 - visibility name: type multiplicity = default {property-string}
- Operations
 - visibility name (parameter-list) : return-type {property-string}
 - direction name: type = default

Week 7
Lecture Slide 17

Class Diagram



- A UML class corresponds to a Java class.

Week 7
Lecture Slide 15

Attribute Syntax

visibility name: type multiplicity = default {property-string}

- optional visibility: + public, - private, # protected
- name: the name of this attribute
- optional type: data type of this attribute
- optional default: initial value of attribute
- optional property string: OCL, e.g. ordered, readonly

- Examples:
 - firstName
 - middleName
 - lastName: String
 - age: int = 0
 - birthSign: String = "Gemini"

Week 7
Lecture Slide 18

Operation Syntax

- visibility name (parameter-list) : return-type**
{property-string}
- optional visibility: + public, - private, # protected
 - name: the name of this operation
 - optional parameters-list: parameters to this operation
- direction name: type = default**
- optional direction: in, out, inout
 - name
 - optional type
 - optional default
- optional type: return type of operation
 - optional property string

Week 7
Lecture Slide 19

Property String

- Any information that cannot be expressed in the diagram notation can be included as text
 - Example: constraints
- Example:
 - customerNumber: int { >=0 }
- All diagram elements can be annotated with constraints
- Can be:
 - Natural language text
 - Object Constraint Language
 - Predefined properties
 - Examples on next two slides
 - Any other text

Week 7
Lecture Slide 22

Operation Syntax

- Examples:
 - getFirstName()
 - +getMiddleName(): String
 - setLastName(name: String)
 - +paintPortrait(inout c: Canvas, subject: Person)

Week 7
Lecture Slide 20

Property Strings on Attributes

- changeable (default)
 - Value of attribute can be changed
 - May want to list legal/possible values
- addOnly
 - Can add possible values, but can't change existing ones
- frozen
 - Can't add or change

Week 7
Lecture Slide 23

Multiplicities

- Descriptive
 - Optional (0 or more)
 - Mandatory (at least 1)
 - Single-valued (upper bound 1)
 - Multi-valued (upper bound of >1, usually *)
- Symbolic
 - 1 (or another number, exactly the number specified)
 - 0..1 (zero or one, i.e. optional)
 - 2..4 (range)
 - * (zero or more, no upper limit; n in Rose)

Week 7
Lecture Slide 21

Property Strings on Operations

- sequential
 - Only one call to a method within an instance
- concurrent
 - Multiple simultaneous calls to a method within an instance may occur
- guarded
 - Multiple simultaneous calls to a method within an instance may occur but only one at a time will be executed
- isQuery
 - Operation doesn't change the value of any attributes


Week 7
Lecture Slide 24

Inheritance

- Generalization correspond to the Java keyword "extends"
 - Generalizations are drawn with a solid line and a white triangular arrow touching the superclass.
- Realization correspond to the Java keyword "implements"
 - Realizations are drawn with a dashed line and a white triangular arrow touching the interface.
- UML itself is not restricted to single inheritance. However, you would not use multiple inheritance if you plan to implement in Java.

Week 7
Lecture Slide 25

Associations



- Relations between classes
- Roles
 - analogous to names of instance variables
- Multiplicities
 - 0, 1, *, 0..1, 1..*, 5..6, and so on
 - says how many objects each object knows
 - would be realized through arrays, Sets, Lists, and so on
- Navigability
 - bidirectional: each class references the other
 - unidirectional: A knows B, but B doesn't know A
 - no arrow heads: means either "bidirectional" or "not specified"

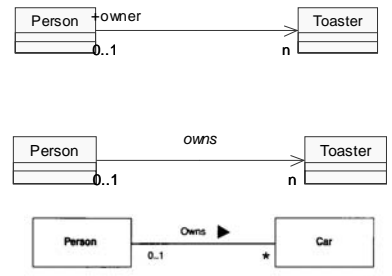
Week 7
Lecture Slide 26

Inheritance

- It is common practice to arrange the diagram so that:
 - Generalization and Realization arrows point upward
 - If one class has many subclasses, the Generalization arrows overlap

