SEG4210 – Advanced Software Design and Reengineering

TOPIC 1

Review of UML

5.1 What is UML?

The Unified Modelling Language is a standard graphical language for modelling object oriented software

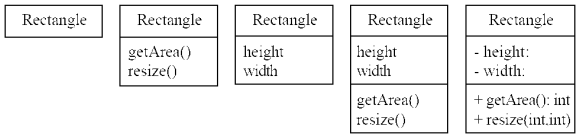
- At the end of the 1980s and the beginning of 1990s, the first object-oriented development processes appeared
- The proliferation of methods and notations tended to cause considerable confusion
- Two methodologists Rumbaugh and Booch merged their approaches in 1994.
  - —They worked together at the 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

Classes

A class is simply represented as a box with the name of the class inside

- The diagram may also show the attributes and operations
- The complete signature of an operation is:

operationName(parameterName: parameterType …): returnType

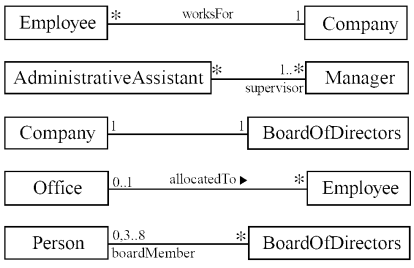


Naming classes

- Use *capital* letters
  - —E.g. `BankAccount` not `bankAccount`
- Use *singular* nouns
- Use the right level of generality
  - —E.g. `Municipality`, not `City`
- Make sure the name has only *one* meaning
  - —E.g. 'bus' has several meanings

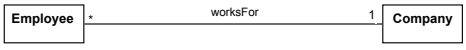
Labelling associations

- Each association can be labelled, to make explicit the nature of the association



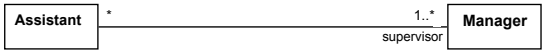
Analyzing and validating associations

- **Many-to-one**
  - —A company has many employees,
  - —An employee can only work for one company.
    - This company will not store data about the moonlighting activities of employees!
  - —A company can have zero employees
    - E.g. a 'shell' company
  - —It is not possible to be an employee unless you work for a company



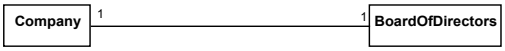
Analyzing and validating associations

- **Many-to-many**
  - —A secretary can work for many managers
  - —A manager can have many secretaries
  - —Secretaries can work in pools
  - —Managers can have a group of secretaries
  - —Some managers might have zero secretaries.
  - —Is it possible for a secretary to have, perhaps temporarily, zero managers?



Analyzing and validating associations

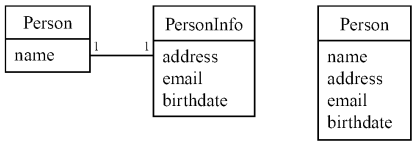
- **One-to-one**
  - —For each company, there is exactly one board of directors
  - —A board is the board of only one company
  - —A company must always have a board
  - —A board must always be of some company



Analyzing and validating associations

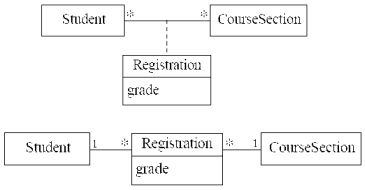
Avoid unnecessary one-to-one associations

Avoid this **do this**



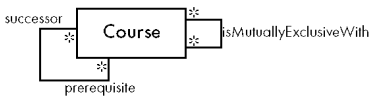
Association classes

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



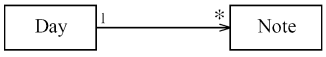
Reflexive associations

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



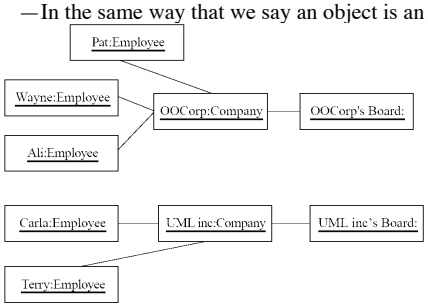
Directionality in associations

- Associations are by default *bi-directional*
- It is possible to limit the direction of an association by adding an arrow at one end

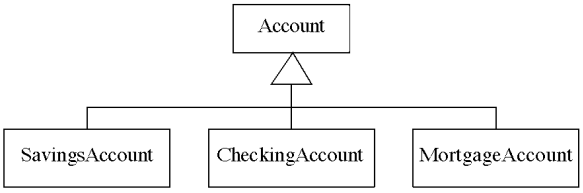


Object Diagrams

- A *link* is an instance of an association
  - —In the same way that we say an object is an instance



An Example Inheritance Hierarchy



Inheritance

- The *implicit* possession by all subclasses of features defined in its superclasses

The Isa Rule

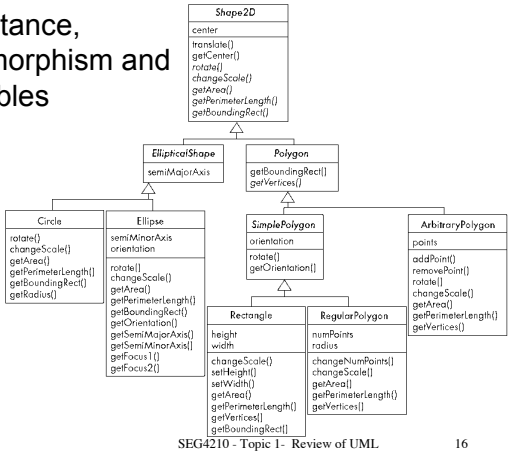
Always check generalizations to ensure they obey the isa rule

- "A checking account ***is an*** account"
- "A village ***is a*** municipality"

Should 'Province' be a subclass of 'Country'?

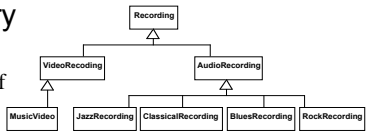
- No, it violates the isa rule
  - —"A province ***is a*** country" is invalid!

Inheritance, Polymorphism and Variables

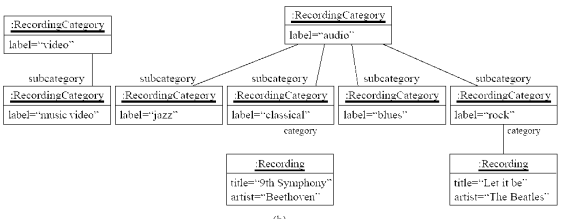


Avoiding unnecessary generalizations

Inappropriate hierarchy of classes, which should be instances

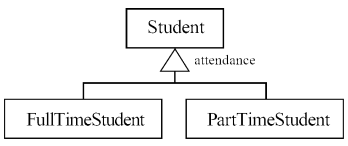


Improved class diagram, with its corresponding instance diagram



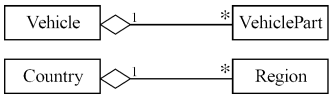
Avoiding having instances change class

- An instance should never need to change class



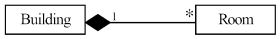
5.6 More Advanced Features: Aggregation

- 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`

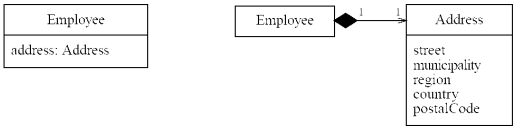


Composition

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



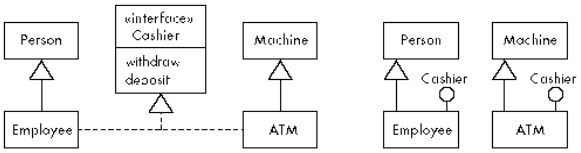
- Two alternatives for addresses



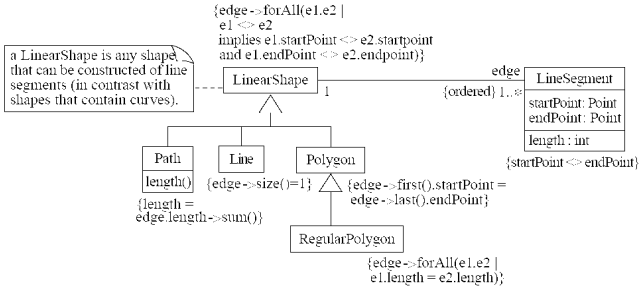
Interfaces

An interface describes a *portion of the visible behaviour* of a set of objects.

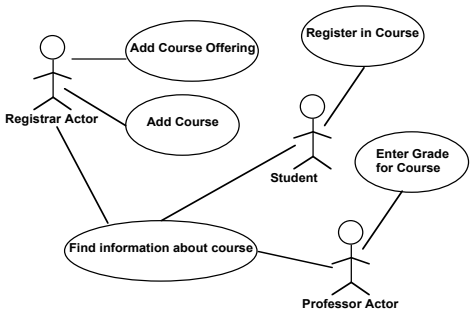
- An *interface* is similar to a class, except it lacks instance variables and implemented methods



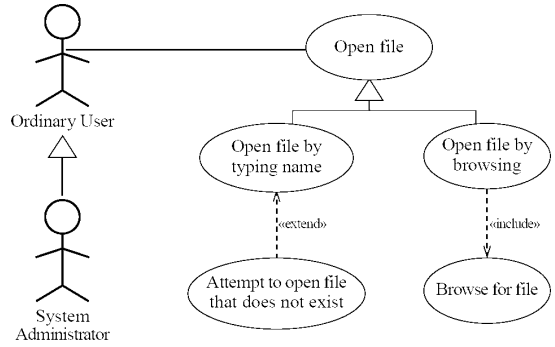
Introduction to OCL



Use Case Diagrams



Example of generalization, extension and inclusion



Example Description of a Use Case

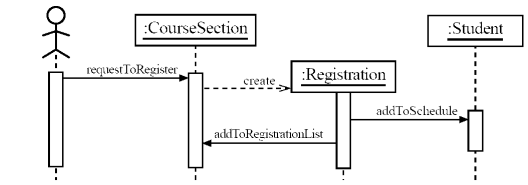
Use case: **Open file**

Related use cases:
Generalization of:
- Open file by typing name
- Open file by browsing

Steps:

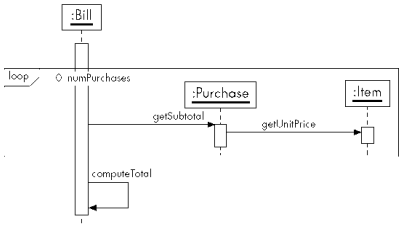
Actor actions	**System responses**
1. Choose 'Open…' command	2. File open dialog appears
3. Specify filename	
4. Confirm selection	5. Dialog disappears

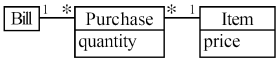
Sequence diagrams – an example



Sequence Diagrams – an example with replicated messages

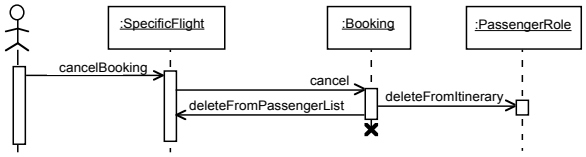
- An *iteration* over objects is indicated by an asterisk



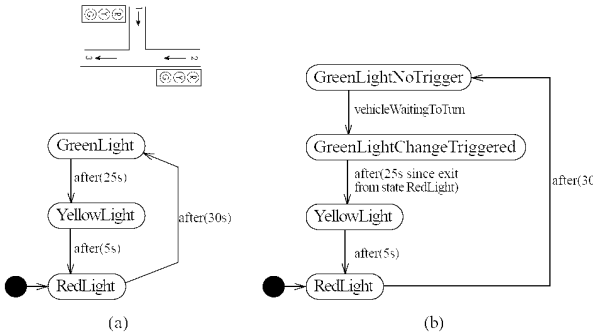


Sequence Diagrams – an example with object deletion

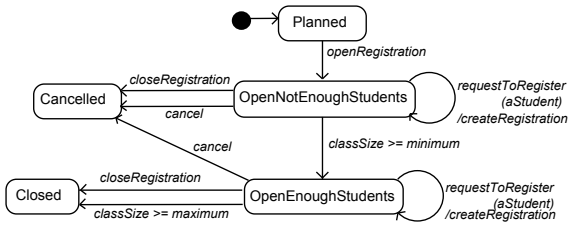
- If an object's life ends, this is shown with an X at the end of the lifeline



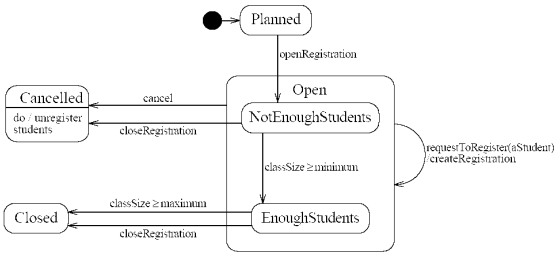
State Diagrams – an Example of Transitions with Time-outs and Conditions



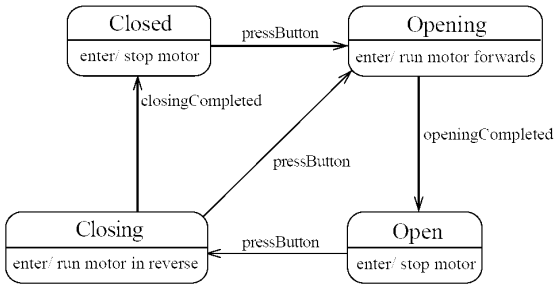
State Diagrams – Example with Conditional Transitions - CourseSection class



State Diagram – An Example with Substates CourseSection Class Again



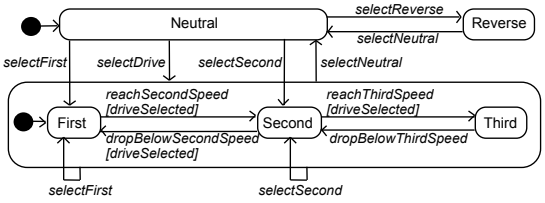
State Diagram – an Example with Actions – Garage Door Opener



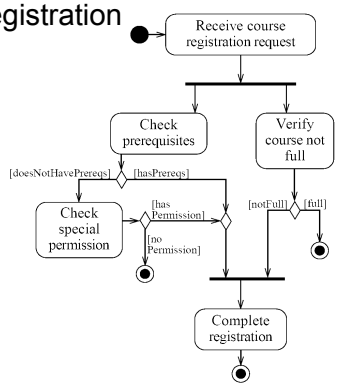
Nested Substates and Guard Conditions – A Car's Automatic Transmission

A state diagram can be nested inside a state.

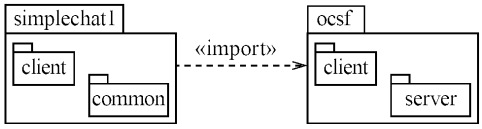
- The states of the inner diagram are called *substates*.



