

Object-Oriented Software Engineering

Using UML, Patterns, and Java

Chapter 9, Object Design: Specifying Interfaces



9. Object Design: Specifying Interfaces

- During object design
 - our understanding of each object **deepens**
 - specify **the type signatures** and **the visibility** of each of the operations
 - Describe **the conditions** under which an operation can be invoked and those under which the operation raises an exception
- the focus of object design
 - on specifying the **boundaries between objects**
- The focus of interface specification
 - for developers to **communicate clearly and precisely** about increasingly **lower-level details** of the system.

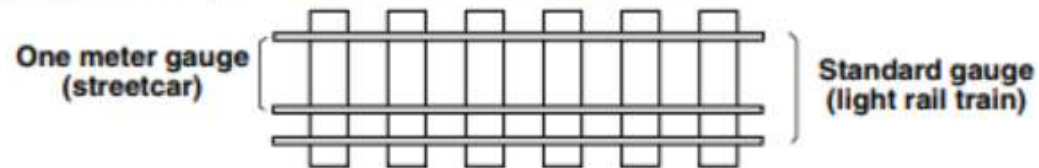
- The interface specification activities of object design include
 - identifying missing attributes and operations
 - specifying type signatures and visibility
 - specifying invariants
 - specifying preconditions and postconditions.

9.1 Introduction: A Railroad Example

- For **rail tracks**, the interface of **the streetcar** is the **wheels**.
- For **passengers**, the interface to **the streetcar** is **the door**. If the door of the street car is higher, the passengers need a higher platform to enter the street car
- For **the signaling system**, the interface is the driver who **monitors the traffic signals**.
- objects interact with other objects through **interfaces** that include a set of operations, each accepting a set of parameters and producing a result, and a set of assumptions about the behavior of each operation

Gauge(차폭)/Station platform(역플랫폼)/Signaling(신호)

- Gauge.* The light rail system used a standard gauge (1450 mm), and the streetcar system used a meter gauge (1000 mm). As several stations needed to accommodate several lines, the tracks had to be fitted with three rails instead of the usual two, so that both types of cars could use the same track. In addition to more rails, this resulted in more complex switches.



- Station platforms.* Streetcars were designed for passengers who entered the cars from the street level. Light rail cars were designed for passengers who enter from a raised platform. For stations that accommodated both types of rolling stock, platforms were raised on one end and low on the other end. Because each part of the platform must be at least as long as its respective trains, all platforms became much longer.



Streetcar

Light rail train

- Signaling.* Streetcars observed signals that were similar to traffic lights observed by cars: traffic lights open and close on a periodic basis, mostly independent of the current traffic situation. Light rail signaling, however, is similar to freight train signaling, in which trains are dispatched from a central location and track circuits monitor the location of each train. Consequently, the signaling for the light rail had to be compatible with street car rolling stock.

9.2 An Overview of Interface Specification

- **The goal** of interface specification,
 - to describe **the interface** of each object **precisely** enough so that objects realized by individual developers fit together with minimal **integration** issues.
- interface specification includes the following **activities**:
 - Identify missing attributes and operations.
 - Specify visibility and signatures.
 - specify the return type of each operation as well as the number and type of its parameters.
 - Specify contracts.

9.3 Interface Specification Concepts

- present the principal concepts of interface specification:
 - Class Implementor, Class Extender, and Class User (Section 9.3.1)
 - Types, Signatures, and Visibility (Section 9.3.2)
 - Contracts: Invariants, Preconditions, and Postconditions (Section 9.3.3)
 - *Object Constraint Language (Section 9.3.4)*
 - *OCL Collections: Sets, Bags, and Sequences (Section 9.3.5)*
 - *OCL Qualifiers: forAll and exists (Section 9.3.6).*

9.3.1 Class Implementor, Class Extender, and Class User

- Developers(Class implementor, Class user, Class Extender) view the specifications from radically different point of views.
- The class implementor
 - responsible for **realizing** the class under consideration
 - **design** the internal **data structures** and **implement** the code for each **public operation**
- The class user (== Client user)
 - invokes the operations provided by the class under consideration during the realization of another class, called **the client class**
 - **the interface specification** discloses **the boundary** of the class in terms **of the services** it provides and the assumptions it makes about the client class.

- The class extender
 - develops **specializations of the class** under consideration
 - the class extenders focus **on specialized versions** of the same services
 - **the interface specification** specifies **the current behavior** of the class and **any constraints** on the services provided by the specialized class