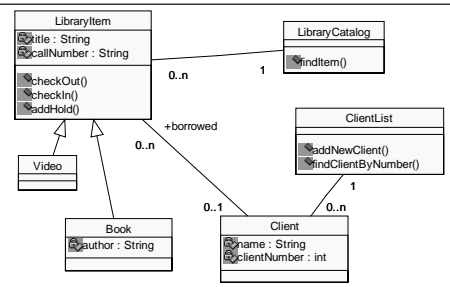
Week 7
Lecture Slide 26

Association Name vs. Role Names



Week 7
Lecture Slide 29

Class Diagram



Week 7
Lecture Slide 27

Types of Associations

- Composition = Black diamond: parts are created with the whole and stay with exactly one whole until both are destroyed together.
- Aggregation = White diamond: parts can join the whole and later leave; one part could be part of more than one whole.
 - Some operations will be recursively applied to the parts of a whole

Week 7
Lecture Slide 30

Association Type: Composition

- A *composition* is a strong kind of aggregation
 - if the aggregate is destroyed, then the parts are destroyed as well

Week 7
Lecture Slide 31

Example Aggregation

Week 7
Lecture Slide 34

Association Type: Aggregation

- Aggregations are special associations that represent 'part-whole' relationships.
 - The 'whole' side is often called the *assembly* or the *aggregate*
 - This symbol is a shorthand notation association named *isPartOf*

Week 7
Lecture Slide 32

Some Mnemonics

- Generalization = is-a
- Composition = has-a
- Aggregation = part-of

- Examples:
 - Toaster is-a Appliance
 - Toaster has-a slot
 - Member part-of club

Week 7
Lecture Slide 35

When to use an aggregation

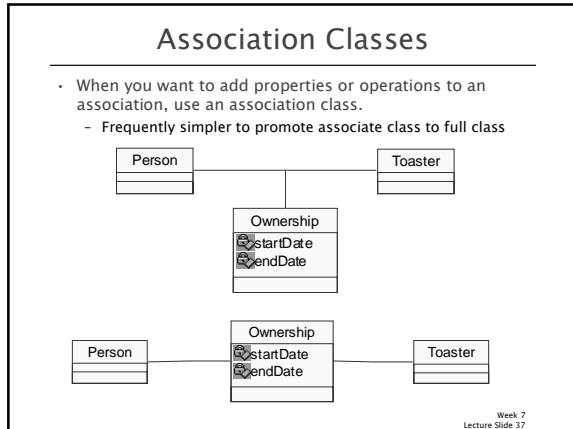
- As a general rule, you can mark an association as an aggregation if the following are true:
 - You can state that
 - the parts 'are part of' the aggregate
 - or the aggregate 'is composed of' the parts
 - When something owns or controls the aggregate, then they also own or control the parts
- Use with care

Week 7
Lecture Slide 33

Associations and Properties

- Associations are another way of representing properties

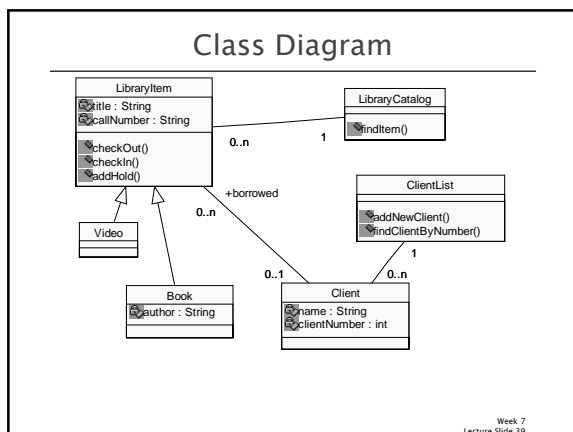
Week 7
Lecture Slide 36



- ### Hints for Class Diagrams
- Remember: models are for communication
 - Remember: include only important stuff
 - How do I find classes, attributes and so on?
 - Classes often correspond to nouns
 - Associations often correspond to verbs
 - A class should
 - Represent a coherent concept
 - Principle: Low Coupling, High Cohesion
 - Have a small, well-defined set of responsibilities
 - Be named with a singular noun that says what each instance of the class is
 - Have no more than 10-20 operations
- Week 7
Lecture Slide 40