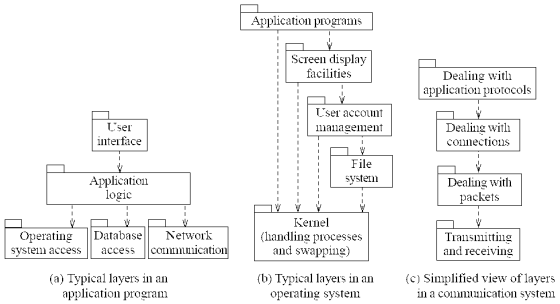
Activity Diagrams – An Example – Course Registration



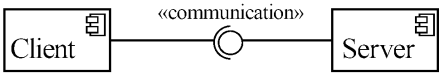
Package Diagrams



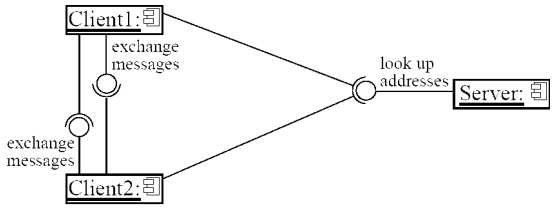
Example of Multi-Layer Systems



Component Diagrams

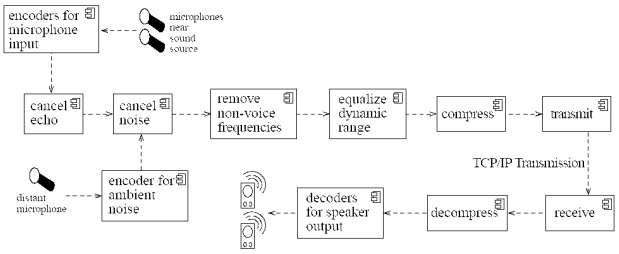


An Example of a Distributed System

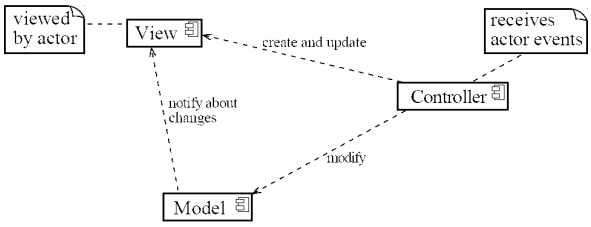


Example of a Broker System

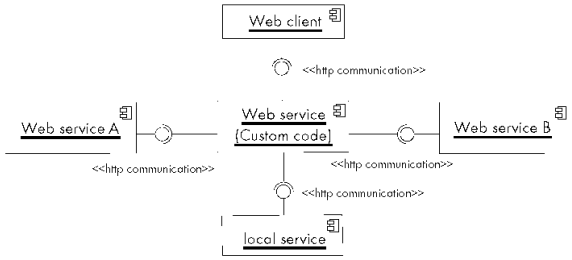
Example of a Pipe-and-Filter System - Sound Processing



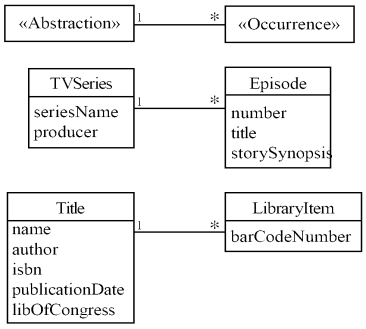
Example of the MVC Architecture for a User Interface



Example of the Service-Oriented Architecture

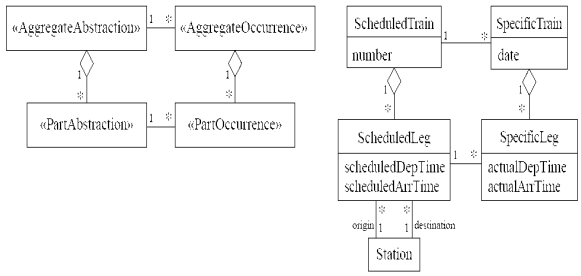


Abstraction-Occurrence

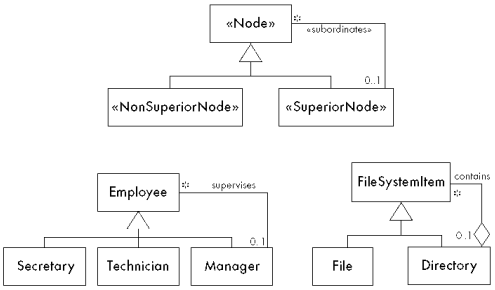


Abstraction-Occurrence

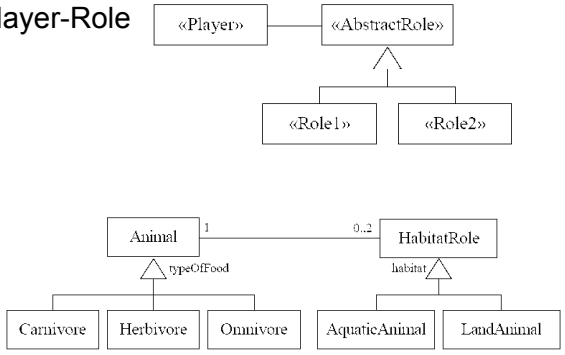
Square variant



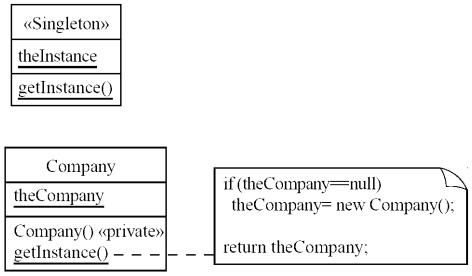
General Hierarchy



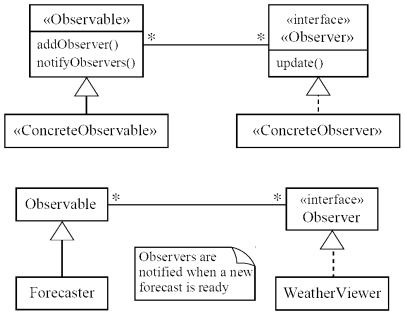
Player-Role



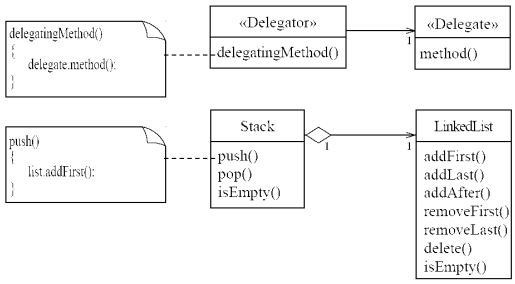
Singleton



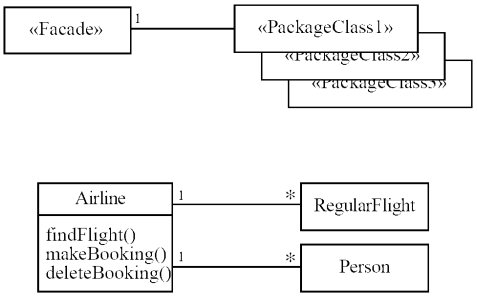
Observer



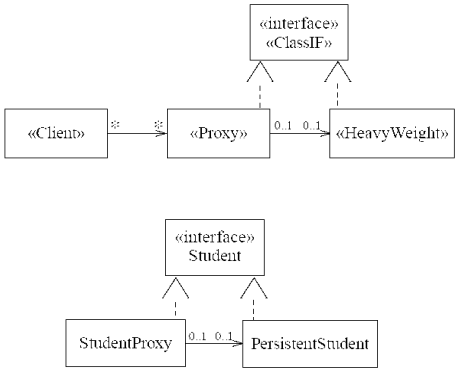
Delegation



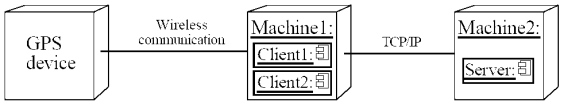
Façade



Proxy



Deployment Diagrams



SEG4210 – Advanced Software Design and Reengineering

TOPIC 2
UML Extension Mechanisms

Why an extension mechanism?

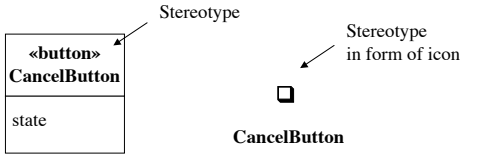
- Although UML is very well-defined, there are situations in which it needs to be customized to specific problem domains
- UML extension mechanisms are used to extend UML by:
  - adding new model elements,
  - creating new properties,
  - and specifying new semantics
- There are three extension mechanisms:
  - stereotypes, tagged values, constraints and notes

Stereotypes

- Stereotypes are used to extend UML to create new model elements that can be used in specific domains
- E.g. when modeling an elevator control system, we may need to represent some classes, states etc. as
  - «hardware»
  - «software»
- Stereotypes should always be applied in a consistent way

Stereotypes (cont.)

- Ways of representing a stereotype:
  - Place the name of the stereotype above the name of an existing UML element (if any)
    - The name of the stereotype needs to be between «» (e.g. «node»)
    - Don't use double '<' or '>' symbols, there are special characters called open and close guillemets
  - Create new icons

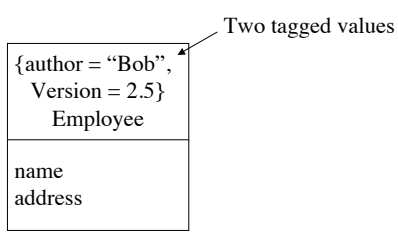


Tagged Values

- Tagged values
  - Define additional properties for any kind of model elements
  - Can be defined for existing model elements and for stereotypes
  - Are shown as a tag-value pair where the tag represent the property and the value represent the value of the property
- Tagged values can be useful for adding properties about
  - code generation
  - version control
  - configuration management
  - authorship
  - etc.

Tagged Values (cont.)

- A tagged value is shown as a string that is enclosed by brackets {} and which consists of:
  - the tag, a separator (the symbol =), and a value

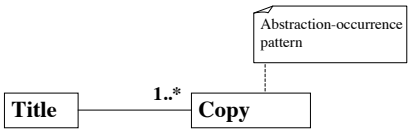


Constraints

- Constraints are used to extend the semantics of UML by adding new rules, or modifying existing ones.
- Constraints can also be used to specify conditions that must be held true at all times for the elements of a model.
- Constraints can be represented using the natural language or OCL (Object Constraint Language)
  - We will learn more about OCL in future lectures

Comments

- Comments are used to help clarify the models that are being created
  - e.g. comments may be used for explaining the rationale behind some design decisions
- A comment is shown as a text string within a note icon.
- A note icon can also contain an OCL expression



The UML Metamodel

- A metamodel is a model representing the structure and semantics of a particular set of models
- A UML model is an instance of the UML metamodel
- The UML metamodel
  - Describes the UML model elements
  - Is defined using a subset of UML
  - Is organized in the form of packages

The UML Metamodel (cont.)

- UML metamodel is defined according to the following concepts:
  - Abstract Syntax: The metamodel of UML is described using UML class diagrams
  - Well-formedness rules: Well-formedness rules are used to express constraints on the model elements
    -E.g. a class cannot have two names
  - Semantics: describes using the natural language the semantics of the model elements
- We will learn more about metamodelling in a later lecture

UML Profiles

- UML Profiles provide an extension mechanism for building UML models for particular domains
  - e.g. real-time systems, web development, etc…
- A profile consists of a package that contains one or more related extension mechanisms (such as stereotypes, tagged values and constraints)
  - that are applied to UML model elements
- Profiles do not extend the UML metamodel. They are also called *the UML light-weight extension mechanism*

UML Profiles (cont.)

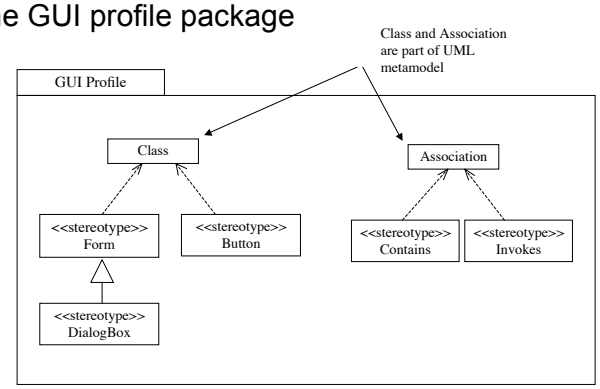
- A UML profile is a specification that does one or more of the following:
  - Identifies a subset of the UML metamodel (which may be the entire UML metamodel)
  - Specifies stereotypes and/or tagged values
  - Specifies well-formedness rules beyond those that already exist
  - Specifies semantics expressed in natural language

Example of a profile

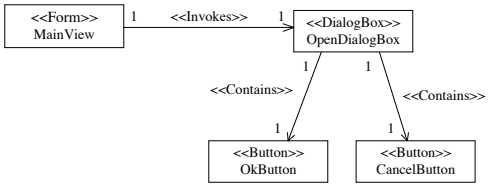
inspired by the research report of Cabot et al. (2003)

- We would like to create a UML profile for representing basic GUI components.
- We suppose that our GUI contains the following components:
  - Forms (which can also be dialog boxes)
  - Buttons
- Constraints: (in practice, we need to be more precise)
  - A form can invoke a dialog box
  - A form as well as a dialog box can contain buttons

The GUI profile package



Instance Diagram of the GUI Profile

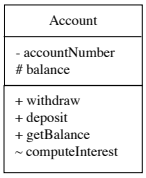


SEG4210 – Advanced Software Design and Reengineering

TOPIC 3
Advanced UML and Umple

Class Access

- Visibility operators define the scope of the attributes and methods of the class.
- Possibilities are:
  - public (+): access is granted to any other classes
  - protected (#): access is granted to classes that are derived from this class
  - private(-): access is granted to the defining class only
  - package(~): access granted to classes in the same package (or nested subpackages)

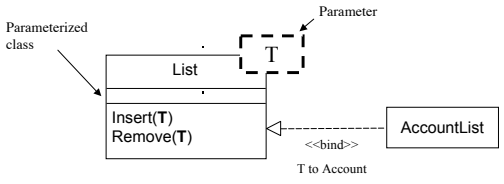


Parameterized Classes (Templates)

- Class templates provide means for *generic programming*
- Defining a generic class enables different type-specific versions of such a class to be generated
  - no need to repeatedly writing the class code for each type
- A class template contains one or more **unbound formal parameters**
- To use the template, we need to bound its parameters to actual values
  - This creates a bound form, which is the usable class
- In UML, parameterization does not apply only to classes
  - it can also be applied to packages and collaborations

