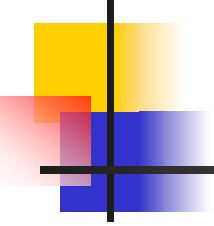


- 
- (1) Introduction to UML
 - (2) Type Hierarchy
-

Lecture 5: OOP, autumn 2003



Introduction to UML



Software abstractions and architectures

- Concepts so far:
 - Procedural abstraction: methods, exceptions
 - Data abstraction: objects, types, classes, interfaces, modules
 - Specifications, invariants
- New concepts: inheritance, polymorphism, etc.
- Relationships, interaction among concepts
- Putting it all together – **architecture**:
 - **structure of the components of a program/system,**
 - **their interrelationships,**
 - **and principles and guidelines governing their**
 - **design and**
 - **evolution over time.**
- Real architectures are very complex: buildings, airplanes, software
- How to describe, reason and communicate architectures?



Modeling

- What is a model?
 - **A description of observed behavior, simplified by ignoring certain details** (www.foldoc.org)
 - **A representation of one system by another, usually more familiar, whose workings are supposed analogous to that of the first.** (Oxford dictionary of philosophy)
- Why model?
 - Bridge the gap between problem and solution
 - Articulate and communicate ideas between project members
 - Represent abstractions to manage complexity
 - Provide structure for problem solving
 - Manage the risk of mistakes
- What do we model?
 - reality vs. understanding
- How to describe, communicate, visualize software models?
 - Textual vs. graphical representation
 - Invent your own notation
 - The Liskov book graphical notation
 - Standardized notation



Unified Modeling Language (UML)

- Visual modeling language for
 - specifying
 - constructing
 - documentingthe artifacts of software systems
- Origins
 - Booch method - Grady Booch
 - Object Modelling Technique (OMT) – J. Rumbaugh
 - Object-Oriented Software Engineering (OOSE) – I. Jacobson
- OMG standard (www.omg.org/uml)
 - Current version 1.4
 - Version 2.0 on the way



Unified Modeling Language (UML)

- Goals of UML:
 - Be expressive and relatively simple
 - Provide extensibility of its core concepts
 - Be independent of
 - programming languages
 - development processes
 - Support high-level concepts as: components, collaborations, frameworks, patterns
 - Integrate best practices
- What UML is NOT:
 - Programming language
 - Process (there is no 'how', only 'what')
 - Modeling tool



UML diagrams

- Buildings, cars - multiple blueprints
 - Frame
 - Wiring
 - Pipes
- Software - multiple diagrams
 - Use small set of nearly independent views of a model.
 - No single view is sufficient.
 - Model at different levels of fidelity.
 - The best models are connected to reality.
- In UML: models ↔ diagrams



UML diagram elements

- Graph = nodes + paths
- Information is in topology, not size/position
- Three kinds of visual relationships
 - connection (usually of lines to 2-d shapes),
 - containment (of symbols by 2-d shapes with boundaries), and
 - visual attachment (one symbol being “near” another one on a diagram).
- Four kinds of graphical constructors
 - Icons – are atomic, can’t hold other elements
 - 2-D symbols – can hold other things, can be divided into compartments
 - Paths – both ends always attached, may have terminators
 - Strings



UML types of diagrams

- **Structural Diagrams (static)**
 - *Class diagram, Object diagram, Component diagram, and Deployment diagram.*
- **Behavior Diagrams (dynamic)**
 - Use Case diagram, Sequence diagram, Activity diagram, Collaboration diagram, and Statechart diagram.
- **Model Management Diagrams**
 - Packages, Subsystems, and Models



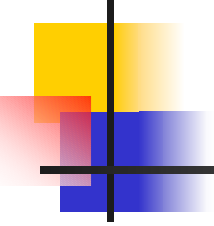
Class diagrams

- Classes => 2D symbols with three compartments: name, attributes, operations
- Can be abbreviated (omit attributes, operations) => *abstraction*
- Same class in many diagrams => *modularity*
- Relationships between classes (single/bi-directional)
 - Association
 - Aggregation
 - Composition (aggregation with life-time dependency)
 - **Generalization**
 - Dependency
 - Interfaces



Object diagrams

- Derived from class diagrams
- Graph of instances
 - Objects
 - Data values
 - Shows snapshot of the system
 - Used to show examples
- Object structure
 - Name
 - State - unspecified language syntax
 - Composite objects (imply composition of classes)



