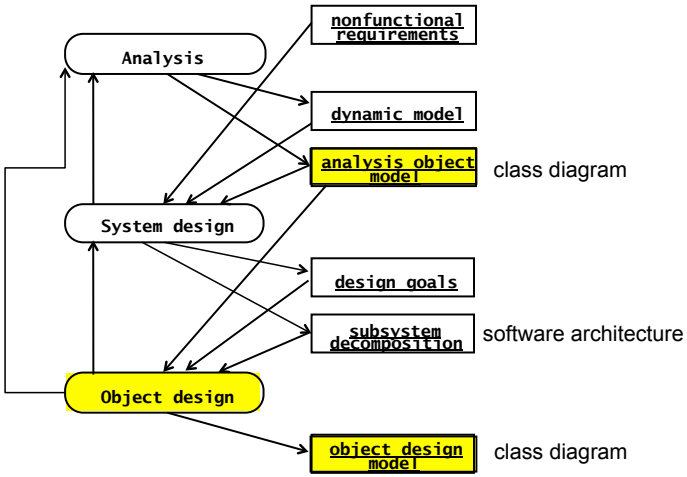
Chapter 9:
Object design: Specifying Interfaces

CS 4354
Fall 2012

Jill Seaman

Review: Leading up to Object design



Main activities of Object Design

- **Reuse**: Developers identify off-the-shelf components and design patterns to make use of existing solutions Chapter 8
- **Interface specification**: Developers precisely describe class interface(s) to represent each subsystem interface.
  - ✦The subsystem API Chapter 9
- **Restructuring**: Developers transform the object design model to improve its understandability and extensibility.
  - ✦In order to meet design goals. Chapter 10
- **Optimization**: Developers transform the object design model to address performance criteria.
  - ✦Such as response time or memory utilization. Chapter 10

9.2 An overview of Interface Specifications

- **Goal**: Describe the interface of each object precisely enough so objects realized by individual developers fit together with minimal integration issues.
- **Identify missing attributes and operations**
  - ✦examine subsystem service an analysis object, identify missing pieces.
- **Specify visibility and signatures**
  - ✦complete prototypes of operations, private/public/package/protected.
- **Specify Contracts** (constraints/conditions)
  - ✦Invariants for classes
  - ✦Preconditions and postconditions for operations

9.3 Interface Specification Concepts
9.3.1 Class implementor, user, extender

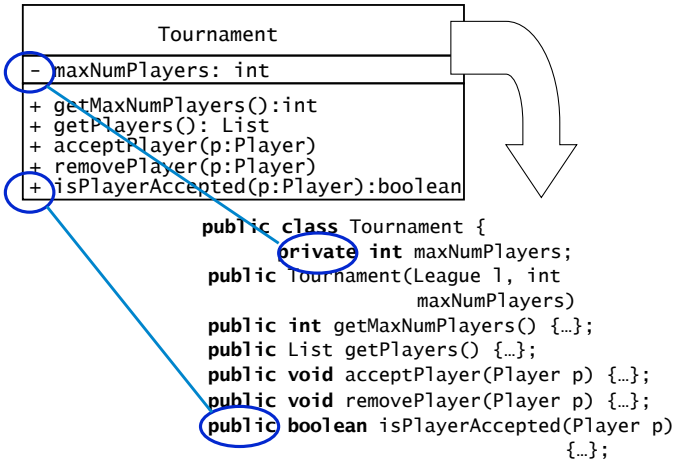
Role of a developer, with respect to a given class under consideration:

- **Class implementor:** responsible for realizing (implementing) the class under consideration.
  - ✦The interface specification is a work assignment.
- **Class user:** invokes the operations provided by the class under consideration during the realization of another class, called the **client class**.
  - ✦The interface specification is a user's guide.
- **Class extender** develops specialization (subclass) of the class under consideration.
  - ✦The interface specification is both a work assignment and user's guide.

9.3.2 Types, Signatures, and Visibility

- **Type:** range of values (int, string, List, boolean, float/double/real)
  - ✦attribute has one type.
  - ✦operation has parameters with types, and (optional) return type.
- **Visibility** of an attribute or operation: which classes have access.
  - ✦**private (-):** accessed only in class in which it is defined. (for use by implementor)
  - ✦**protected (#):** accessed in class in which it is defined and its subclasses. (for use by extender)
  - ✦**public (+):** accessed by any class. Part of the public interface. (for use by class user, client classes)
  - ✦**package (~):** accessed by any class in the same package.