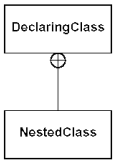
Parameterized Classes (Template)

- Class List represents a collection of objects of *a class*
- That class becomes a parameter to the parameterized class
  - *e.g. list of accounts*



Nested Classes

- A nested class is a class that is defined within another class
- Nested classes are used for implementation convenience rather than for information hiding.
- The nested class belongs to the namespace of the declaring class and may only be used within it



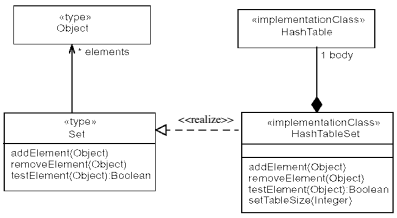
Notation

Type and Implementation Classes

- A Type class is used to specify **a domain** of objects without defining the physical implementation of those objects
- An Implementation Class defines the physical data structure as implemented in traditional languages (Java, C++, Smalltalk, etc.).
- An Implementation Class is said to *realize* a Type if it provides all of the operations defined for the Type
  - an Implementation Class may realize a number of different Types
- Type and Implementation classes are useful for defining data structures

Notation

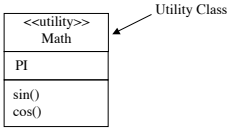
- A type is shown with the stereotype «type»
- An implementation class is shown with the stereotype «implementationClass»
- The implementation of a type is modeled with the *realization relationship*



Source: OMG - UML Specification

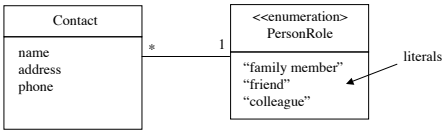
Utility Classes

- A utility is a grouping of global variables and procedures in the form of a class
  - utilities are used for programming convenience
- A utility class is a class with only class-scope attributes and operations
- The stereotype <<utility>> is used to indicate that a class is a utility



Enumeration

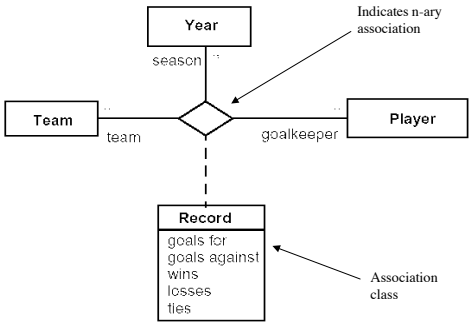
- An Enumeration is a user-defined data type
- The instances of an enumeration are literals (specified by the user)
- The stereotype <enumeration>> is used to indicate that a class is an enumeration



N-ary Association

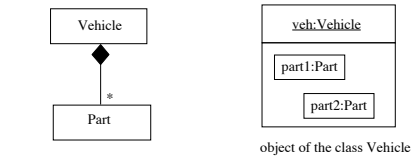
- An n-ary association is an association among three or more classes
  - N equals to the number of classes in the association
- Each instance of the association is an n-tuple of values from the respective classes
- A binary association (seen in the previous lectures) is a special case of the n-ary association
  - the multiplicities of n-ary association are less obvious than multiplicities of binary associations
- It is usually best to use binary associations to keep a model clearer

Example



Composite Objects

- A composite object is an object which is composed of other objects called parts
  - useful for representing real-time systems
- A composite object is an instance of a composite class
  - which implies the composition aggregation between the class and its parts



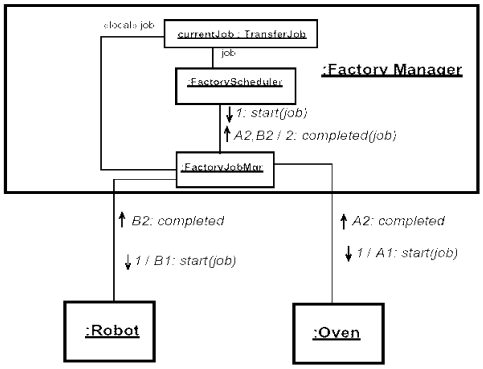
Class diagram with composition

object of the class Vehicle

Active object

- An *active object* is one that owns a thread of control
  - e.g. processes and threads
- A passive object holds data but does not initiate control
  - e.g. Account, Student, etc…
- An active object is shown as a rectangle with a heavy border
- It is very common that an active object is a composite object (contains other objects)

Example

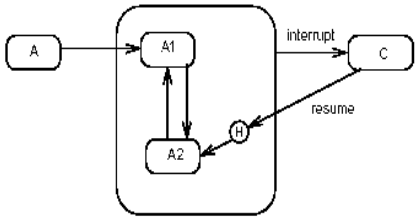


Source: OMG – UML Specification

State Diagrams - History Pseudostates

- History pseudostates are used to make the object return to the previously visited hierarchical state instead of starting for the initial state
- There are two different history pseudostates:
  - Shallow history means that history applies to the current nesting substate only
    - states nested more deeply are not affected by the presence of history pseudostates
    - Shallow history is indicated with an icon labeled H
  - Deep history applies to all levels of nesting, and is indicated with H*

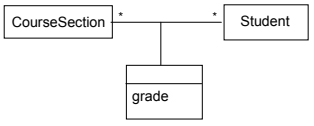
Example



Source: OMG – UML specification

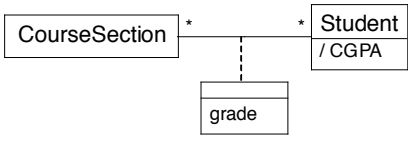
Optimization techniques (caching)

- Storing additional data to make calculations faster
- Also called
  - redundant attributes and
  - redundant associations
- Example:
  - Whenever the students CGPA must be reported, we have to iterate through all course sections the student has taken
  - This is wasteful if queries are more common than updates



Redundant attributes

- After adding a redundant attribute
- The added element is called in UML: **derived element**
  - the name of the derived element is preceded by /

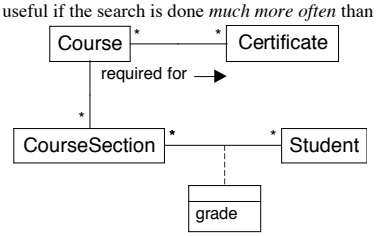


Redundant attributes (cont.)

- Any method that logically affects the validity of the redundant data must invalidate the cache
- Example: when a grade is changed, CGPA is set to NULL
- Any method that accesses the redundant data must:
  - if the cache is invalid, calculate the redundant value and store it
    - *e.g. calculate the CGPA*
  - Always return the redundant value, that is now guaranteed valid

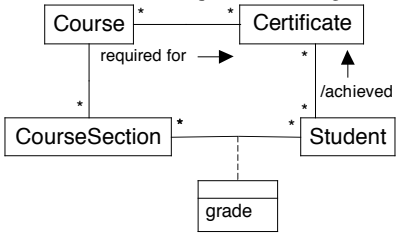
Redundant associations

- We can **calculate** how many students have earned a certain certificate by
  - traversing the students and check all their courses;
  - comparing these to the requirements for certificates
- However we can maintain an extra association to save the search:
  - useful if the search is done *much more often* than updates



Redundant associations (cont.)

- Add a many-to-many association from Certificate to Student
- Every time a student passes a course, the appropriate method would have to update any certificates for which that course was required
- But watch out if certificate requirements change!



Points about caching and redundancy

- They add complexity and reduce maintainability
- They should only be used when there are performance problems or obvious wastes of resources
- Do not add these features until you know you are wasting significant unnecessary resources

Types of caching

- Immediate redundancy (eager evaluation)
  - update the redundant value when the data it depends on is updated
  - don't wait for something to query the redundant value
  - maximizes query speed; minimizes update speed
    - e.g. update the CGPA whenever any grade is changed
- Background redundancy
  - update redundant value when CPU is idle
  - some redundant values will not need to wait for a query before they are updated
- Unlimited caching
  - as described earlier
  - store all calculation results whenever a query is made

Types of caching (cont.)

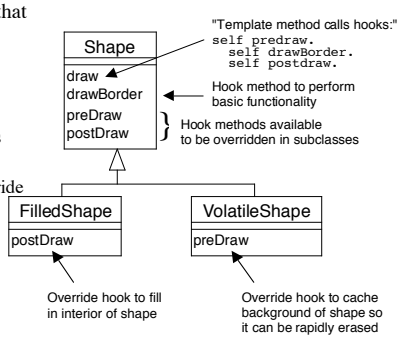
- Limited caching
  - don't keep all redundant values
  - used when redundant values consume much memory
  - keep most recently used or most commonly used
- Calculation on demand
  - no redundancy at all
  - simplest, easiest to understand design
- Deferred calculation (lazy evaluation)
  - don't calculate until you know the results of the calculation are needed
  - when a query is made for a value, return *instructions for how to compute the value*, not the value itself
  - useful if calculating the value is very expensive, yet you are not really sure if you will make use of the queried value

Fine tuning the design

- Several techniques that improve Understandability and Maintainability
- Add simple methods that call methods with more arguments
  - Sometimes it is useful to have several operations that differ only by the number of arguments
  - Defaults are used for missing arguments

Factoring out potential *hooks*

- Add extra methods that implement parts of operations where flexibility is needed
  - these are called by 'template' methods
  - subclasses can implement or override hooks



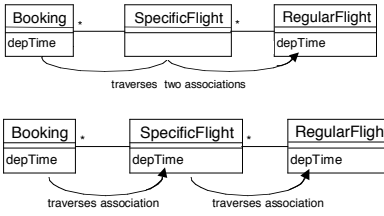
Improve the abstractions of classes

- Creating an abstract superclass that contains some of the features of its subclasses
- Reasons for abstracting out:
  - To make the subclasses simpler
    - *can even be done when there is just one subclass*
  - To make the superclass reusable
    - *identify all possible candidates for reusability*
  - To make the system extensible or more easily configurable
    - *with an abstract superclass we can add new specialized subclasses whenever the need arises - these will just have a few methods to do specialized functions*

Improving delegation

- Delegation occurs when a method does nothing except *forward* a message to another object
  - i.e. a object delegates responsibility to another
- Reasons for using delegation:

A.To ensure a class only talks to its neighbours



Improving delegation (cont.)

B. To avoid duplicating functionality in more than one class
- e.g. it would have been bad to have separate instance variables and calculations of 'dep time' in Specific Flight *and* Regular Flight

C. To reuse code without abusing inheritance
- e.g. stack would *contain* a list rather than being a subclass of list
- Stack would have method that delegates 'size' etc. to the contained list

Reviewing the design

- Ensure as many methods as possible are private
  - *improves information hiding*
- Ensure instance variables are accessed by methods rather than directly
  - *unless this causes an efficiency problem*
- Ensure that the only associations traversed by a method are ones connected to *its* class
  - *only one method should traverse an association (others should call it)*

Reviewing the design (cont.)

- Ensure interfaces are free of implementation details
- Make sure each method is cohesive and as short as possible
  - *split a method into submethods if possible*
- Consider breaking up classes using inheritance or player-role framework if it contains (as guidelines):
  - *more than 8 attributes*
  - *more than 5 associations*
  - *more than 40 operations*

Umple

A programming language family developed in my research lab

Adds associations and attributes to programming languages
- Java
- PHP

Works with Rational Software Modeller and some other tools for diagram generation and code generation

Stand-along code-generator is online at
-

Declaration of classes and attributes

```
class Student
{
  studentNumber; // defaults to String
  String grade;
  Integer entryAverage; // implemented as int
}
```


Associations

```
class Student { id; name; }

class Course { description; code; }

class CourseSection {
  sectionLetter;
  1..* -- 1 Course;
}

association {
   * CourseSection;
   * Student registrant;
}
```


Selected patterns

```
class University {
  singleton;
  String name;
}
```

SEG4210 – Advanced Software Design and Reengineering

TOPIC 4
Metamodelling

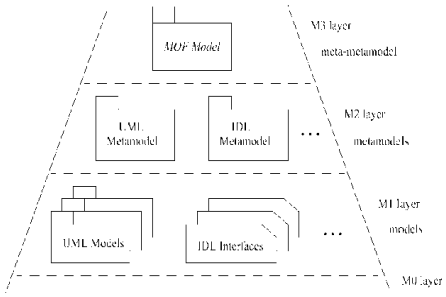
What is a Meta-metamodel?

- A metamodel describes information about models
- A meta-metamodel describes information about metamodels
- Metamodels that are defined using the same meta-metamodel
  - Can exchange information
  - Can be used by the same CASE tools that understand the meta-metamodel

What is MOF?

- MOF stands for Meta Object Facility
  - enables meta-metamodeling of UML level metamodels
- It defines a small set of concepts (such as package, class, method, attribute…) that
  - allow one to define and manipulate **models** of metadata (data about data)
  - are described using a subset UML notation

OMG 4-Layer Architecture



OMG 4-Layer Architecture (Cont.)

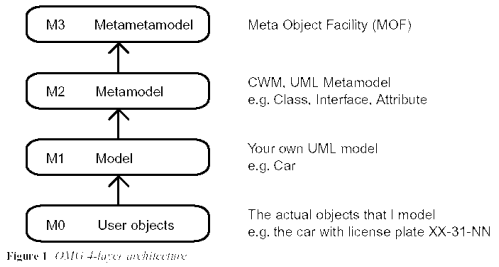
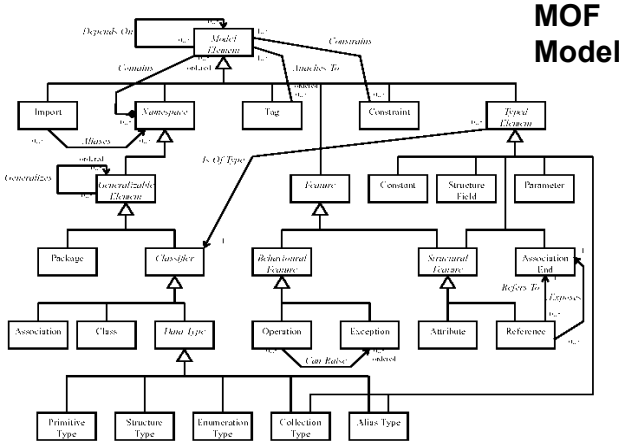
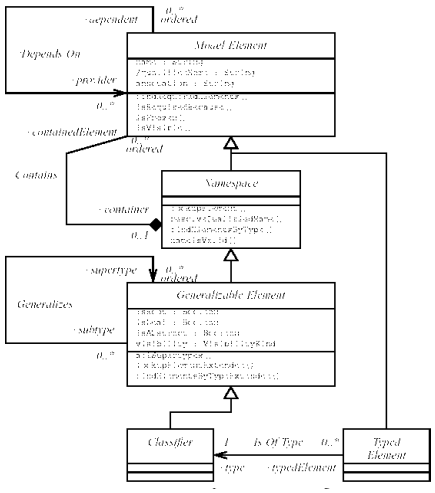


Figure 1 *OMG 4-layer architecture*

MOF Model



MOF Key Abstract Classes



MOF Key Abstract Classes (Cont.)