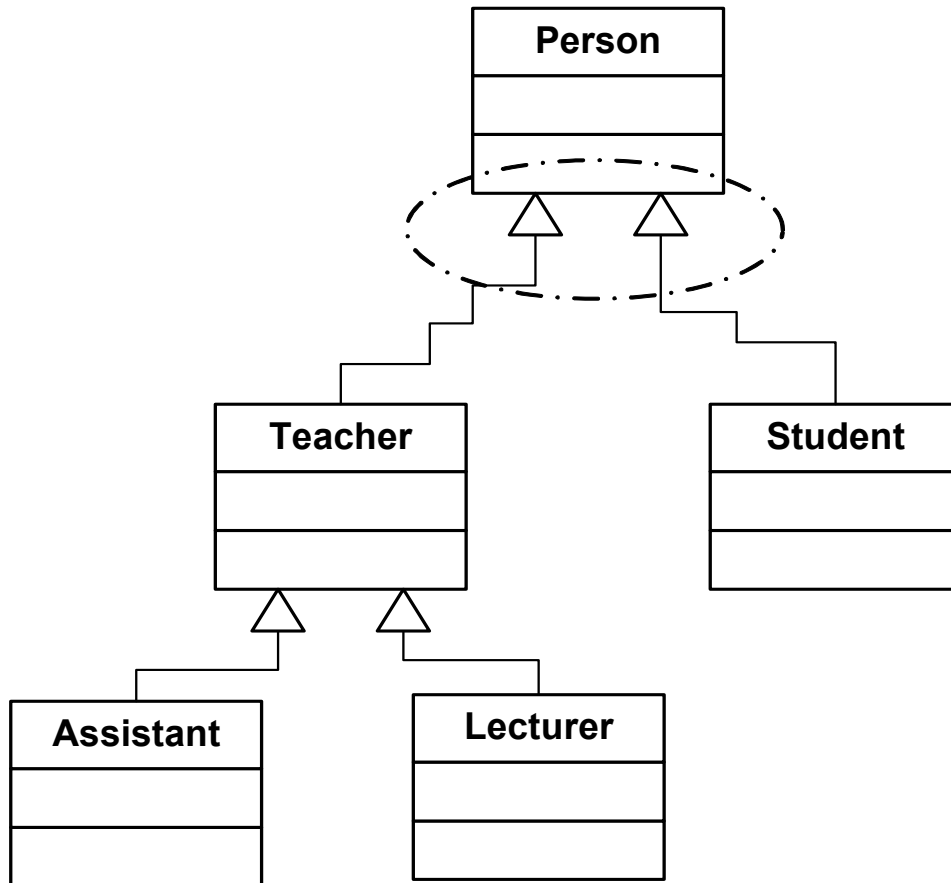
Type Hierarchy



The idea

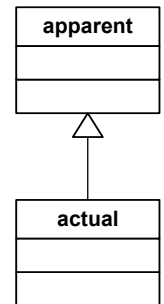
- Many types have common behavior
- Can be grouped into *type families*
- Members share common behavior
- Can be organized into a hierarchy
 - Most common on the top - **supertypes**
 - Most specific at the bottom - **subtypes**
 - Many levels
- Very common way of abstraction in 'real' life
- Substitutability => abstraction by specification
- Typically depicted as upside-down tree
 - Most general at top
 - Most specific at bottom
- Used to
 - Provide different implementations
 - Extend the behavior

UML notation



Assignment with inheritance

- Substitutability => 'looser' assignment
- Variables can hold all objects of a subtypes
 - **apparent** types - compile time
 - **actual** types - run time
- Type checking
 - Programs can use only supertype methods





Dispatching

- Objects at run-time can be of subtypes
- Compile type-checks not sufficient
- Code can't execute methods directly
- Need to find actual implementation at run-time
- Use dispatch vectors



Type hierarchies in Java

- Various languages limit the definition of type hierarchies
- In Java:
 - Inheritance mechanism
 - Classes and interfaces
 - **Concrete** classes - full implementation
 - **Abstract** classes - partial **abstract** implementation, no objects
 - **Interfaces** - only specification
 - **final** and **abstract** methods
 - Reimplementation \Leftrightarrow **overriding**
 - Classes **extend** superclasses
 - Methods **and** attributes are **inherited**
 - Visibility - private, public, **protected**
 - The **super(...)** constructor and **super** keyword
 - Only methods of the most specific class are executed



Overloading vs. overriding

- Overloading - abstract similar behavior in same class
 - Reuse the same name for similar behavior
 - Methods differ in parameters
- Overriding - replace functionality in subclasses
 - Methods have same signatures
 - Different implementations



The substitution principle

- Enables reasoning using only supertype specification (abstraction again!)
- Any subtype object can be used in place of a supertype object without affecting using code
- Requirements
 - Signature rule - inherit all methods, compatible signature
 - Methods rule - similar behavior
 - Properties rule - preserve all superclass properties (e.g. invariants)