Types, Signatures, and Visibility: example



9.3.3 Contracts: Invariants, Preconditions and Postconditions

- **Contracts** are constraints on a class that enable class users, implementers, and extenders to share the same assumptions about the class.

  Contracts include three types of constraints:

- An **invariant** is a predicate that is always true for all instances of a class.
- A **precondition** is a predicate that must be true before an operation is invoked.
- A **postcondition** is a predicate that must be true after an operation is invoked.

9.3.4 Object Constraint Language

- **Object Constraint Language (OCL)** is a formal language that allows constraints to be formally specified
  - ✦on single model elements (e.g., attributes, operations, classes)
  - ✦or groups of model elements (e.g., associations and participating classes).
- A constraint is expressed as a boolean expression returning the value True or False.
- A constraint can be depicted as a note attached to the constrained UML element.
- Alternatively they can be expressed in a textual form.

OCL: textual format, invariants

- Example:

```
context Tournament inv:
   self.getMaxNumPlayers() > 0
```

- **`context`** indicates the entity to which the expression applies.
- This is followed by **inv** (invariant), **pre** (precondition) or **post** (postcondition).
- The second line is the actual OCL expression
- `self` denotes all instances of the class. Same as `this` in Java. Can be omitted if not ambiguous.

OCL: textual format, pre and post-conditions

- context for pre/post is a class operation (for inv it is a class)
- parameters to context operations can be used as variables in the ocl expression.
- @pre.operation(): used to refer to a value of attribute/operation as it was before the operation was called.

```
context Tournament::acceptPlayer(p) pre:
  not isPlayerAccepted(p)

context Tournament::acceptPlayer(p) pre:
  getNumPlayers() < getMaxNumPlayers()

context Tournament::acceptPlayer(p) post:
  isPlayerAccepted(p)

context Tournament::acceptPlayer(p) post:
  getNumPlayers() = self@pre.getNumPlayers() + 1
```


OCL: Collections

- OCL provides additional data types called **collections**.
- These are generally used to represent the objects accessed by navigating across associations.
- OCL collections: Sets, Sequences (ordered), Bags (multi-sets).
- OCL operations for accessing collections:
  - ✦size, includes (membership), select (filter), union, intersection
- OCL also supports “for all” and “there exists” expressions.
  - ✦collection->forAll(variable|expression) is True if expression is True for all elements in the collection
  - ✦collection->exists(variable|expression) is True if expression is True for all elements in the collection

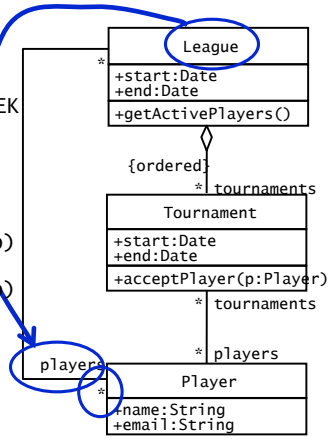
OCL: examples

Local attribute navigation

```
context Tournament inv:
  end - start <= Calendar.WEEK
```

Directly related class navigation

```
context
  Tournament::acceptPlayer(p)
  pre:
  league.players->includes(p)
```



OCL: Collections, forall, exists examples

- OCL forall quantifier

```
/* All Matches in a Tournament occur within the
Tournament's time frame */

context Tournament inv:
matches->forAll(m:Match |
  m.start.after(t.start) and m.end.before(t.end))
```