- **ModelElement** common base Class of all M3-level Classes. Every ModelElement has a name
- **Namespace** base Class for all M3-level Classes that need to act as containers
- **GeneralizableElement** base Class for all M3-level Classes that support generalization (i.e. inheritance)
- **TypedElement** base Class for M3-level Classes such as Attribute, Parameter, and Constant whose definition requires a type specification
- **Classifier** base Class for all M3-level Classes that (notionally) define types. Examples of Classifier include Class and DataType

The MOF Model: Main Concrete Classes

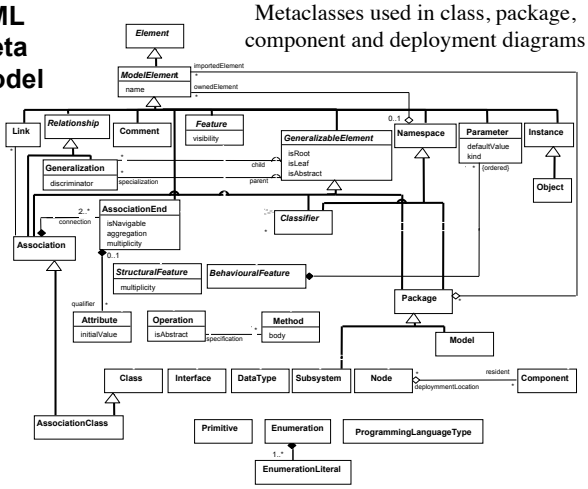
- The key concrete classes (or meta-metaclasses) of MOF are as follows:
  - Class
  - Association
  - Exception (for defining abnormal behaviours)
  - Attribute
  - Constant
  - Constraint

The MOF Model: Key associations

- Contains: relates a ModelElement to the Namespace that contains it
- Generalizes: relates a GeneralizableElement to its ancestors (superclass and subclass)
- IsOfType: relates a TypedElement to the Classifier that defines its type
  - An object is an instance of a class
- DependsOn : relates a ModelElement to others that its definition depends on
  - E.g. a package depends on another package

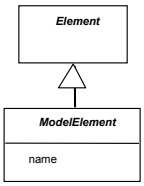
UML Meta Model

Metaclasses used in class, package, component and deployment diagrams



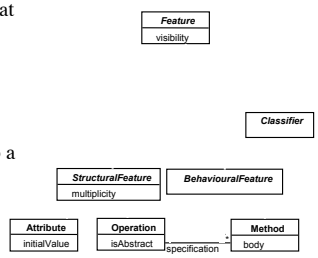
Model Elements

- An element is an atomic constituent of a model.
- Element is the top metaclass in the metaclass hierarchy
- A model element is a named entity in a Model
- It is the base for all modeling metaclasses in UML
  - All other modeling metaclasses are either direct or indirect subclasses of ModelElement



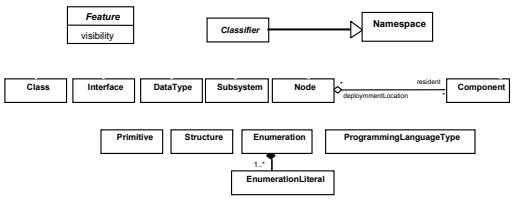
Features

- Feature is an abstract class that declares a behavioral or structural property of
  - an instance of a Classifier
  - the Classifier itself
- A behavioral feature refers to a dynamic feature of a model element
  - E.g. operation or method
- A structural feature refers to a static feature of a model element
  - E.g. attribute



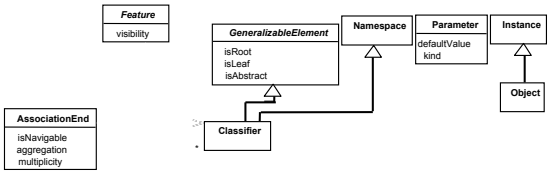
Classifier

- A classifier is an element that describes behavioral and structural features
  - E.g. class, data type, interface, component
- Classifier is an abstract class that
  - declares a collection of Features, such as Attributes, Methods…
  - has a name, which is unique in the Namespace enclosing the Classifier



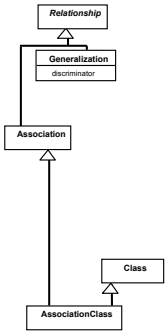
Classifier (cont.)

- A classifier defines a namespace and is a generalizable element
- Can have
  - association ends
  - parameters
  - instances



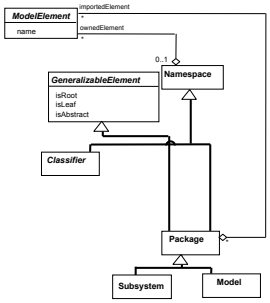
Relationships

- A relationship is a connection among model elements
- UML defines several relationships such as:
  - Association
  - Generalization
- UML defines other types of relationships that are not shown in this diagram, such as:
  - Dependency
  - Flow



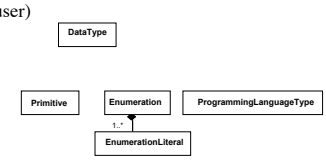
Namespace

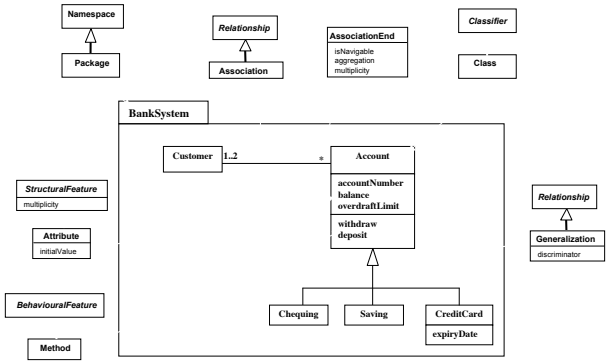
- A namespace is a part of a model that contains a set of other model elements
  - E.g. Associations and Classifiers
  - the name of an owned model element is unique within the namespace
- Namespace is an abstract metaclass and it subclasses are
  - Classifier
  - Package



Data Types

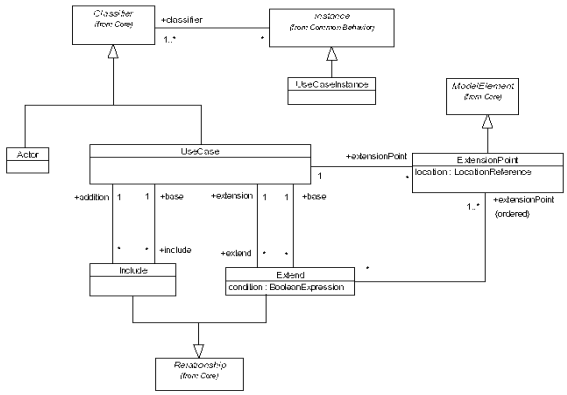
- UML Data types include
  - primitive built-in types (such as integer and string)
  - definable enumeration types (such as Boolean whose literals are false and true)
- Programming languages data types
  - are specified according to the semantics of a particular programming language
  - are not portable among languages (except by agreement among the languages)
  - do not map into other UML classifiers
- Enumerations are a user-defined data types whose instances are literals (specified by the user)



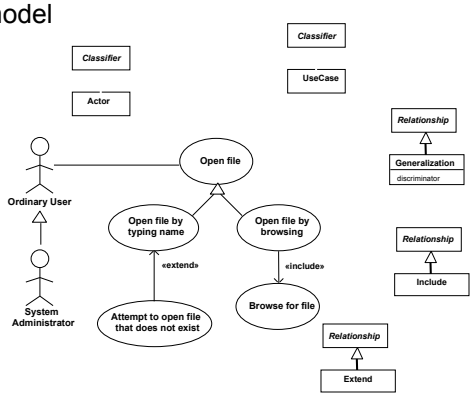


Mapping of UML Models to Metamodel Elements (Example)

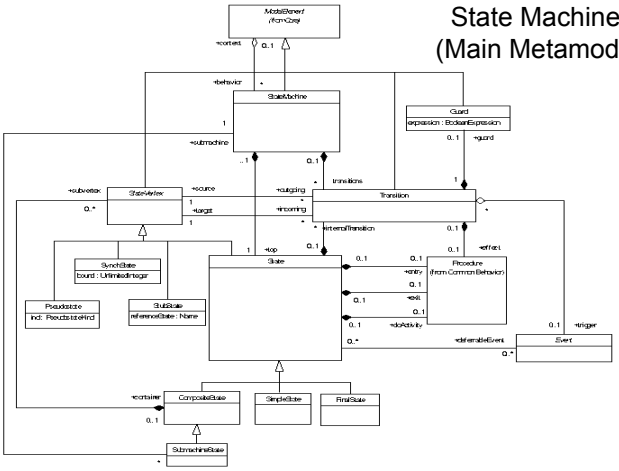
Use Case Diagrams Metamodel



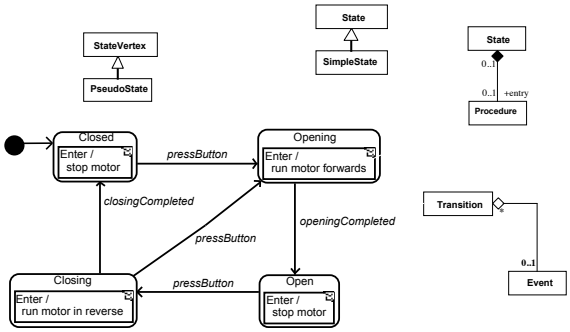
Mapping Use Cases Model to Metamodel



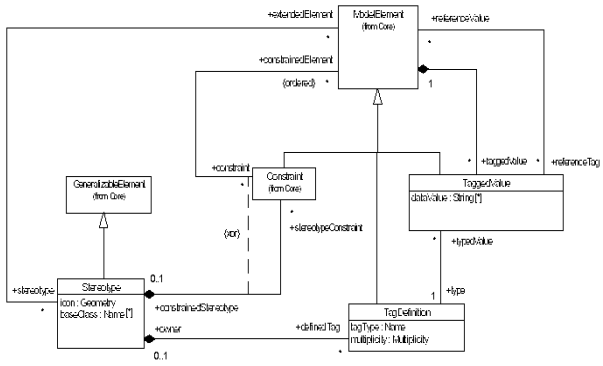
State Machines (Main Metamodel)



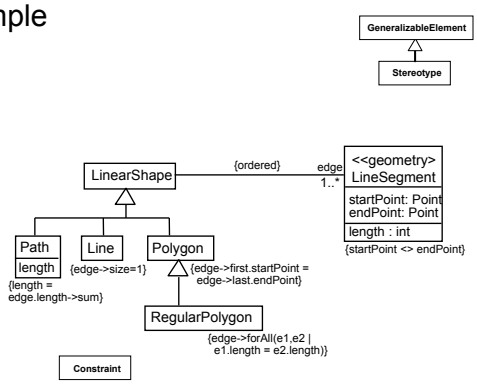
Mapping State Machines to Metamodel



Extension Mechanisms Metamodel



Example


GeneralizableElement
Stereotype
LinearShape
{ordered}
edge
1..*
<<geometry>>
LineSegment
startPoint: Point
endPoint: Point
length : int
{startPoint <> endPoint}
Path
length
{length =
edge.length->sum}
Line
{edge->size=1}
Polygon
{edge->first.startPoint =
edge->last.endPoint}
RegularPolygon
{edge->forAll(e1,e2 |
e1.length = e2.length)}
Constraint

SEG4210 – Advanced Software Engineering and Reengineering

TOPIC 5
Object Constraint Language (OCL)

What is OCL?

OCL is a *formal language* used to express *constraints*.

Constraint:

- An invariant condition that must hold for the system being modeled.

Constraints do not have *side effects*

- Their evaluation cannot alter the state of the executing system.
- No expression you can write in OCL allows side effects
  - —E.g. there is no way to assign a value to an attribute
  - —Expressions can only return a value
- OCL cannot be used as a programming language

More on constraints

Constraints are applied to instances of objects of the UML metamodel

- i.e. to model objects that are instances of Class, Attribute, AssociationEnd, Association, etc. etc.

Without OCL, constraints would have to be expressed in natural language

- This almost always results in ambiguities

The evaluation of an OCL constraint is 'instantaneous'.

- the states of objects in a model cannot change during evaluation.

Formal specification languages in general

Various other formal languages have also been used in software engineering

- e.g. Z (pronounced 'zed')

Most such languages have proven difficult for the average developer to use

- The mathematical symbols are not well known and use obscure fonts

Formal specification languages, continued

Formal specification languages are for modelling not programming

- Some things in OCL are not necessarily executable.
  - —E.g. there is an 'allInstances' operation
- Implementation issues (e.g. the data structures to be used to implement an association) cannot be referenced

As with other formal specification languages, most of OCL represents mathematical concepts from

- Logic
- Set theory

OCL started as a business modeling language within the IBM insurance division

Places to use OCL in UML models

- To specify invariants on classes and types in the class model
- To specify type invariant for stereotypes
- To describe pre- and post- conditions on operations and methods
- To describe guards
- As a navigation language
- To specify constraints on operations

Comments in OCL

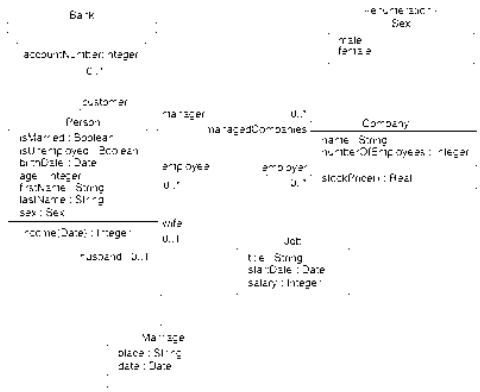
Comments follow two dashes

-- this is a comment

Most of the examples on subsequent slides come from the UML 1.5 spec:

http://www.omg.org/technology/documents/formal/uml.htm

A class diagram for discussion (from the OCL specification



OCL is a strongly *Typed* Language

To be well formed, an OCL expression must conform to type conformance rules

- E.g., you cannot compare an Integer with a String.

Each Classifier within a UML model becomes an OCL type

- E.g. each class you create

OCL includes a set of supplementary predefined types

Basic types in OCL

Boolean -> true, false

- Ops: and, or, xor, not, implies, if-then-else

Integer -> 1, -5, 2, 34, 26524, ...