- ### More in Scott
- Keywords (<<guillemets>>)
 - Stereotypes
 - Examples: calls, interface, permit,...
 - Responsibilities
 - Another compartment in class
 - Items prefixed by --
 - Template classes
- Week 7
Lecture Slide 38

- ### Hints for Class Diagrams
- Class diagrams should
 - have a single purpose
 - have a title that expresses the purpose
 - show only things that are relevant for this purpose
 - Avoid
 - cyclical dependencies, if possible
 - generalization hierarchies with more than 5 levels
 - crossing edges
- Week 7
Lecture Slide 41



- ### Hints for Class Diagrams
- Use colors judiciously
 - to highlight and group things
 - unless you have to print it in black-and-white!
 - Lay out classes in a meaningful way
 - similar classes close to each other
 - top: closer to the user, bottom: closer to the data structures
- Week 7
Lecture Slide 42

Object Diagrams

- Show instantiation or specification of classes
- Associated with a particular use or instance of the model
- Differences between Classes and Objects
 - Name: class is underlined
 - Attributes and operations included as needed
 - Fields have data added
- Useful for showing interactions between interfaces, abstract classes, etc.
 - Where functionality is not clear until instantiation

Week 7
Lecture Slide 43

Package Diagrams

- Package is a grouping construct
 - Most commonly used for class diagrams, but can be used with any UML diagram or elements
 - Used to create a hierarchy or higher level of abstraction
 - Corresponds to package in Java
- Each package represents a namespace
 - Like Java, can have classes with same name in different packages

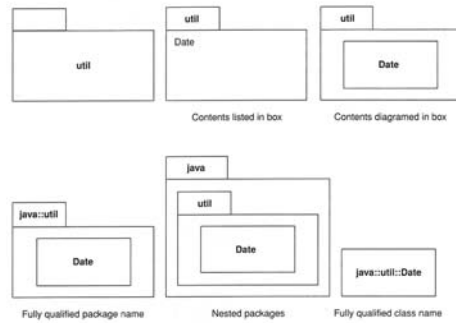
Week 7
Lecture Slide 46

Object Diagrams

- Show instantiation or specification of classes
- Associated with a particular use or instance of the model
- Differences between Classes and Objects
 - Name: class is underlined
 - Attributes and operations included as needed
 - Fields have data added
- Useful for showing interactions between interfaces, abstract classes, etc.
 - Where functionality is not clear until instantiation

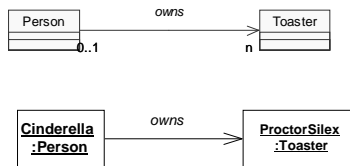
Week 7
Lecture Slide 44

Representing Packages



Week 7
Lecture Slide 47

Example: Class to Object Diagrams



Week 7
Lecture Slide 45

Types of UML Diagrams

Structure

- Class diagram
- Object diagram
- Package diagram
- Composite structure diagram
- Component diagram
- Deployment diagram

Behavior

- Activity diagram
- Use case diagram
- State machine diagram
- Interaction diagrams
 - Sequence diagram
 - Communication diagram
 - Interaction overview diagram
 - Timing diagram

Week 7
Lecture Slide 48

Behavior Diagrams

- Extended UML object diagrams with messages
- Show behavior of a set of objects
 - dynamic
 - have a time scale
- Like all structure diagrams: can't be complete
 - show only an example of the behavior
- Seven Types of UML behavior diagrams
 - sequence diagrams
 - use case diagrams
 - state machine diagrams

Week 7
Lecture Slide 49

Why write use cases?

- Use cases are fast and easy to write (after some practice). They require no training to read.
- Use cases are fairly concrete examples of how the system could be used. They are easier to understand than other, more abstract, ways of writing requirements.
- Use cases are examples of usage. Starting with just a few examples is easy. More examples can be added incrementally as needed.
- Use cases can be reviewed by non-technical stakeholders to help validate requirements.