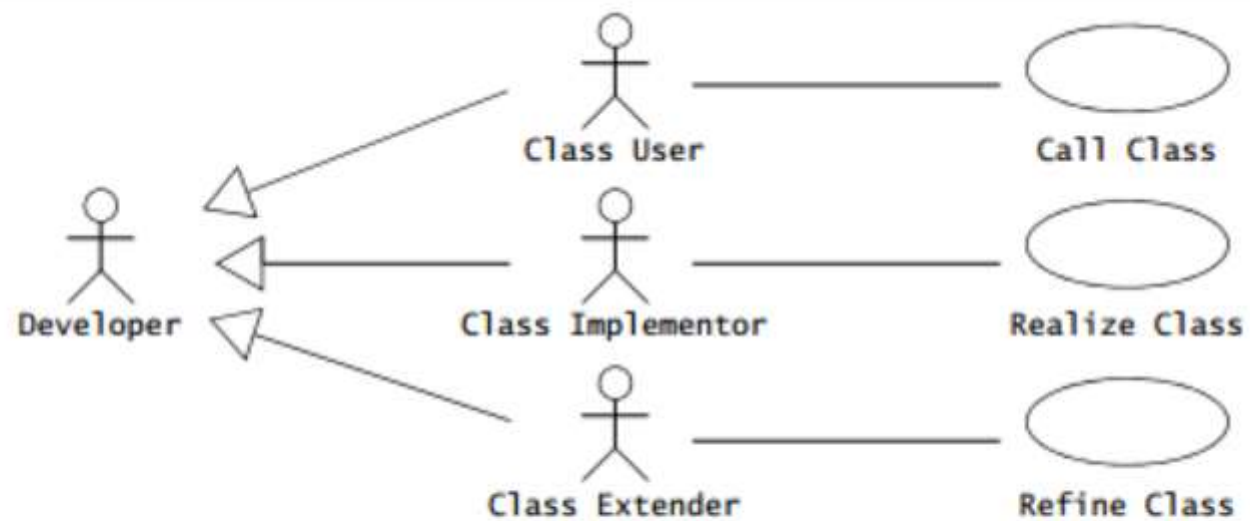


Figure 9-1 The Class Implementor, the Class Extender, and the Class User role (UML use case diagram).

- Class implementor
 - responsible for realizing the Game class, including **operations that apply to all Games**
- class users of Game
 - Developers responsible **for League and Tournament**
 - **invoke operations** provided by the *Game* interface to organize and start Matches
- extenders of Game.
 - The **TicTacToe and Chess** classes are concrete Games that provide specialized extensions to the Game class.

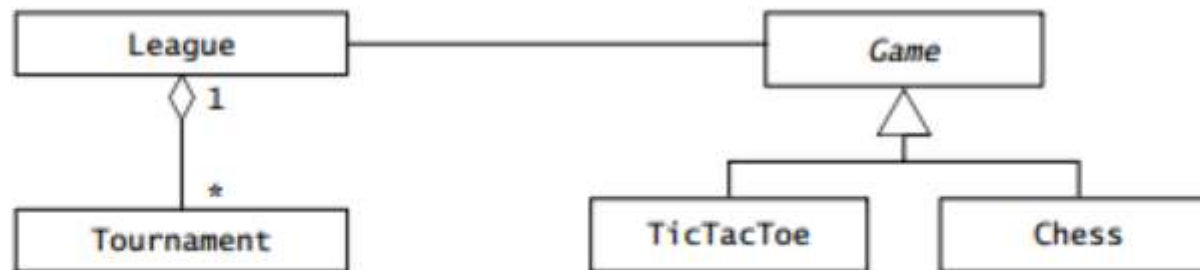
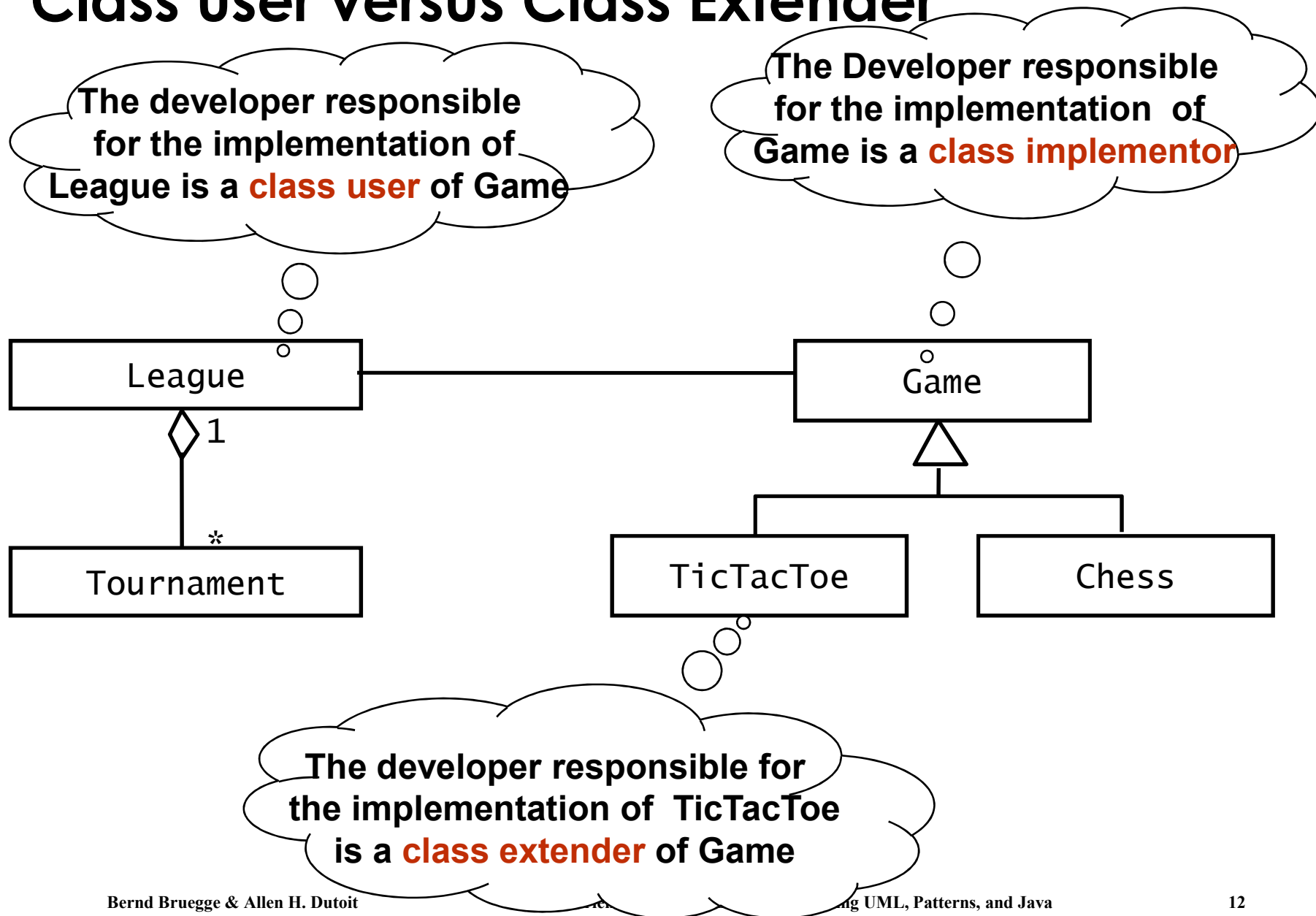


Figure 9-2 ARENA Game abstract class with user classes and extender classes.