- Ops: *, +, -, /, abs()

Real -> 1.5, 3.14, ...

- Ops: *, +, -, /, floor()

String -> 'To be or not to be...'

- Ops: toUpper(), concat()

Use of enumeration types

context Person inv:
sex = Sex::male

OCL expressions can refer to model elements from the UML metamodel such as

Properties

- Attributes
- AssociationEnds
- Methods where isQuery is true

Examples

context AType inv:
self.property …

context Person inv:
self.age > 0

Collection types and navigation in OCL expressions

If self is class C, with attribute a
- Then **self.a** evaluates to the **object** stored in a.

If C has a one-to-many association called assoc to another class D
- Then **self.assoc** evaluates to a **Set** whose elements are of type D.
- If assoc is {ordered} then a **Sequence** results

If D has attribute b
- Then the expression **self.assoc.b** evaluates to the set of all the b's belonging to D

Example expressions

```
context Company
  inv: self.manager.isUnemployed = false
    -- self.manager evaluates to a Person
  inv: self.employee->notEmpty()
     -- self.employee evaluates to a Set
```

Note: **->notEmpty()** is a call to an OCL built in Collection function
- Discussed a bit later

Some OCL functions defined on all objects

oclIsTypeOf(t : OclType) : Boolean
- true if the type of self and t are the same.
- E.g.

```
context Person
  inv: self.oclIsTypeOf( Person ) -- is true
  inv: self.oclIsTypeOf( Company) -- is false
```

oclIsKindOf(t : OclType) : Boolean
- true if t is either the direct type or one of the supertypes of an object.

OCL built-in Collection functions

You can define Collections by navigation or directly:
- Set { 1 , 2 , 5 , 88 }
- Set { 'apple' , 'orange', 'strawberry' }
- Sequence { 1, 3, 45, 2, 3 }
- Sequence { 'ape', 'nut' }

The notation *->function()* is used to call a built in function on an OCL Collection
- Do not confuse this with the dot '.' which is used to
  - — access properties and navigate
  - — including calling query functions defined in the model

Some important OCL built in Collection functions

```
aCollection->isEmpty(), ->notEmpty
aCollection->size()
aCollection->includes(anObject)
aCollectionOfNumbers->sum()

aCollection->exists(booleanExpression)
```
- Returns true if booleanExpression is true for any element of aCollection
- This is the equivalent of the ∃ symbol in mathematical logic

Select and reject: Picking subcollections

```
context Company inv:
  self.employee->select(age > 50)->notEmpty()
  -- select(age > 50) picks the people over 50

context Company inv:
  self.employee->select(p | p.age > 50)->notEmpty()
```

->reject() just picks the opposite subset

Collect: generating parallel collections

self.employee->collect(birthdate)

Creates a collection of equal size as the set of employees, with the employee dates

- The result will be a Bag
- A Bag can contain duplicates!

forAll: Evaluating some expression on every element of a collection (∀ in logic)

collection->forAll(v : Type | bool-expr-with-v)
collection->forAll(v | boolean-expression-with-v)
collection->forAll(boolean-expression)

The following are true if everybody in the company is called Jack

inv: self.employee->forAll(forename = 'Jack')
inv: self.employee->forAll(p | p.forename = 'Jack')
inv: self.employee->forAll(p : Person | p.forename = 'Jack')

Some more collection functions

c1->includesAll(c2)
- True if every element of c2 is found in c1

c1->excludesAll(c2)
- True if no element of c2 is found in c1

For sets:
s1->intersection(s2)
- The set of those elements found s1 and also in s2

s1->union(s2)
- The set of those elements found in either s1 or s2

s1->excluding(x)
- The set s1 with object x omitted.

For sequences
seq->first()

Logical implication

```
context Person inv:
  (self.wife->notEmpty() implies
    self.wife.age >= 18)
      and
  (self.husband->notEmpty() implies
    self.husband.age >= 18)
```


forAll with two variables

Considers each pair in the *Cartesian product* of employees

```
context Company inv:
  self.employee->forAll( e1, e2 : Person |
  e1 <> e2 implies e1.forename <> e2.forename)
```

This is the same as

```
self.employee->forAll(e1 | self.employee->forAll (e2 |
   e1 <> e2 implies e1.forename <> e2.forename)))
```


Let expressions

```
context Person inv:
  let income : Integer = self.job.salary->sum()
  let hasTitle(t : String) : Boolean =
       self.job->exists(title = t) in
  if isUnemployed then
    self.income < 100
  else
    self.income >= 100 and self.hasTitle('manager')
  endif
```

Let expressions that define values for use in other expressions

These use the 'def' keyword

```
context Person def:
  let income : Integer = self.job.salary->sum()
  let hasTitle(t : String) : Boolean =
    self.job->exists(title = t)
```


An example class invariant

The following all say the same thing

- numberOfEmployees > 50
- self.numberOfEmployees > 50
- context Company inv:
  self.numberOfEmployees > 50
- context c : Company inv:
  c.numberOfEmployees > 50
- context c : Company inv enoughEmployees:
  c.numberOfEmployees > 50

Example preconditions and postconditions

```
context Person::income(d : Date) : Integer
  post: result = 5000

context Typename::operationName(
          param1 : Integer ): Integer
  pre parameterOk: param1 > 5
  post resultOk: result < 0
```


Use of @pre in postconditions

You often want to write a postcondition that states what has changed with respect to a precondition

- Use **property@pre** to refer to the value of **property** prior to execution

e.g.

```
context Company::hireEmployee(p : Person)
  pre : not employee->includes(p)
  post: employee->includes(p) and
  stockprice() = stockprice@pre() + 10
```


Precedence Rules for OCL expressions

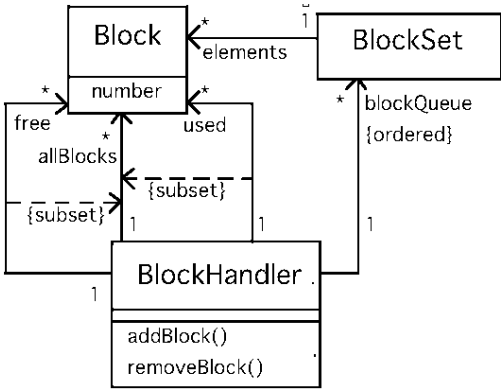
Precedence order for operations, starting with highest:

- @pre
- dot and arrow operations: '.' and '->'
- unary 'not' and unary minus '-'
- '*' and '/'
- '+' and binary '-'
- 'if-then-else-endif'
- '<', '>', '<=', '>='
- '=', '<>'
- 'and', 'or' and 'xor'
- 'implies'

Parentheses '(' and ')' can be used to change precedence

- Often useful simply too make it clearer

Examples based on a block handler



Copyright acknowledgement

The preceding diagram and the examples that follow were created by Prof. Lethbridge and appear in the 6th edition of Roger Pressman's "Software Engineering: A Practitioner's Approach" to bbe published soon

Examples OCL expressions in the BlockHandler problem

No block will be marked as both unused and used

```
context BlockHandler inv:
  (self.used->intersection(self.free)) ->isEmpty()
```

All the *sets* of blocks held in the queue will be subsets of the collection of currently used blocks

```
context BlockHandler inv:
  blockQueue->forAll(aBlockSet |
     used->includesAll(aBlockSet.elements ))
```


Examples … 2

No elements of the queue will contain the same block numbers.

```
context BlockHandler inv:
  blockQueue->forAll(blockSet1, blockSet2 |
    blockSet1 <> blockSet2 implies
    blockSet1.elements.number
      ->excludesAll(blockSet2.elements.number))
```


Examples … 3

The collection of used blocks and blocks that are unused will be the total collection of blocks that make up files.

```
context BlockHandler inv:
  allBlocks = used->union(free)
```

The collection of unused blocks will have no duplicate block numbers.

```
context BlockHandler inv:
  free->isUnique(aBlock | aBlock.number)
```

The collection of used blocks will have no duplicate block numbers.

```
context BlockHandler inv:
  used->isUnique(aBlock | aBlock.number)
```


Examples … 4

```
context BlockHandler::removeBlocks()
  pre: blockQueue->size() >0
  post:

    and
   free = free@pre->
    union((blockQueue@pre->first().elements)
    and
```


If time permits …

We will do examples on board based on Traffic Light system

SEG4210 – Advanced Software Design and Reengineering

TOPIC 6
Aspect Oriented Programming and Aspect/J

What is Aspect-Oriented Programming?

A way of dividing up a program

- Sometimes there are *aspects* of a program that you need to implement in many different classes and methods, e.g.
  - Logging
  - Security
  - Persistence
  - Caching and redundancy
- Sometimes you want to design two features separately even though they are going to be coded in the same classes

These "separate concerns" are implemented as *aspects*

Aspects, continued

An aspect is said to *cross-cut* several classes or methods

We code each aspect separately without making the code for several classes of methods more complex

- Each chunk of code we write ends up being much simpler

Before running the compiler an *aspect weaver* merges the aspects together to create the final system

Before designing aspects, we have to understand the *joint points*

- Places in classes or methods where code for different aspects can be automatically executed

Aspect-Oriented Programming Languages

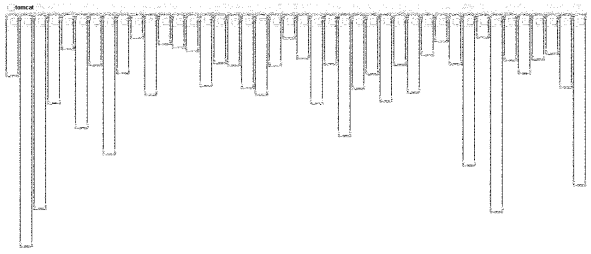
Aspect/J

- Adds AOP to Java
- The aspect weaver modifies the byte code
- Developed by Xerox/PARC
  - —Some of the following slides come from there
- Aspectj.org
- Strong support within Eclipse

Aspect/C++

HyperJ

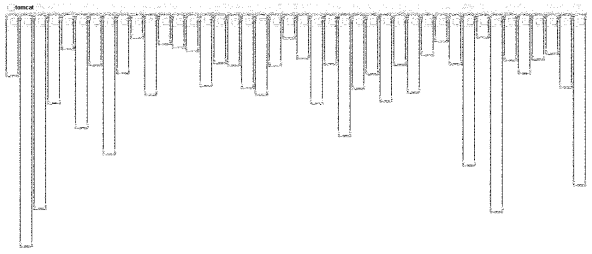
Good modularity



URL pattern matching in org.apache.tomcat

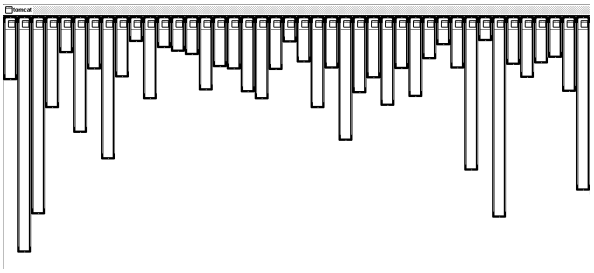
- red shows relevant lines of code
- nicely fits in two boxes (using inheritance)

Even better modularity



XML parsing in Apache

Modularity problem



Logging in Apache

- Not even in a small number of places

Another modularity problem - session expiration in Apache

Problems with lack of modularity in cross-cutting concerns

Redundant code

- Many little bits that do the same thing all over the place

Difficulty understanding code

- What do all those little bits do?

Difficult to change code

- You have to change many places consistently
- Easy to break code (introduce bugs)
  - —By forgetting to change something
  - —By making changes inconsistently

Concepts in an Aspect/J aspect

Aspect

- A new modular unit of code that imlements the design-time aspect

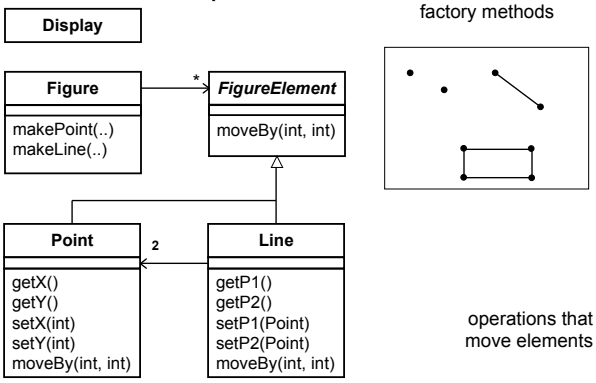
Pointcuts

- These specify the join points

Advice

- Instructions about *what to do* at join points picked out by a pointcut

Figure Editor example



Challenge: Update the display whenever any FigureElement changes

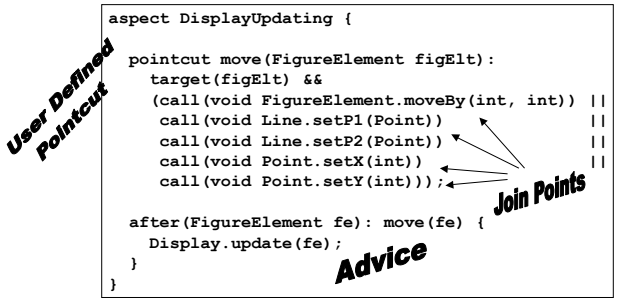
Traditional approach

- Inform the display subsystem to update the screen
- By inserting a call to display after *every possible line of code* that changes the way a shape looks

Aspect-Oriented approach

- Create an aspect that automatically performs the updating in the right places
- Every time you compile the code, you run a the aspect weaver
  - —'edits' the code to add the aspect code in the right places

An Aspect in aspectJ




```
aspect DisplayUpdating {

  pointcut move(FigureElement figElt):
    target(figElt) &&
    (call(void FigureElement.moveBy(int, int)) ||
     call(void Line.setP1(Point))              ||
     call(void Line.setP2(Point))              ||
     call(void Point.setX(int))                ||
     call(void Point.setY(int)));

  after(FigureElement fe): move(fe) {
    Display.update(fe);
  }
}
```


Some Aspect/J Join Point Designators for writing Pointcuts - 1

when a particular method body executes (i.e. a call is made to the code)
- `execution(void Point.setX(int))`
- Selects code that is *called*

when the code calls a method
- `call(void Point.setX(int))`
- Selects the *caller*

when an exception handler executes
- `handler(ArrayOutOfBoundsException)`

Some Aspect/J Join Point Designators for writing Pointcuts - 2

when the object currently executing (the caller) is of type SomeType
- `this(SomeType)`

when the target object of a call is of type SomeType
- `target(SomeType)`

when the executing code belongs to class MyClass
- `within(MyClass)`
- The executing code might have been inherited, but *originated* in MyClass

in the control flow of a call to Test's main() method
- `cflow(void Test.main())`
- I.e. somewhere down in the call stack initiated by Test.main()

Use of wildcards to pick larger sets of join points

```
target(Point)
target(graphics.geom.Point)
target(graphics.geom.*)          any type in graphics.geom
target(graphics..*)              any type in any sub-package
                                 of graphics

call(void Point.setX(int))
call(public * Point.*(..))       any public method on Point
call(public * *(..))             any public method on any type

call(void Point.getX())
call(void Point.getY())
call(void Point.get*())
call(void get*())                any getter

call(Point.new(int, int))
call(new(..))                    any constructor
```


aspectJ advice categories

before advice
- Runs before entering the join point

```
        (FigureElement fe) : move(fe)
  { System.out.println("About to move figure " +
    fe); }
```

after advice
- Runs on the way back out

```
after(FigureElement fe): move(fe)  {
  Display.update(fe); }
```

- "after returning" advice (gives access to the return value)
- "after exception" advice (gives access to the exception)

around advice
- Runs *instead* of the join point. The original join point action can be invoked via the *proceed* call.