- OCL exists quantifier

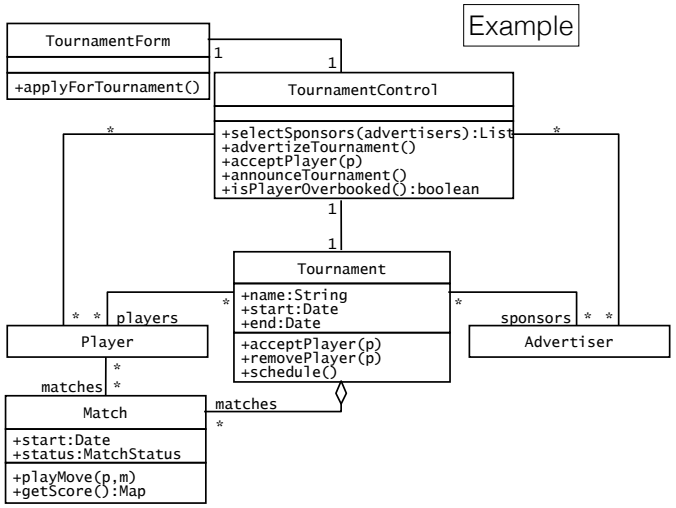
```
/* Each Tournament conducts at least one Match on the
first day of the Tournament */

context Tournament inv:
  matches->exists(m:Match | m.start.equals(start))
```


9.4 Interface Specification Activities
9.4.1 Identifying Missing Attributes and Operations

- During this step, we examine service descriptions of the subsystem and identify missing attributes and operations.
- This is a chance to refine the object model after it has potentially been modified during system design.
  - ✦Partitioning into subsystems.
  - ✦Addition of boundary use cases.
- We want the model to be as complete as possible before the next steps.

9.4.2 Specifying Types, Signatures, and Visibility

Example



9.4.3 Specifying Pre- and Postconditions

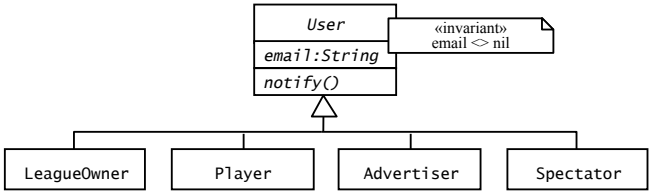
- We define contracts for each public operation of each class.
- **Preconditions** are the part of the contract the class user must respect.
- **Postconditions** are the part of the contract that the class implementor guarantees if the user fulfills his/her part.

9.4.4 Specifying Invariants

- Invariants provide an overview of the essential properties of the class
  - ✦They are more difficult to write than preconditions and postconditions.
  - ✦Often involve collections, forall and exists
- Invariants constitute a permanent contract that extends and overwrites the operation-specific contracts.
- It is easy to generate a large number of invariants for each class
  - ✦Focus on simple, short constraints that describe boundary cases that may not otherwise be obvious.

9.4.5 Inheriting Contracts

- **Contract inheritance:** the class user expects that a contract that holds for the superclass also holds for the subclass.



- If we decide Spectators do not require an email address,
  - ✦Spectators should be taken out of the User hierarchy OR
  - ✦the invariant should be revised (terms of contract reformulated)

Inheriting contracts:

Contracts are inherited in the following manner:

- *Precondition*. A method of subclass is allowed to <u>weaken</u> the preconditions of the method it overrides.
  - ✦In other words, the subclass method is allowed to handle MORE cases than its superclass
- For example,
  - ✦A concrete TournamentStyle class, SimpleKnockOutStyle, can deal with any number of Players that is a power of 2.
  - ✦A complexKnockOutStyle class which is a subclass of the SimpleKnockOutStyle could <u>weaken</u> this precondition by planning Tournaments for any number of Players.
  - ✦The subclass method does not violate the superclass contract.

Inheriting contracts:

- *Postcondition.* Methods must ensure the same postconditions as their ancestors or stricter ones.
  - ✦In other words, the subclass method is allowed to guarantee even more specific results than the superclass
- A negative example:
  - ✦Assume a Set is implemented by inheriting from a List (this is implementation inheritance).
  - ✦The postcondition of List.add() is that the size of the List increases by 1.
  - ✦But by adding an element to a set does not necessarily increase the size of the set. So Set should not be implemented as a subclass of *List.*

Inheriting contracts:

- *Invariants*. A subclass must respect all invariants of its superclasses. However, a subclass can strengthen the inherited invariants.
- For example,
  - ✦List inherits from Collection.
  - ✦Collection has an invariant specifying that its size cannot be negative.
  - ✦List must respect that but can add that its elements are ordered.