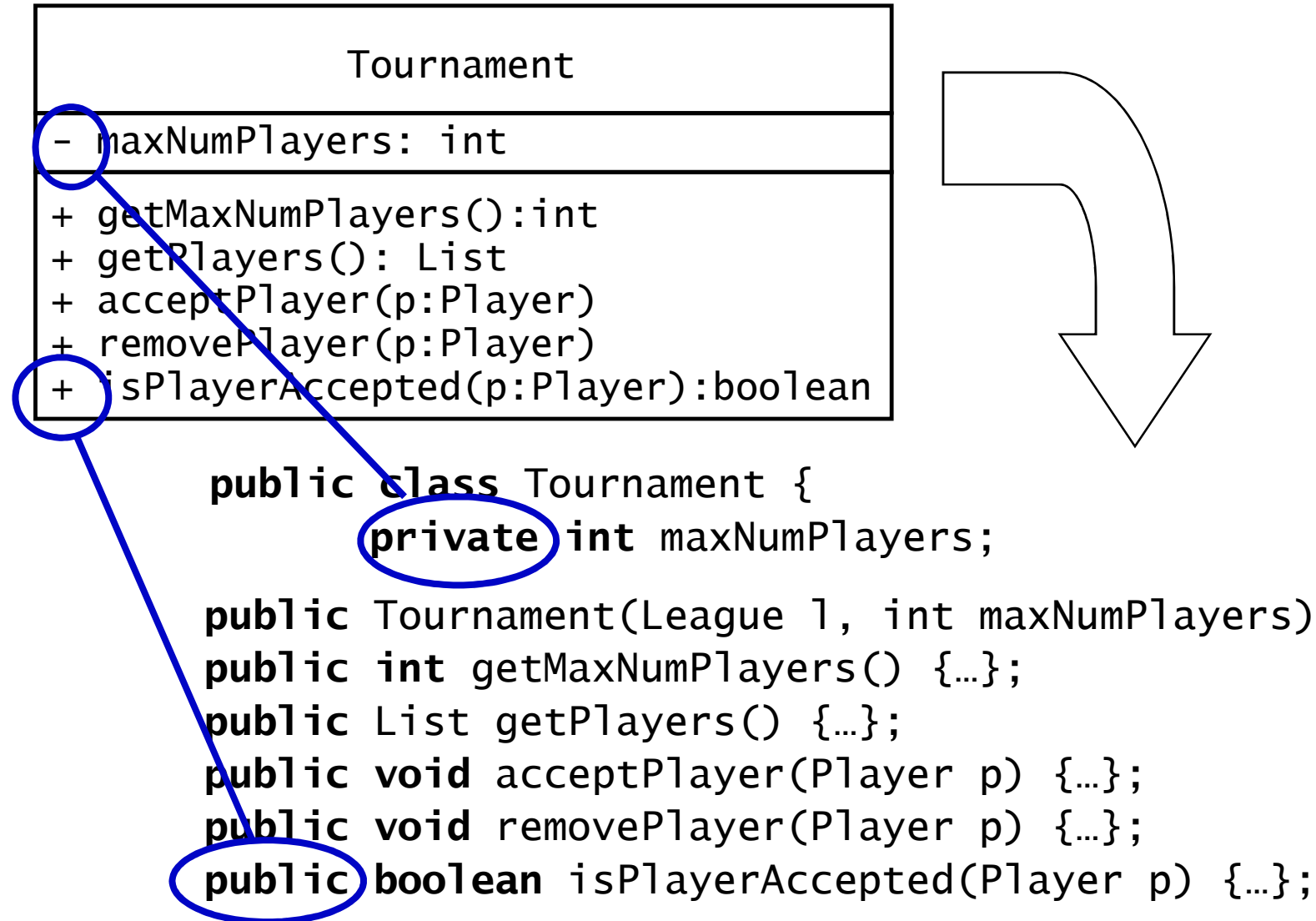
Class user versus Class Extender



9.3.2 Types, Signatures, and Visibility

- The **type** of an attribute
 - specifies the range of values the attribute can take and the operations that can be applied to the attribute.
- the **signature** of the operation.
 - The tuple made out of the types of its parameters and the type of the return value is called
- The **visibility** of an attribute or an operation
 - a mechanism for specifying whether the attribute or operation can be used by other classes or not
- UML defines four levels of **visibility**:
 - A **private** attribute
 - A **protected** attribute or operation
 - can be accessed by the class in which it is defined and by any descendant of that class
 - A **public** attribute or operation
 - **package**