An aspect for logging

```
public aspect AutoLog{
  pointcut publicMethods() : execution(public * mypackage..*(..));
  pointcut logObjectCalls() : execution(* Logger.*(..));
  pointcut loggableCalls() : publicMethods() && ! logObjectCalls();

  before() : loggableCalls(){
    Logger.entry(thisJoinPoint.getSignature().toString());
  }
  after() : loggableCalls(){
    Logger.exit(thisJoinPoint.getSignature().toString());
  }
}
```

SEG4210 – Advanced Software Design and Reengineering

TOPIC 7

Java Collections Framework

Collections Frameworks

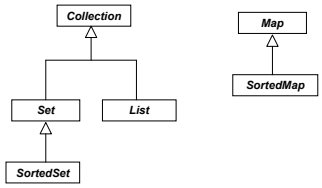
- A *collection* is an object that represents a group of objects
- A collections framework is a unified architecture for representing and manipulating collections
  - the manipulation should be independent of the implementation details
- Advantages of having a collections framework
  - reduces *programming effort* by providing useful data structures and algorithms
  - increases *performance* by providing high-performance data structures and algorithms
  - *interoperability* between the different collections
  - Having a *common language* for manipulating the collections

The components of Java Collections Framework from the user's perspective

- Collection Interfaces: form the basis of the framework
  - such as sets, lists and maps
- Implementations Classes: implementations of the collection interfaces
- Utilities - Utility functions for manipulating collections such as sorting, searching…
- The Java Collections Framework also includes:
  - algorithms
  - *wrapper* implementations
    - add functionality to other implementations such as synchronization

Collections Interfaces

- Java allows using:
  - Lists
  - Sets
  - Hash tables (maps)



Interface: Collection

- The Collection interface is the root of the collection hierarchy
- Some Collection implementations allow
  - duplicate elements and others do not
  - the elements to be ordered or not
- JDK doesn't provide any direct implementations of this interface
  - It provides implementations of more specific sub interfaces like Set and List

```
public interface Collection {
   // Basic Operations
   int size();
   boolean isEmpty();
   boolean contains(Object element);
   boolean add(Object element);
   boolean remove(Object element);
   Iterator iterator();

   // Bulk Operations
   boolean containsAll(Collection c);
   boolean addAll(Collection c);
   boolean removeAll(Collection c);
   boolean retainAll(Collection c);
   void clear();

   // Array Operations
   Object[] toArray();
   Object[] toArray(Object a[]);
}
```


Interface: Set

- A Set is a collection that cannot contain duplicate elements
- A set is not ordered
  - however, some subsets maintain order using extra data structures
- This is similar to the mathematical concept of *sets*

```
public interface Set {
   // Basic Operations
   int size();
   boolean isEmpty();
   boolean contains(Object element);
   boolean add(Object element);
   boolean remove(Object element);
   Iterator iterator();

   // Bulk Operations
   boolean containsAll(Collection c);
   boolean addAll(Collection c);
   boolean removeAll(Collection c);
   boolean retainAll(Collection c);
   void clear();

   // Array Operations
   Object[] toArray();
   Object[] toArray(Object a[]);
}
```

Interface: List

- A List is ordered (also called a *sequence*)
- Lists can contain duplicate elements
- The user can access elements by their integer index (position)

```
public interface List extends Collection {
   // Positional Access
   Object get(int index);
   Object set(int index, Object element);
   void add(int index, Object element);
   Object remove(int index);
   abstract boolean addAll(int index, Collection c);

   // Search
   int indexOf(Object o);
   int lastIndexOf(Object o);

   // Iteration
   ListIterator listIterator();
   ListIterator listIterator(int index);

   // Range-view
   List subList(int from, int to);
}
```


Interface: Map

- A Map is an object that maps keys to values
- Maps cannot contain duplicate keys
- Each key can map to at most one value
- Hashtables are an example of Maps

```
public interface Map {
   // Basic Operations
   Object put(Object key, Object value);
   Object get(Object key);
   Object remove(Object key);
   boolean containsKey(Object key);
   boolean containsValue(Object value);
   int size();
   boolean isEmpty();

   // Bulk Operations
   void putAll(Map t);
   void clear();

   // Collection Views
   public Set keySet();
   public Collection values();
   public Set entrySet();

   // Interface for entrySet elements
   public interface Entry {
      Object getKey();
      Object getValue();
      Object setValue(Object value);
   }
}
```


Interface: SortedSet

- A SortedSet is a Set that maintains its elements in an order
- Several additional operations are provided to take advantage of the ordering

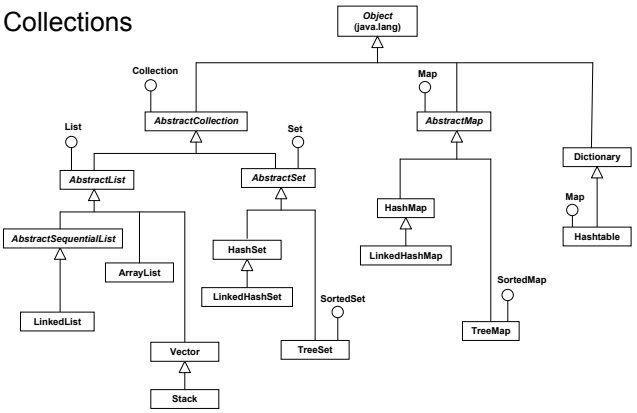
```
public interface SortedSet extends Set {
   // Range-view
   SortedSet subSet(Object fromElement, Object
toElement);
   SortedSet headSet(Object toElement);
   SortedSet tailSet(Object fromElement);

   // Endpoints
   Object first();
   Object last();

   // Comparator access
   Comparator comparator();
}
```


Interface: SortedMap

- A SortedMap is a Map that maintains its mappings in ascending key order
- The SortedMap interface is used for apps like dictionaries and telephone directories

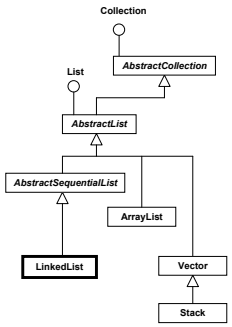
```
public interface SortedMap extends Map {
   Comparator comparator();

   SortedMap subMap(Object fromKey, Object
toKey);
   SortedMap headMap(Object toKey);
   SortedMap tailMap(Object fromKey);

   Object firstKey();
   Object lastKey();
}
```


Java Collections



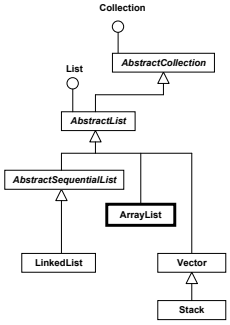
Java Lists: LinkedList

- Java uses a doubly-linked list
  - it can be traversed from the beginning or the end
- LinkedList provides methods to get, remove and insert an element at the beginning and end of the list
  - these operations allow a linked list to be used as a stack or a queue
- LinkedList is not synchronized
  - problems if multiple threads access a list concurrently
  - LinkedList *must* be synchronized externally



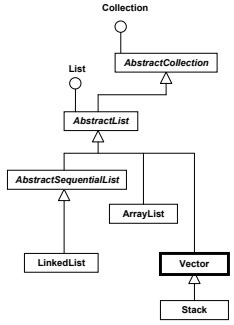
Java Lists: ArrayList

- Uses an array to store the elements
- In addition to the methods of the interface List
  - it provides methods to manipulate the size of the array (e.g. ensureCapacity)
- More efficient than LinkedList for methods involving indices – get(), set()
- It is not synchronized



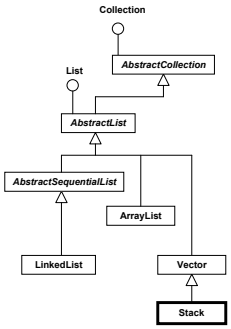
Java Lists: Vector

- Same as an the class ArrayList
- The main difference is that:
  - The methods of Vector are synchronized



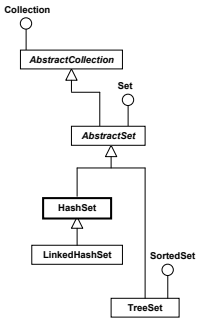
Java Lists: Stack

- The Stack class represents a last-in-first-out (LIFO) stack of objects
- The common push and pop operations are provided
- As well as a method to peek at the top item on the stack is also provided
- Note: It is considered bad design to make Stack a Subclass of vector
  - Vector operations should not be accessible in a Stack
  - It is designed this way because of a historical mistake



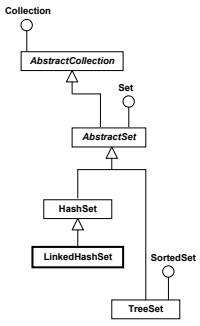
Java Sets: HashSet

- HashSet uses a hash table as the data structure to represent a set
- HashSet is a good choice for representing sets if order of the elements is not important
- HashSet methods are not synchronized



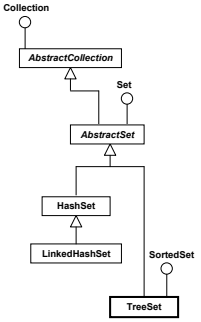
Java Sets: LinkedHashSet

- Hash table and linked list implementation of the Set interface
- LinkedHashSet differs from HashSet in that the order is maintained
- Performance is below that of HashSet, due to the expense of maintaining the linked list
- Its methods are not synchronized



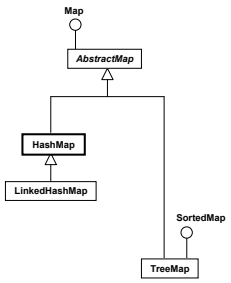
Java Sets: TreeSet

- Stores the elements in a balanced binary tree
  - A binary tree is a tree in which each node has at most two children
- TreeSet elements are sorted
- Less efficient than HashSet in insertion due to the use of a binary tree
- Its methods are not synchronized



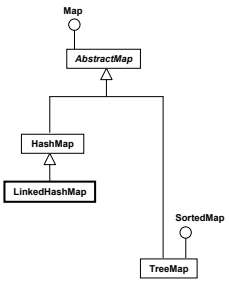
Java Maps: HashMap

- Stores the entries in a hash table
- Efficient in inserting (put()) and retrieving elements (get())
- The order is not maintained
- Unlike HashTable, HashMap's methods are not synchronized



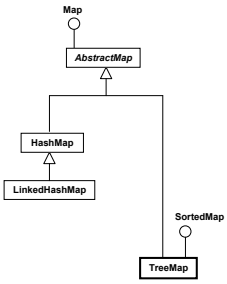
Java Maps: LinkedHashMap

- Hash table and linked list implementation of the Map interface
- LinkedHashMap differs from HashMap in that the order is maintained
- Performance is below that of HashMap, due to the expense of maintaining the linked list
- Its methods are not synchronized



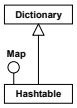
Java Maps: TreeMap

- Red-Black tree based implementation of the SortedMap interface
- The keys are sorted according to their natural order
- Less efficient than HashMap for insertion and mapping
- Its methods are not synchronized



The class Hashtable

- Same as HashMap except that its methods are synchronized
- The class Hashtable was introduced in JDK 1.0 and uses Enumerations instead of Iterators



The Iterator Interface

- JCF provides a uniform way to iterate through the collection elements using the Iterator Interface
- The Iterator interface contains the following methods
  - boolean **hasNext**(): returns true if the iteration has more elements.
  - Object **next**(): returns the next element in the iteration.
  - void **remove**(): removes from the collection the last element returned by the iterator
- Iterator replaces Enumeration in the Java Collections Framework. Iterators differ from enumerations in two ways:
  - They allow the caller to remove elements from the collection during the iteration with well-defined semantics (a major drawback of Enumerations)
  - Method names have been improved

The class Arrays

- This class contains various static methods for manipulating arrays such as:
  - Sorting, comparing collections, binary searching…
- Refer to: for a complete list of utilities

References

The Java Collections Framework:

The Java 1.4.2 APIs:

SEG4210 – Advanced Software Design and Reengineering

TOPIC 8
Java Reflection

Note:

The examples used in these slides are taken from:

Java Tutorial: Reflection API.

They have been enhanced with comments for the purpose of this course

What is Reflection?