Week 7
Lecture Slide 52

Use Cases

- A *use case* is a typical sequence of actions that a user performs in order to complete a given task
 - The objective of *use case analysis* is to model the system
 - ... from the point of view of how users interact with this system
 - ... when trying to achieve their objectives.
- A *use case model* consists of
 - a set of use cases
 - an optional description or diagram indicating how they are related
- A requirements gathering technique

Week 7
Lecture Slide 50

Why write use cases?

- Use cases can easily be refined from user stories, and can later be refined into diagrams and test cases.
- Use cases cut across the features of the product, thus providing a good complement to the more abstract, individual feature specifications.

Week 7
Lecture Slide 53

Use Cases

- In general, a use case should cover the full sequence of steps from the beginning of a task until the end.
- A use case should describe the user's interaction with the system ...
 - not the computations the system performs.
- A use case should be written so as to be as independent as possible from any particular user interface design.
- A use case should only include actions in which the actor interacts with the computer.

Week 7
Lecture Slide 51

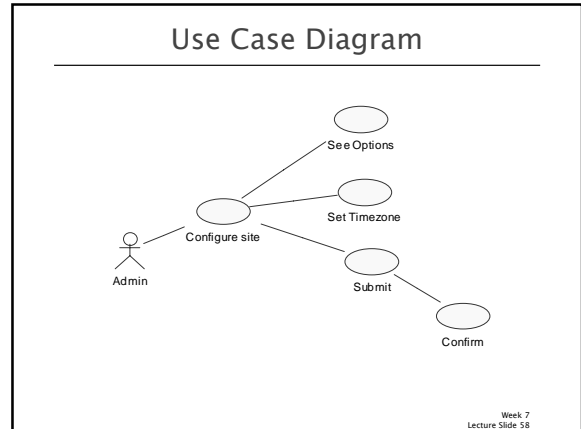
How to write use cases

- A use case template, like the ReadySET use case template, is helpful
- Start with a Main Success Scenario (MSS)
 - The use case should focus on one success.
 - Possible error conditions or other failures are simply noted as extensions to the main success scenario.
- Provide a descriptive name for the use case
- Identify actors or roles
 - Can be users or another software system
- Detail the steps to be taken to achieve the desired goal
 - Include enough detail to check for correctness

Week 7
Lecture Slide 54

UC-00: Configure the site	
Summary:	The administrator navigates to the site configuration page and uses it to change the behavior of the web application. Specifically, the user-visible timezone is being set.
Priority:	Essential
Use Frequency:	Rarely
Direct Actors:	Admin: Web-site administrator
Main Success Scenario:	1.visit SiteConfiguration page 2.see site configuration options 3.enter timezone abbreviation for date displays 4.submit form 5.confirm changes 6.see SiteConfiguration page again, with updated values
Alternative Scenario Extensions:	If the timezone abbreviation is incorrect, an error message will be displayed and no changes will be made.
Notes and Questions	How will administrators know the right timezone abbreviation? They would know it if they live in that timezone. Maybe we could provide a dropdown list of all choices, but each would need some explanation.

Week 7
Lecture Slide 55



- ### How to write use cases
- As you go, add notes to yourself and questions to come back to.
 - Are there any preconditions that you assumed?
 - Are there alternative ways to accomplish the same thing? E.g., cash withdrawal vs. fast cash. You can note them as extensions, or write whole new use cases.
 - Are there ways that the user can fail? Note some as extensions.
- Week 7
Lecture Slide 56

- ### Elements of Use Case Diagrams
- The main purposes of a use case diagram:
 - Show all the names of the use cases, like a table of contents
 - Show relationships between actors and use cases
 - Show relationships between use cases
 - Stick figure: Actor
 - Oval: Use case
 - Extension
 - Optional interactions to cover exceptions
 - Inclusion
 - For common substeps; can be re-used in diagram
 - Generalization
 - Like superclasses; for representing several similar use cases
- Week 7
Lecture Slide 59