Fig 9.3 Implementation of UML Visibility in Java



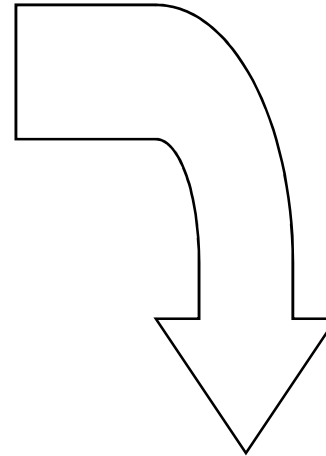
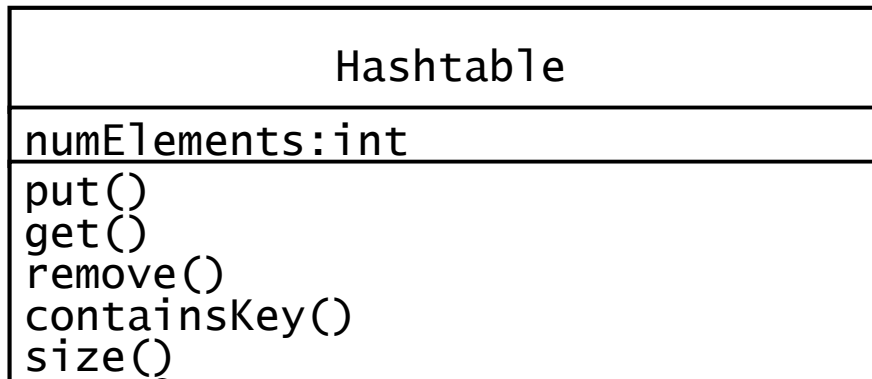
Information Hiding Heuristics

- Carefully define the public interface for classes as well as subsystems
 - For subsystems use a **façade design** pattern if possible
- Always apply the “**Need to know**” principle:
 - Only if somebody needs to access the information, make it publicly possible
 - Provide only well defined channels, so you always know the access
- **The fewer details** a class user has to know
 - the easier the class can **be changed**
 - the less likely they will be affected by any changes in **the class implementation**
- Trade-off: Information hiding vs. efficiency
 - Accessing a private attribute might be too slow.

Information Hiding Design Principles

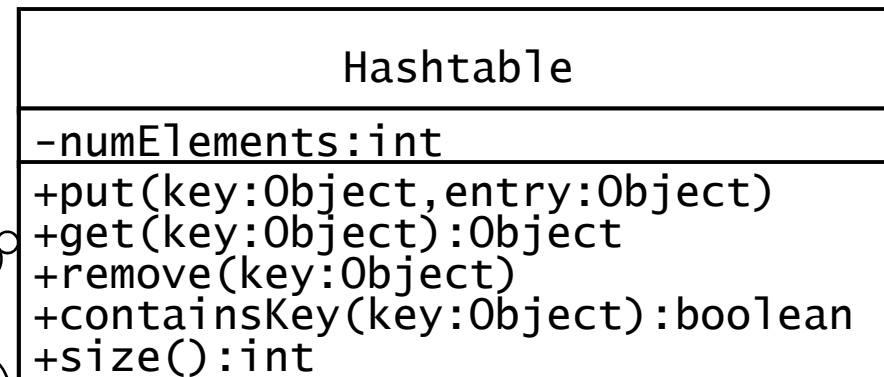
- Only the operations of a class are allowed to manipulate its attributes
 - Access attributes only via operations
- Hide external objects at subsystem boundary
 - Define **abstract class interfaces** which mediate between the external world and the system as well as between subsystems
- Do not apply an operation to the result of another operation
 - Write a new operation that combines the two operations.

Add Type Signature Information

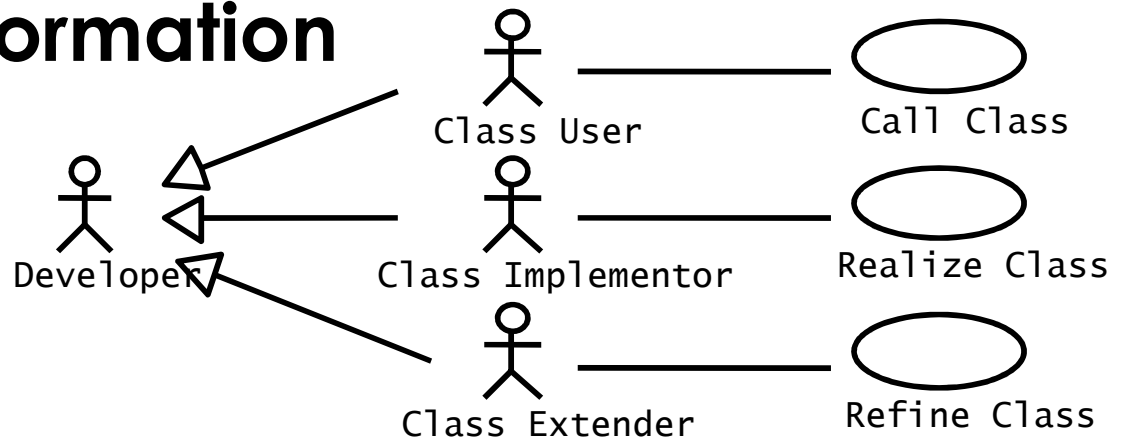


Attributes and operations without visibility and type information are ok during requirements analysis

During object design, we decide that the hash table can handle any type of keys, not only Strings.



Add Visibility Information



Class user ("Public"): +

- Public attributes/operation can be accessed by any class

Class implementor ("Private"): -

- Private attributes and operations can be accessed only by the class in which they are defined
- They cannot be accessed by subclasses or other classes

Class extender ("Protected"): #

- Protected attributes/operations can be accessed by the class in which they are defined and by any descendent of the class.