- Reflection is a mechanism for manipulating classes, interfaces, and objects at runtime
  - useful for developing tools such as debuggers, class browsers, dynamic analysis tools for Java programs, etc.
- Java Core Reflection is an API with methods to:
  - Determine the class of an object at runtime
  - Get information about a class: modifiers, instance variables (fields), methods, constructors and superclasses
  - Create an instance of a class whose name is not known until runtime
  - Get and set the value of an object's field, even if the field name is unknown until runtime
  - Invoke a method on an object, even if the method is unknown until runtime

Examining Classes

- Java Runtime Environment (JRE) maintains an object of the class java.lang.Class that contains information about a class
  - fields, methods, constructors and interfaces
- Ways to retrieve the Class object:
  - If an instance of the class is available, invoke getClass of the instance:
    - Class c = inst.getClass();
  - If the name of the class is known at compile time, retrieve its Class object by appending *.class* to its name:
    - Class c = myPackage.Helloworld.class;
  - If the class name is unknown at compile time, but available at runtime, the forName method can be used:
    - Class c = Class.forName("myPackage.HelloWorld");

Class Modifiers

- A class declaration includes the following modifiers:
  - public, abstract, or final.
- The method getModifiers of the class Class returns a flag (as an integer) that corresponds to the class modifier
- To interpret these flags, we need to use the static methods of the Modifier class
- The following example displays the modifier of the String class

Example 1: Retrieving class modifiers

```
import java.lang.reflect.*; //contains Java Reflection Model API
import java.awt.*;

class SampleModifier {
  public static void main(String[] args) {
      String s = new String();
      printModifiers(s);
  }

  public static void printModifiers(Object o)  {
      Class c = o.getClass(); //returns the Class object of o
      int m = c.getModifiers(); //return the class modifiers
      if (Modifier.isPublic(m)) // checks if is public
         System.out.println("public");
      if (Modifier.isAbstract(m)) //checks if it is abstract
         System.out.println("abstract");
      if (Modifier.isFinal(m)) //checks if it is final
         System.out.println("final");
  }
}
```

Finding Superclasses

- The method getSuperclass returns an instance of Class that represents the superclass of the Class object that uses it
- To identify all ancestors of a class, we need to call getSuperclass iteratively until it returns null
- The program that follows finds the names of the Button class's hierarchy of superclasses

Example 2: Finding Superclasses

```
import java.lang.reflect.*;
import java.awt.*;

class SampleSuper {

 public static void main(String[] args) {
   Button b = new Button();
   printSuperclasses(b);
 }

 static void printSuperclasses(Object o) {
   Class subclass = o.getClass(); //return the Class object of o
   Class superclass = subclass.getSuperclass(); //return the superclass

   while (superclass != null) {
    String className = superclass.getName(); //getName returns the class
                                             // full name. e.g. java.awt.Button
      System.out.println(className);
      subclass = superclass;
      superclass = subclass.getSuperclass(); // return superclass
  }
 }
}
```


Identifying the Interfaces Implemented by a Class

- The method getInterfaces returns an array of objects of Class that represent the interfaces implemented by the Class object that uses it
- An interface is considered as a class
- The method isInterface (of Class) returns true if an instance of Class is an interface
- The program that follows displays the interfaces implemented by the RandomAccessFile class

Example 3: Identifying Interfaces

```
import java.lang.reflect.*;
import java.io.*;

class SampleInterface {

  public static void main(String[] args) {
     try {
        RandomAccessFile r = new RandomAccessFile("myfile", "r");
        printInterfaceNames(r);
      } catch (IOException e) {
          System.out.println(e);
      }
  }

   static void printInterfaceNames(Object o) {
    Class c = o.getClass(); // returns the Class object that corresponds to o

    // the next statement returns the interfaces implemented by c which
    // represents the class o

    Class[] theInterfaces = c.getInterfaces();

    for (int i = 0; i < theInterfaces.length; i++) {
         String interfaceName = theInterfaces[i].getName(); // return the name
                                                            // of the interface
         System.out.println(interfaceName);
     }
   }
}
```


Identifying Class Fields

- The method getFields returns an array of Field objects containing the class's accessible public fields
- A public field is accessible if it is a member of either:
  - this class
  - a superclass of this class
  - an interface implemented by this class
  - an interface extended from an interface implemented by this class
- The getDeclaredFields method can be used to return private, protected and package-scope fields. It does not include inherited fields
- The methods provided by the Field class allow retrieving the field's name, set of modifiers, etc.

Example 4: Identifying Class Fields

```
import java.lang.reflect.*;
import java.awt.*;

class SampleField {
   public static void main(String[] args) {
      GridBagConstraints g = new GridBagConstraints();
      printFieldNames(g);
   }
   static void printFieldNames(Object o) {
      Class c = o.getClass();
      Field[] publicFields = c.getFields(); //returns all the accessible
                                            //public fields of c
      for (int i = 0; i < publicFields.length; i++) {
         String fieldName = publicFields[i].getName(); //gets the field's name
         Class typeClass = publicFields[i].getType(); //gets the field's type
         String fieldType = typeClass.getName(); //gets the type's name
         System.out.println("Name: " + fieldName +
           ", Type: " + fieldType);
        }
      }
}
```

Class Constructors

- The method getConstructors returns an array of Constructor objects that represents all public constructors of the Class object that uses it
- The method getDeclaredConstructors can be used to retrieve all other constructors of the class
- The methods of the Constructor class can be used to determine the constructor's name, set of modifiers, parameter types…
- The class Constructor can also be used to create a new Instance dynamically using the static method: Constructor.newInstance
- The program that follows displays the parameter types for each constructor in the Rectangle class

Example 5: Class Constructors

```
import java.lang.reflect.*;
import java.awt.*;

class SampleConstructor {

   public static void main(String[] args) {
      Rectangle r = new Rectangle();
      showConstructors(r);
   }

   static void showConstructors(Object o) {
      Class c = o.getClass();
      Constructor[] theConstructors = c.getConstructors(); //get all public
                                                           //constructors
      for (int i = 0; i < theConstructors.length; i++) {
         System.out.print("( ");
         Class[] parameterTypes =
                theConstructors[i].getParameterTypes(); //get the constructor
                                                        //parameters types
         for (int k = 0; k < parameterTypes.length; k ++) {
            //get the name of each parameter
            String parameterString = parameterTypes[k].getName();
            System.out.print(parameterString + " ");
         }
         System.out.println(")");
      }
   }
}
```


Class Methods

- The method getMethods returns an array that contains objects of the Method class that represent the accessible public methods of the Class object that uses it
  - this includes the inherited methods
- To retrieve private, protected and package-scope methods getDeclaredMethods can be used. This method does not include the inherited methods
- The methods of the Method class can be used to return a method's name, return type, parameter types, set of modifiers, etc
- The static method Method.invoke can be used to call the method itself
- The following program displays the name, return type, and parameter types of every public method in the Polygon class

Example 6: Methods of a class

```
import java.lang.reflect.*;
import java.awt.*;
class SampleMethod {
   public static void main(String[] args) {
      Polygon p = new Polygon();
      showMethods(p);
   }
   static void showMethods(Object o) {
      Class c = o.getClass();
      Method[] theMethods = c.getMethods(); //get the class public methods
      for (int i = 0; i < theMethods.length; i++) {
         String methodString = theMethods[i].getName(); //get the method name
         System.out.println("Name: " + methodString);
         //get the method return type
         string returnString = theMethods[i].getReturnType().getName();
         System.out.println("   Return Type: " + returnString);
         //get the method parameters types
         Class[] parameterTypes = theMethods[i].getParameterTypes();
         System.out.print("   Parameter Types:");
         for (int k = 0; k < parameterTypes.length; k ++) {
            //get the name of each parameter
            String parameterString = parameterTypes[k].getName();
            System.out.print(" " + parameterString);
         }
      }
   }
}
```


Useful Methods of java.lang.Class - 1

public static Class forName(String className)
returns a Class object that represents the class with the given name

public String getName()
returns the full name of the Class object, such as "java.lang.String".

public int getModifiers()
returns an integer that describes the class modifier: public, final or abstract

public Object newInstance()
creates an instance of this class at runtime

Useful Methods of java.lang.Class - 2

public Class[] getClasses()
returns an array of all inner classes of this class

public Constructor getConstructor(Class[] params)
returns all public constructors of this class whose formal parameter types match those specified by *params*

public Constructor[] getConstructors()
returns all public constructors of this class

public Field getField(String name)
returns an object of the class Field that corresponds to the instance variable of the class that is called *name*

Useful Methods of java.lang.Class - 3

public Field[] getFields()

returns all accessible public instance variables of the class

public Field[] getDeclaredFields()

returns all declared fields (instance variables) of the class

public Method getMethod(String name, Class[] params)

returns an object Method that corresponds to the method called name with a set of parameters params

Useful Methods of java.lang.Class - 4

public Method[] getMethods()

returns all accessible public methods of the class

public Method[] getDeclaredMethods()

returns all declared methods of the class.

public Package getPackage()

returns the package that contains the class

public Class getSuperClass()

returns the superclass of the class

More examples

- The following examples will show you how to use the reflection model to:
  - Create objects
  - Set/get field values
  - Invoke methods using Method.invoke

Example 7: Getting the field value

```
// this example returns the value of the height field of the class Rectangle
import java.lang.reflect.*;
import java.awt.*;
class SampleGet {
   public static void main(String[] args) {
      Rectangle r = new Rectangle(100, 325);
      printHeight(r);
   }
   static void printHeight(Rectangle r) {
      Field heightField; //declares a field
      Integer heightValue;
      Class c = r.getClass(); //get the Class object
      try {
        heightField = c.getField("height"); //get the field object
        heightValue = (Integer)heightField.get(r); //get the value of the field
        System.out.println("Height: " + heightValue.toString());
      } catch (. . .) {
          . . .
      }
   }
}
```


Example 8: Setting the field value

```
// this example sets the width field to 300
import java.lang.reflect.*;
import java.awt.*;

class SampleSet {
   public static void main(String[] args) {
      Rectangle r = new Rectangle(100, 20);
      System.out.println("original: " + r.toString());
      modifyWidth(r, new Integer(300));
      System.out.println("modified: " + r.toString());
   }

   static void modifyWidth(Rectangle r, Integer widthParam ) {
      Field widthField; //declare a field
      Integer widthValue;
      Class c = r.getClass(); //get the Class object
      try {
        widthField = c.getField("width"); //get the field object
        widthField.set(r, widthParam); //set the field to widthParam =300
      } catch (. . . ) {
        . . .
      }
   }
}
```


Example 9: Invoking Methods

```
import java.lang.reflect.*;
class SampleInvoke {
   public static void main(String[] args) {
      String firstWord = "Hello ";
      String secondWord = "everybody.";
      String bothWords = append(firstWord, secondWord);
      System.out.println(bothWords);
   }
   public static String append(String firstWord, String secondWord) {
      String result = null;
      Class c = String.class; //get the Class object of the class String
      Class[] parameterTypes = new Class[] {String.class}; //create a list of
                               // parameters that consists of only one parameter
                               // of type String
      Method concatMethod;
      Object[] arguments = new Object[] {secondWord};
      try {
       //get an object that represents the concat method of the String class
        concatMethod = c.getMethod("concat", parameterTypes);

       //call the concat method through the concatMethod object
        result = (String) concatMethod.invoke(firstWord, arguments);
      } catch (..) {
          . . .
      }
      return result;
   }
}
```

Example 10: Creating Objects

```
import java.lang.reflect.*;
import java.awt.*;

class SampleNoArg {

   public static void main(String[] args) {
      Rectangle r = (Rectangle) createObject("java.awt.Rectangle");
      System.out.println(r.toString());
   }

   static Object createObject(String className) {
      Object object = null;
      try {
          //get the Class object that correponds to className (here
          //java.awt.Rectangle)
          Class classDefinition = Class.forName(className);

          //Create an object of this class
          object = classDefinition.newInstance();
      } catch (. . .) {
          . . .
      }
      return object;
   }
}
```


References

***Java Tutorial*: Reflection API.**

SEG4210 – Advanced Software Design and Reengineering

TOPIC 9 Introduction to C++
For those who know Java
And OO Principles in General

What aspects of C++ are similar to Java?

- **Comments:**

```
//
/* stuff */
```

- **Basic syntax such as:**

```
int I
for(i=1; i<5; i++) { }
class Classname { … )
```

- **C++ is strongly typed**

Some differences from Java

- **Instance methods are generally called member functions**
  - But the concept is the same
- **Instance variables are generally called data members**
  - But the concept is the same
- **To declare a group of members private, public etc., precede the group by**
  - private:
  - public:

Example code for a file Student.h

```
class Student {
 private:
   char *name;
   CourseOfferingList *registrations;
 public:
    // ... maybe other stuff here too ...
    char *name(void);
    void changeName(char *newName);
    void register(CourseOffering *newCourseOffering);
    void deregister (CourseOffering *oldCourseOffering);
}
```


Creating a derived class (subclass) from a base class (superclass)

```
class GraduateStudent : public Student
{
     private:
            Professor *supervisor
     public:
            ...
}
```

- **The keyword ‘public’ on line 1 means the inherited public methods and variables are also to be public here**
- **It is bad design practice to use ‘private’ in this context**
  - Causes inherited things to be private here

Use ‘virtual’ to enable polymorphism

```
class Student {
  ...
  virtual int cgpa(void);
}

class GraduateStudent : public Student {
  // repeating virtual is optional
  virtual int cgpa(void);
}

class UndergradStudent : public student {
  virtual int cgpa(void);
}
```

To declare an abstract method

```
class Student
{
  ...
  virtual int cgpa(void) = 0;
  ...
}
```

- **The term 'pure virtual' is used instead of 'abstract'**

Bodies of methods are declared separately from the declarations in .cpp files

```
char *Student::name(void)
{
  return name;
}

void Student::changeName(char *newName)
{
  delete name;    // get rid of old name
  name = new char[strlen(newName)+1];
  strcpy(name, newName);
}
```


Variables can be…

- **Pointers**
  - As in C, you can have a pointer to a primitive

    ```
    int *c;
    ```
  - You can have a pointer to an object of a class

    ```
    Student *aStudent;
    ```
- **Values (for both primitives and objects)**

  ```
  int c;
  Student s; // the variable contains
             // the storage for the object
  ```
- **References (pointers that are operated on like values)**

  ```
  int &s;
  Student &s;
  ```


Example using pointers vs references

- **Standard C code for a swapping function using pointers**

```
swap(int *a, int *b)
{
    int temp;
    temp = * a;
    (* a) = * b;
    (* b) = temp;
}

swap(&array[pos1],&array[pos2]);
```


Example using pointers vs references, cont.