9.3.3 Contracts: Invariants, Preconditions, and Postconditions

- Contracts
 - **constraints** on a class that enable class users, implementors, and extenders to **share the same assumptions about the class**
- Contracts include three types of constraints
 - 1) An invariant
 - a predicate that is **always true** for all instances of a class
 - used to specify consistency constraints among class attributes.
 - 2) A precondition
 - a predicate that must be true before an operation is invoked
 - used to specify constraints that a class user must meet before calling the operation.

- 3) A postcondition
 - a predicate that must be true after an operation is invoked
 - used to specify constraints that the class implementor and the class extender must ensure after the invocation of the operation.

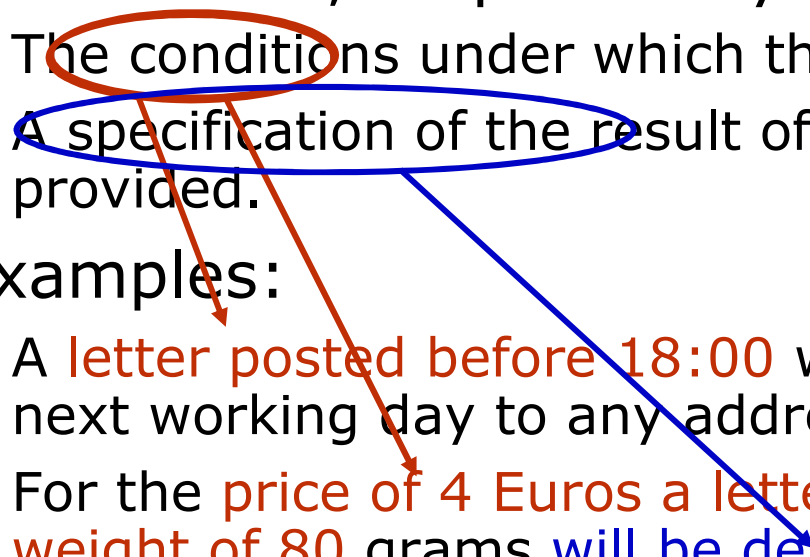
Contract

- **Contract:** A lawful agreement between two parties in which both parties accept obligations and on which both parties can found their rights
 - The remedy for breach of a contract is usually an award of money to the injured party
- **Object-oriented contract:** Describes the services that are provided by an object **if certain conditions are fulfilled**
 - services = "obligations", conditions = **"rights"**
 - The remedy for breach of an OO-contract is the generation of **an exception**.

Object-Oriented Contract

- An **object-oriented contract** describes the services that are provided by an object. For each service, it specifically describes two things:
 - **The conditions** under which the service will be provided
 - A specification of the result of the service
- Examples:
 - A letter posted before 18:00 will be delivered on the next working day to any address in Germany
 - For the price of 4 Euros a letter with a maximum weight of 80 grams will be delivered anywhere in the USA within 4 hours of pickup.

Object-Oriented Contract

- An **object-oriented contract** describes the services that are provided by an object. For each service, it specifically describes two things:
 - The conditions under which the service will be provided
 - A specification of the result of the service that is provided.
 - Examples:
 - A letter posted before 18:00 will be delivered on the next working day to any address in Germany.
 - For the price of 4 Euros a letter with a maximum weight of 80 grams will be delivered anywhere in Germany within 4 hours of pickup.
- 

Modeling OO-Contracts

- Natural Language
- Mathematical Notation
- Models and contracts:
 - A language for the formulation of constraints with the formal strength of the mathematical notation and the easiness of natural language:
 - ⇒ UML + OCL (Object Constraint Language)
 - Uses the abstractions of the UML model
 - OCL is based on predicate calculus

Contracts and Formal Specification

- Contracts enable the caller and the provider to **share the same assumptions** about the class
- A contract is an exact specification of the interface of an object
- A contract include three types of constraints:
 - **Invariant**:
 - A predicate that is always true for all instances of a class
 - **Precondition ("rights")**:
 - Must be true before an operation is invoked
 - **Postcondition ("obligation")**:
 - Must be true after an operation is invoked.

9.4 Interface Specification **Activities**

- Identifying Missing Attributes and Operations (Section 9.4.1)
- Specifying Type Signatures and Visibility (Section 9.4.2)
- Specifying Preconditions and Postconditions (Section 9.4.3)
- Specifying Invariants (Section 9.4.4)
- Inheriting Contracts (Section 9.4.5).

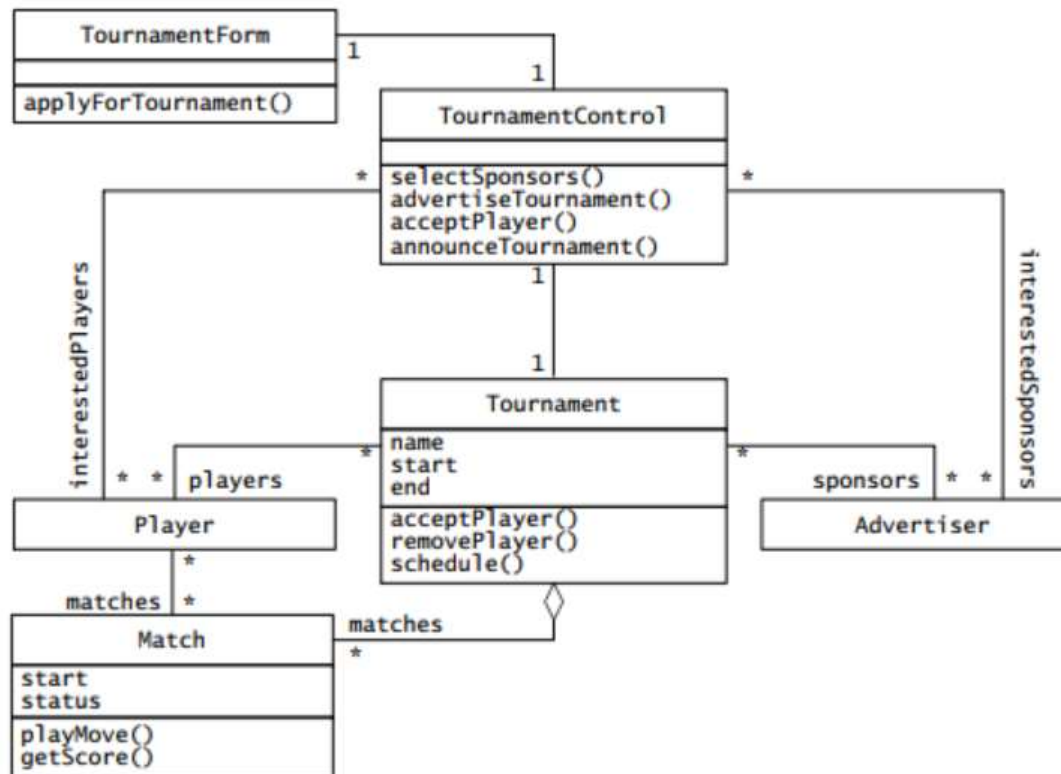


Figure 9-9 Analysis objects of ARENA identified during the analysis of AnnounceTournament use case (UML class diagram). Only selected information is shown for brevity.

9.4.1 Identifying Missing Attributes and Operations

- examine the service description of the subsystem and identify **missing attributes and operations**.
- Figure 9-10
 - This sequence diagram leads to the identification of a new operation

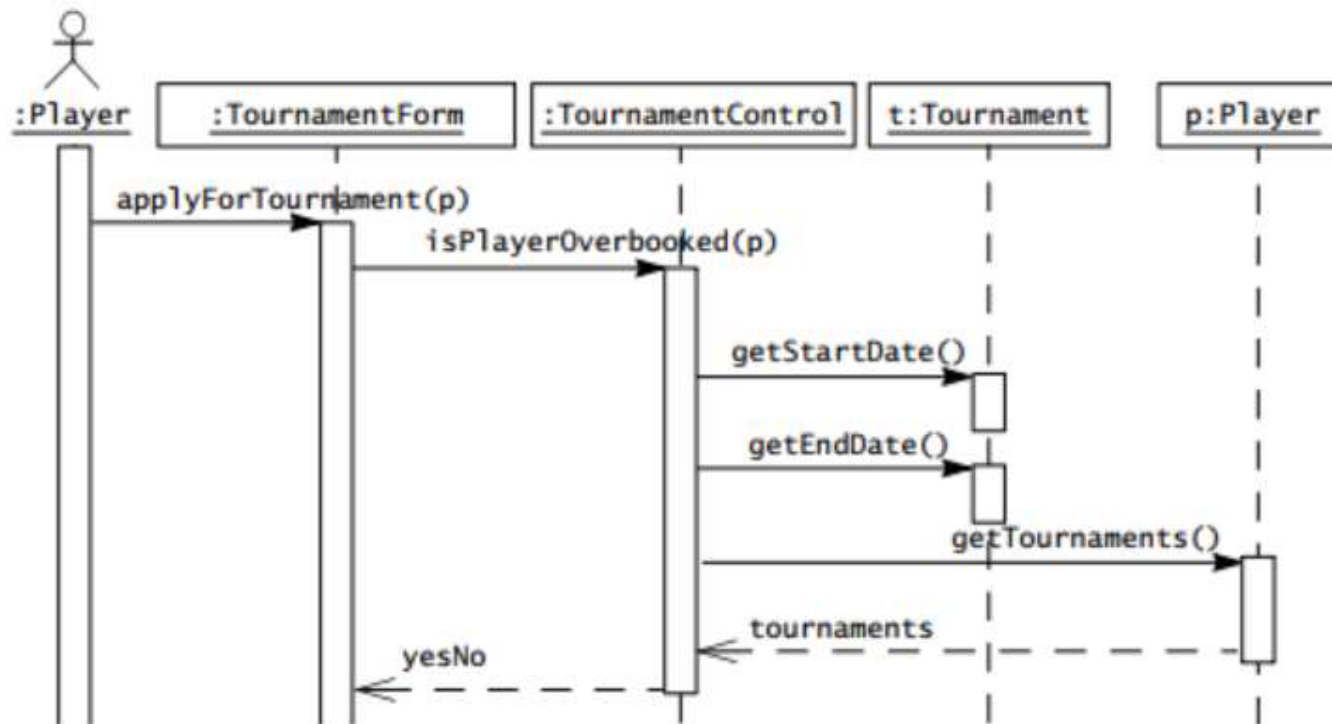


Figure 9-10 A sequence diagram for the `applyForTournament()` operation (UML sequence diagram). This sequence diagram leads to the identification of a new operation, `isPlayerOverbooked()` to ensure that players are not assigned to Tournaments that take place simultaneously.