- **Use references to avoid excessive dereferencing**

```
swap(int &a, int &b)
{
    int temp;
    temp = a;
    a = b;
    b = temp
}

swap(array[pos1],array[pos2]);
```


Calling a member function (method)

- **If the variable student1 contains a pointer**

  ```
  student1->register(courseOffering1);
  ```
- **If student1 contains a reference or a value**

  ```
  student1.register(courseOffering1);
  ```

Creating and deleting objects

- **Use 'new' as in Java to create an object**
- **Use 'delete' to free the memory when done**
  - C++ does not have garbage collection by default
  - You have to define 'destructors'
    - —Discussed later

Declaring a default constructor in a .h file

- **Actually if you omit code, an empty default constructor like this is**

```
class Student
{
    Student();
}
```


Defining the default constructor in the corresponding .cpp file

```
Student::Student()
// Default constructor to initialize an
// empty student
{
  static const defaultName = "unknown";
  name = new char[strlen(defaultName)+1];
  strcpy(name, defaultName);
  registrations = new CourseOfferingList;
  year = 0;
}
```

Constructors with different arguments can be created

You should declare and define a *copy constructor* in each class

- **Used when**
  - Assigning an object to a *newly defined* value variable
    - —The following are equivalent

```
Student student2 = student1;
Student student3(student1);
```

  - Calling by value

Defining a copy constructor

```
Student::Student(const Student &original)
// Constructor to copy a student
{
  char *tempName;

  tempName = original.name();
  name = new char[strlen(tempName)+1];
  strcpy(name, tempName);
  // copy registrations
  registrations = new CourseOfferingList(
    original.registrations());
  year = original.year;
}
```


You should declare and define an overloaded assignment operator

- **Used when**
  - Assigning an object to a *existing* value variable

```
student2 = student1;
```

Defining an overloaded assignment operator

```
Student &Student::operator=(
  const Student &old)
{
  delete name;
  delete registrations;
  // performs function of copy
  // constructor because we cannot call
  // the copy constructor directly
  copy(&old);
  return *this;
}
```


Additional syntactic oddities regarding constructors

- **You can supply default arguments**
  - When an object is created, not all arguments need to be supplied

    ```
    X::X(int, int=0)
    ```

- **You can declare variables in two ways**

  ```
  X x1(1);
  X x2 = 1;
  ```


More object construction syntax

- **Initialization to be performed before the body of a constructor is executed**

  ```
  class X
  {
    int a, b, &c;
  }

  X::X(int i; int j): a(i), b(j), c(a)
  {
  }
  ```

  Is the same as

  ```
  X::X(int i; int j)
  {
   a=i; b=j; c=a;
  }
  ```


Destructors

```
class Student
{
  ~Student(void);
}
```

in body (student.cpp):

```
Student::~Student(void)
// Student destructor
{
  delete name;
  delete registrations;
}
```


Operator overloading

```
NumHolder& NumHolder::operator++()
{
  // update the instance variable
  ++value;
  ...
  // Return reference to self
  return *this;
}

NumHolder NumHolder::operator+(NumHolder &other)
{
  NumHolder sum;
  sum.value(value+other.value());
  return sum
}
```


Stream output

```
int a;
...
// the following scans for an integer
// and puts it in a

cin >> a;
...

// the following prints a string then an
// integer
cout << "the number is " << a;
```

Templates (generic programming)

```
template <class TYPE, int SIZE>
class Stack {
  private:
     TYPE data[SIZE];
     int top
  public
     Stack() {top = 0;}
     void Push (const TYPE &c)
       {data[top++] = c;}
     void Pop(TYPE &c)
       {c = data[--top];}
}

Stack<char, 23> mystack;
Stack<int, 100> anotherstack;
```

SEG4210 – Advanced Software Design and Reengineering

TOPIC 10
C++ Standard Template Library

Standard Template Library (STL)

- The Standard Template Library defines powerful, template-based, reusable components
  - That implements common data structures and algorithms
- STL extensively uses *generic programming* based on *templates*
- Divided into three components:
  - Containers: data structures that store objects of any type
  - Iterators: used to manipulate container elements
  - Algorithms: searching, sorting and many others

Containers

- Three types of containers
  - Sequence containers:
    - linear data structures such as vectors and linked lists
  - Associative containers:
    - non-linear containers such as hash tables
  - Container adapters:
    - constrained sequence containers such as stacks and queues
- Sequence and associative containers are also called first-class containers

Iterators

- **Iterators are pointers to elements of first-class containers**
  Type **const_iterator** defines an iterator to a container element that *cannot* be modified
  Type **iterator** defines an iterator to a container element that *can* be modified
- **All first-class containers provide the members functions** begin() **and** end()
  return iterators pointing to the first and last element of the container

Iterators (cont.)

- If the iterator **it** points to a particular element, then
  - **it++ (or ++it)** points to the next element and
  - ***it** refers to the value of the element pointed to by **it**
- The iterator resulting from end() can only be used to detect whether the iterator has reached the end of the container
- We will see how to use begin() and end() in the next slides

Sequence Containers

- STL provides three sequence containers
  - vector: based on arrays
  - deque (double-ended queue): based on arrays
  - list: based on linked lists

Sequence Containers: **vector**

- The implementation of a vector is based on arrays
- Vectors allow direct access to any element via indexes
- Insertion at the end is normally efficient.
  - The vector simply grows
- Insertion and deletion in the middle is expensive
  - An entire portion of the vector needs to be moved

Sequence Containers: **vector** (cont.)

- When the vector capacity is reached then
  - A larger vector is allocated,
  - The elements of the previous vector are copied and
  - The old vector is deallocated
- To use vectors, we need to include the header <vector>
- Some functions of the class vector include
  - size, capacity, insert…

Example of using the class vector

```
#include <iostream>
#include <vector>  //vector class-template

using std;

int main()
{
  vector<int> v;

  // add integers at the end of the vector
  v.push_back( 2 );
  v.push_back( 3 );
  v.push_back( 4 );

  cout << "\nThe size of v is: " << v.size()
       << "\nThe capacity of v is: "
       << v.capacity();

  // display the content of v
   vector<int>::const_iterator it;

   for (it = v.begin(); it != v.end(); it++)
   {
       cout <<  *it << '\n';
   }
    return 0;
}
```


Sequence Containers: **list**

- list is implemented using a doubly-linked list
- Insertions and deletions are very efficient at any point of the list
  - But you have to have access to an element in the middle of the list first
- bidirectional iterators are used to traverse the container in both directions
- Include header <list> when using lists
- Some functions of the class list
  - push_front, pop_front, remove, unique, merge, reverse and sort

Sequence Containers: **deque**

- deque stands for double-ended queue
- deque combines the benefits of vector and list
- It provides indexed access using indexes (which is not possible using lists)
- It also provides efficient insertion and deletion in the front (which is not efficient using vectors) and the end

deque (cont.)

- Additional storage for a deque is allocated using blocks of memory
  - that are maintained as an array of pointers to those blocks
- Same basic functions as vector, in addition to that
  - deque supports push_front and pop_front for insertion and deletion at beginning of deque

Associative Containers

- Associative containers use keys to store and retrieve elements
- There are four types: multiset, set, multimap and map
  - **all** associative containers maintain keys in sorted order
  - **all** associative containers support bidirectional iterators
  - **set** does not allow duplicate keys
  - **multiset** and **multimap** allow duplicate keys
  - **multimap** and **map** allow keys and values to be mapped

Associative Containers: **multiset**

- Multisets are implemented using a red-black binary search tree for fast storage and retrieval of keys
- Multisets allow duplicate keys
- The ordering of the keys is determined by the STL comparator function object **less<T>**
- Keys sorted with **less<T>** must support comparison using the **<** operator

Example of using a multiset

```
#include <iostream>
#include <set>

using std;

int main()
{
        multiset< int, less< int > > ms;
        ms.insert( 10 );  // insert 10
        ms.insert( 35 );  // insert 35
        ms.insert( 10 );  // insert 10 again (allowed)

        cout << "There are " << ms.count( 10 ); // returns the number of entries = 10

        multiset < int, less< int > >::iterator it; // creates an iterator

        it = ms.find(10);

        if ( it != ms.end() ) // iterator not at end
            cout << "\n10 was found";

        return 0;
}
```


Associative Containers: **set**

- Sets are identical to multisets **except** that they **do not** allow duplicate keys
- To use sets, we need to include the header file <set>

Associative Containers: **multimap**

- Multimaps associate keys to values
- They are implemented using red-black binary search trees for fast storage and retrieval of keys and values
- Insertion is done using objects of the class **pair** (with a key and value)
- Multimaps allow duplicate keys (many values can map to a single key)
- The ordering of the keys is determined by the STL comparator function object **less<T>**

Example of using a multimap

```
#include <iostream>
#include <map>
using std;

typedef multimap< int, double, std::less< int > > mp_type; // creates a mutlimap type

int main()
{
        mp_type mp;

        // value_type is overloaded in multimap to create objects of the class pair

        mp.insert( mp_type::value_type( 10, 14.5 ) );
        mp.insert( mp_type::value_type( 10, 18.5 ) ); //allowed

        cout << "There are " << mp.count( 15 ) << "\n";

        // use iterator to go through mp
        for (mp_type::iterator it = mp.begin(); it != mp.end(); it ++)
            cout << it->first << '\t' << it->second << '\n';

        return 0;
}
```

Associative Containers: **map**

- They are implemented using red-black binary search trees just like multimaps
- Unlike multimaps, they allow storage and retrieval of **unique** key/value pairs. **They do not allow duplicates of keys**
- The class map overloads the [ ] operator to access values in a flexible way

Example of using a map

- The following code fragment shows how to use indexes with an object of the class map

```
map<int, double, less<int>>  map_obj;

// sets the value of key 20 to 125.25. If subscript
// 20 is not in map then creates new
// key=20,  value=125.25 pair

map_obj [20] = 125.25;
```


Container Adapters

- STL supports three container adapters:
  - stack, queue and priority_queue
- They are implemented using the containers seen before
  - They do not provide actual data structure
- Container adapters do not support iterators
- The functions **push** and **pop** are common to all container adapters

Container Adapters: **stack**

- Last-in-first-out data structure
- They are implemented with **vector**, **list**, and **deque (by default)**
- Header file <stack>
- Example of creating stacks
  - A stack of int using a vector: **stack < int, vector < int > > s1;**
  - A stack of int using a list: **stack < int, list < int > > s2;**
  - A stack of int using a deque: **stack < int > s3;**

Container Adapters: **queue**

- First-in-first-out data structure
- Implemented with **list** and **deque (by default)**
- Header file <queue>
- Example:
  - A queue of int using a list: **queue <int, list<int>> q1;**
  - A queue of int using a deque: **queue <int> q2;**

Container Adapters: **priority_queue**

- Insertions are done in a sorted order
- Deletions from front similar to a queue
- They are implemented with **vector (by default)** or **deque**
- The elements with the highest priority are removed first
  - less<T> is used by default for comparing elements
- Header file <queue>

Algorithms

- STL separates containers and algorithms
  - The main benefit is to avoid virtual function calls
  - This cannot be done in Java or C# because they do not have such flexible mechanisms for dealing with function objects
    - Smalltalk does … all the following can be easily implemented in Smalltalk
- The subsequent slides describe most common STL algorithms

Fill and Generate

- **fill(iterator1, iterator2, value);**
  fills the values of the elements between iterator1 and iterator2 with *value*
- **fill_n(iterator1, n, value);**
  changes specified number of elements starting at iterator1 to *value*
- **generate(iterator1, iterator2, function);**
  similar to fill except that it calls a function to return value
- **generate_n(iterator1, n, function)**
  same as fill_n except that it calls a function to return value

Comparing sequences of values

- **bool equal(iterator1, iterator2, iterator3);**
  - compares sequence from iterator1 to iterator2 with the sequence beginning at iterator3
  - return true is they are equal, false otherwise
- **pair mismatch(iterator1, iterator2, iterator3);**
  - compares sequence from iterator1 to iterator2 with the sequence starting at iterator3
  - returns a **pair** object with iterators pointing to where mismatch occurred
  - example of the a pair object

```
pair <<vector>::iterator, <vector>::iterator> par_obj;
```


Removing elements from containers

- **iterator remove(iterator1, iterator2, value);**
  - removes all instances of *value* in a range iterator1 to iterator2
  - does not physically remove the elements from the sequence
  - moves the elements that are not removed forward
  - returns an iterator that points to the "new" end of container
- **iterator remove_copy(iterator1, iterator2, iterator3, value);**
  - copies elements of the range iterator1-iterator2 that are not equal to *value* into the sequence starting at iterator3
  - returns an iterator that points to the last element of the sequence starting at iterator3

Replacing elements (1)

- **replace(iterator1, iterator2, value, newvalue);**
  replaces *value* with *newvalue* for the elements located in the range iterator1 to iterator2
- **replace_if(iterator1, iterator2, function, newvalue);**
  replaces all elements in the range iterator1-iterator2 for which *function* returns true with *newvalue*

Replacing elements (2)

- **replace_copy(iterator1, iterator2, iterator3, value, newvalue);**
  replaces and copies elements of the range iterator1-iterator2 to iterator3
- **replace_copy_if(iterator1, iterator2, iterator3, function, newvalue);**
  replaces and copies elements for which *function* returns true where iterator3

Search algorithms

- **iterator find(iterator1, iterator2, value)**
  returns an iterator that points to first occurrence of value
- **iterator find_if(iterator1, iterator2, function)**
  returns an iterator that points to the first element for which *function* returns true.

Sorting algorithms

- **sort(iterator1, iterator2)**
  sorts elements in ascending order
- **binary_search(iterator1, iterator2, value)**
  searches in an ascending sorted list for *value* using a binary search

Copy and Merge

- **copy(iterator1, iterator2, iterator3)**
  copies the range of elements from iterator1 to iterator2 into iterator3
- **copy_backward(iterator1, iterator2, iterator3)**
  copies in reverse order the range of elements from iterator1 to iterator2 into iterator3
- **merge(iter1, iter2, iter3, iter4, iter5)**
  ranges iter1-iter2 and iter3-iter4 must be sorted in ascending order and copies both lists into iter5 in ascending order

Unique and Reverse order

- **iterator unique(iterator1, iterator2)**
  - removes (logically) duplicate elements from a sorted list
  - returns iterator to the new end of sequence
- **reverse(iterator1, iterator2)**
  - reverses elements in the range of iterator1 to iterator2

Utility algorithms (1)

- **random_shuffle(iterator1, iterator2)**
  randomly mixes elements in the range iterator1-iterator2
- **int count(iterator1, iterator2, value)**
  returns number of instances of value in the range
- **int count_if(iterator1, iterator2, function)**
  counts number of instances for which *function* returns true

Utility algorithms (2)

- **iterator min_element(iterator1, iterator2)**
  returns iterator to smallest element
- **iterator max_element(iterator1, iterator2)**
  returns iterator to largest element
- **accumulate(iterator1, iterator2)**
  returns the sum of the elements in the range

Utility algorithms (3)

- **for_each(iterator1, iterator2, function)**
  calls function on every element in range
- **transform(iterator1, iterator2, iterator3, function)**
  calls *function* for all elements in range iterator1-iterator2, and copies result to iterator3

Algorithms on sets (1)

- **includes(iter1, iter2, iter3, iter4)**
  returns true if iter1-iter2 contains iter3-iter4. Both sequences must be sorted
- **set_difference(iter1, iter2, iter3, iter4,iter5)**
  copies elements in range iter1-iter2 that do not exist in second range iter3-iter4 into iter5
- **set_intersection(iter1, iter2, iter3, iter4, iter5)**
  copies common elements from the two ranges iter1-iter2 and iter3-iter4 into iter5

Algorithms on sets (2)

- **set_symmetric_difference(iter1, iter2, iter3, iter4, iter5)**
  copies elements in range iter1-iter2 that are not in range iter3-iter4 and vice versa, into iter5. Both sets must be sorted
- **set_union(iter1, iter2, iter3, iter4, iter5)**
  copies elements in both ranges to iter5. Both sets must be sorted

References

- **"C++, How to program", Harvey M. Deitel, Paul J. Deitel , 4th edition, Prentice Hall**
- **"The C++ Programming Language", Bjarne Stroustrup, 3rd Edition, Addison-Wesley**