9.4.2 Specifying Types, Signatures, and Visibility

- specify the types of the attributes, the signatures of the operations, and the visibility of attributes and operations.
- 1) **Specifying types** refines the object design model in two ways.
 - First, we add detail to the model by specifying the **range of each attribute**
 - Second, we map classes and attributes of the object model to **built-in types** provided by the development environment
 - selecting **String** to represent the name attributes of Leagues and Tournaments

- 2) consider the relationship between the classes we identified and the classes from existing components
- 3) determine the visibility of each attribute and operation during this step

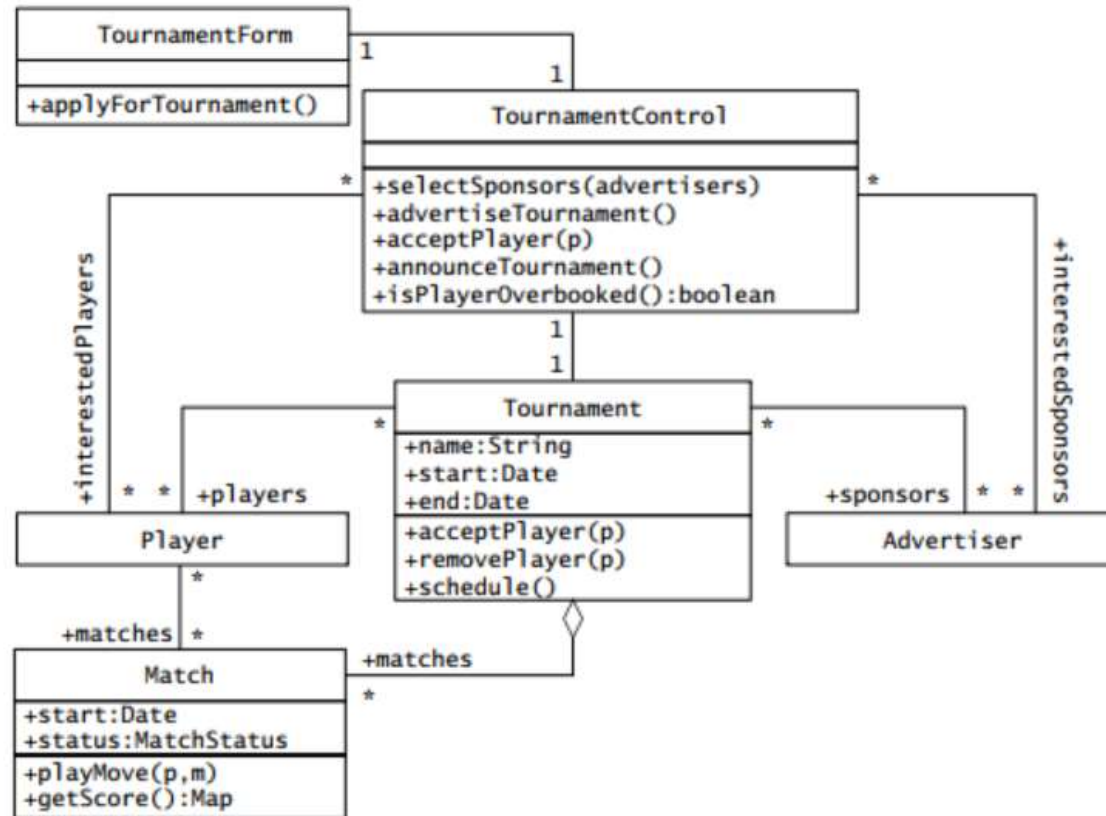
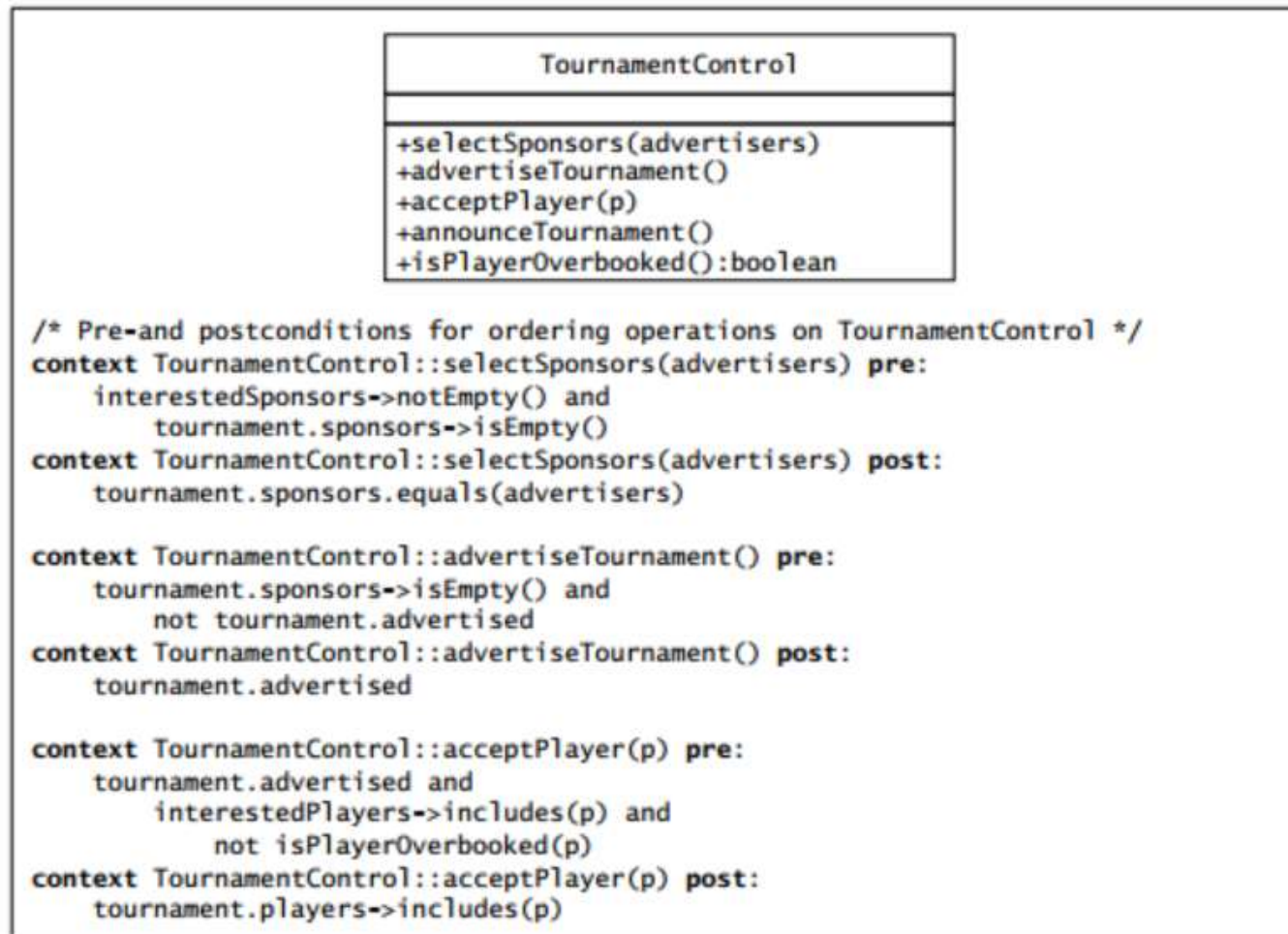


Figure 9-11 Adding type information to the object model of ARENA (UML class diagram). Only selected information is shown for brevity.

9.4.3 Specifying Pre- and Postconditions

- The preconditions of an operation describe the part of the contract that the class user must respect.
- The postconditions describe what the class implementor guarantees in the event the class user fulfilled her part of the contract



9.4.4 Specifying Invariants

- provide an overview of the essential properties of the class
- Invariants constitute a permanent contract that extends and overwrites the operation-specific contracts.

Heuristics for writing readable constraints

Focus on the lifetime of a class. Constraints that are specific to operations or that hold only when the object is in certain states are better expressed as pre- and postconditions. For example:

- “Different players have different E-mail addresses” is an invariant.
- “A player can apply to only one tournament at a time” is a precondition of the `TournamentForm.applyForTournament()` operation.

Identify special values for each attribute. Zero and null values, attributes that have unique values within a certain scope, and attributes that depend on other attributes are often sources of misunderstandings and errors. For example:

- Matches involve at least one player.
- Completed matches have exactly one winner.

Identify special cases for associations. Identify, in particular, any special case that cannot be specified with multiplicity alone. For example:

- `tournament.players` is a subset of `tournament.league.players`

Identify ordering among operations. For example, see `TournamentControl` in Section 9.4.3.

Use helper methods to compute complex conditions. This yields shorter and more understandable constraints and code that can be more easily tested. For example:

- `Tournament.overlaps(t:Tournament)` to check if two tournament overlap in time.

Avoid constraints that involve many association traversals. This often increases the coupling among sets of unrelated classes, which is not desirable. For example:

- When specifying that the duration of a match should be under a maximum limit set by the game, we could be tempted to write `match.tournament.league.game.maxMatchDuration`. Instead, consider adding a `Match.getGame()` method, which would simplify the expression to `match.getGame().maxMatchDuration`.

Figure 9-12 Heuristics for writing readable constraints.

일기 쉬운 제약 조건을 작성하기 위한 경험법칙

9.4.3의 문맥에 주의를 준다. 연산에만 국한하거나 객체가 특정한 상태에 있는 경우에만 동작하는 제약 조건을 논리적 조건과 사후 조건으로 더 잘 표현한다. 예를 들어:

- "이론 선수들은 다른 1-계급 수호를 갖는다"는 불변 조건이다.
- "player는 한 번에 하나의 대회만 지원할 수 있다"는 `TournamentForm.applyForTournament()` 연산의 사후 조건이다.

9.4.3에 대한 특별한 경우를 식별한다. 0과 null(Null) 값, 특정 범위 내에서 고유한 값을 갖는 속성과 다른 속성에 의존하는 속성값은 종종 오해와 오류의 근원이다. 예를 들어:

- 줄거리는 적어도 한 명의 선수를 포함한다.
- 종료된 경기는 정확히 한 명의 승자를 갖는다.

논리적 조건과 특별한 경우를 식별한다. 특별히 다정식 하나만으로 지정될 수 없는 특별한 경우를 식별한다. 예를 들어:

- `tournament.players`는 `tournament.league.players`의 부분집합이다.

연산을 자석의 순서를 확인한다. 예를 들어, 9.4.3절에 `TournamentControl`를 참조한다.

특별한 조건을 생산하는 도우미 메소드를 사용한다. 이 방법은 읽고 더욱 이해하기 쉬운 제약 조건을 더 쉽게 메소드로 수 있는 코드를 생성한다. 예를 들어:

- 두 토너먼트가 시간적으로 중복되는지를 체크하기 위한 `Tournament.overlaps(t: Tournament)`

불변 연산을 저장하는 제약 조건을 따른다. 이것은 종종 연관되지 않은 클래스들 간의 집합사이의 결합을 증가시킨다. 예를 들어:

- 경기 시간이 게임에서 설정된 최대 한계보다 작아야 한다는 것을 지정할 때, `match.tournament.league.game.maxMatchDuration`을 작성하는 유틸리티 있을 수 있다. 대신에 `match.getGame().maxMatchDuration`에 표현식을 간단히 하는 `match.getGame()` 메소드를 추가할 것을 고려한다.

9.4.5 Inheriting Contracts

- contract inheritance.
 - the class user expects that a contract that holds for the superclass still holds for the subclass.
- Contracts are inherited in the following manner:
 - Preconditions. A method of subclass is allowed **to weaken** the preconditions of the method it overrides.
 - Postconditions. Methods must ensure **the same postconditions** as their ancestors or **stricter ones**
 - Invariants. A subclass must **respect all** invariants of its superclasses

