Object Design

- Object design is the process of adding details to the requirements analysis and making implementation decisions.
- The object designer must choose among different ways to implement the analysis model with the goal to minimize execution time, memory and other measures of cost.
  - Requirements Analysis: The functional model and the dynamic model deliver operations for the object model.
  - Object Design: We decide on where to put these operations in the object model.
- Object design serves as the basis of implementation.

Object Design: Closing the Gap

Developers play different Roles during Object Design

Specifying Interfaces

- Requirements analysis activities
  - Identifying attributes and operations without specifying their types or their parameters.
- Object design: Three activities
  1. Add visibility information.
  2. Add type signature information.
  3. Add contracts.
1. Add Visibility Information

UML defines three levels of visibility:

 dhe Private (Class implementor):
* A private attribute can be accessed only by the class in which it is defined.
* A private operation can be invoked only by the class in which it is defined.
* Private attributes and operations cannot be accessed by subclasses or other classes.

dhe Protected (Class extender):
* A protected attribute or operation can be accessed by the class in which it is defined and on any descendent of the class.

dhe Public (Class user):
* A public attribute or operation can be accessed by any class.

Implementation of UML Visibility in Java

Tournament

```
maxNumPlayers: int
+ getMaxNumPlayers(): int
+ getPlayers(): List
+ isPlayerAccepted(p: Player): boolean
```

```
public class Tournament {
  private int maxNumPlayers;
  public Tournament(League l, int maxNumPlayers) {
    this.maxNumPlayers = maxNumPlayers;
  }
  public int getMaxNumPlayers() {…};
  public List getPlayers() {…};
  public void acceptPlayer(Player p) {…};
  public void removePlayer(Player p) {…};
  public boolean isPlayerAccepted(Player p) {…};
```

2. Add Type Signature Information

```
Hashtable
+ put(key: Object, entry: Object)
+ get(key: Object): Object
+ remove(key: Object)
+ containsKey(key: Object): boolean
+ size(): int
```

```
Hashtable
+ put()
+ get()
+ remove()
+ containsKey()
+ size()
```

```
Attributes and operations without type information are acceptable during analysis
```

Team Activity: Visibility and Signatures

**Description:** Select one of your classes. Complete the visibility and signature for that class.

**Process:**
- Work in teams
- You have about 10 minutes.

Information Hiding Heuristics

- Carefully define the public interface for classes as well as subsystems (façade)
- Always apply the “Need to know” principle.
  - Only if somebody needs to access the information, make it publicly possible, but then only through well defined channels, so you always know the access.
- The fewer an operation knows
  - the less likely it will be affected by any changes
  - the easier the class can be changed
- Trade-off: Information hiding vs efficiency
  - Accessing a private attribute might be too slow (for example in real-time systems or games)

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 system and external world 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.
3. Add Contracts

Contracts on a class enable caller and callee to share the same assumptions about the class.

Contracts include three types of constraints:

- **Invariant:** A predicate that is always true for all instances of a class. Invariants are constraints associated with classes or interfaces.

- **Precondition:** Preconditions are predicates associated with a specific operation and must be true before the operation is invoked. Preconditions are used to specify constraints that a caller must meet before calling an operation.

- **Postcondition:** Postconditions are predicates associated with a specific operation and must be true after an operation is invoked. Postconditions are used to specify constraints that the object must ensure after the invocation of the operation.

### Expressing constraints in UML Models

- OCL (Object Constraint Language)
  - OCL allows constraints to be formally specified on single model elements or groups of model elements
  - A constraint is expressed as an OCL expression returning the value true or false. OCL is not a procedural language (cannot constrain control flow).

- OCL expressions for Hashtable operation put():
  - **Invariant:** context Hashtable inv: numElements >= 0
  - **Precondition:** context Hashtable::put(key, entry) pre: !containsKey(key)
  - **Postcondition:** context Hashtable::put(key, entry) post: containsKey(key) and get(key) = entry

### Contract for acceptPlayer in Tournament

```
context Tournament::acceptPlayer(p) pre:
  not isPlayerAccepted(p)
  getNumPlayers() < getMaxNumPlayers()
context Tournament::acceptPlayer(p) post:
  isPlayerAccepted(p)
  getNumPlayers() = @pre.getNumPlayers() + 1
```

### Contract for removePlayer in Tournament

```
context Tournament::removePlayer(p) pre:
  isPlayerAccepted(p)
context Tournament::removePlayer(p) post:
  not isPlayerAccepted(p)
  getNumPlayers() = @pre.getNumPlayers() - 1
```

### Annotation of Tournament class

```java
public class Tournament {
  /** The maximum number of players is positive at all times.
   * @invariant maxNumPlayers > 0 */
  private int maxNumPlayers;

  /** The players List contains references to Players who are
   * are registered with the Tournament. */
  private List players;

  /** Returns the current number of players in the tournament. */
  public int getNumPlayers() {…}

  /** Returns the maximum number of players in the tournament. */
  public int getMaxNumPlayers() {…}

  /** The acceptPlayer() operation assumes that the specified
   * player has not been accepted and in the Tournament yet.
   * @pre !isPlayerAccepted(p)
   * @pre getNumPlayers() < getMaxNumPlayers()
   * @post isPlayerAccepted(p)
   * @post getNumPlayers() = @pre.getNumPlayers() + 1 */
  public void acceptPlayer(Player p) {…}

  /** The removePlayer() operation assumes that the specified
   * player is currently in the Tournament.
   * @pre isPlayerAccepted(p)
   * @post !isPlayerAccepted(p)
   * @post getNumPlayers() = @pre.getNumPlayers() - 1 */
  public void removePlayer(Player p) {…}
}
```
**Team Activity: Contracts**

- **Description:** Select one of your classes. Complete the contracts for that class.
- **Process:**
  - Work in teams
  - You have about 10 minutes.

**Constraints can involve more than one class**

How do we specify constraints on more than one class?

---

**3 Types of Navigation through a Class Diagram**

1. Local attribute
2. Directly related class
3. Indirectly related class

Any OCL constraint for any class diagram can be built using only a combination of these three navigation types!

**Model Refinement with 3 additional Constraints**

- A Tournament’s planned duration must be under one week.
- Players can be accepted in a Tournament only if they are already registered with the corresponding League.
- The number of active Players in a League are those that have taken part in at least one Tournament of the League.
- To better understand these constraints we instantiate the class diagram for a specific group of instances
  - 2 Leagues, 2 Tournaments and 5 Players

---

**ARENA Example: League, Tournament and Player**

**Instance Diagram: 2 Leagues, 2 Tournaments, and 5 Players**
Specifying the Model Constraints

Local attribute navigation
context: Tournament
inv: end - start <= Calendar.WEEK

Directly related class navigation
context: Tournament
:: acceptPlayer(p)
pre: league.players->includes(p)

Indirectly related class navigation
context: League
:: getActivePlayers
post: result = tournaments.players->asSet

OCL supports Quantification

- 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))

Summary
- There are three different roles for developers during object design
  - Class user, class implementor and class extender
- During object design - and only during object design - we specify visibility rules
- Constraints are boolean expressions on model elements
- Contracts are constraints on a class enable class users, implementors and extenders to share the same assumption about the class ("Design by contract")
- OCL is a language that allows us to express constraints on UML models
- Complicated constraints involving more than one class, attribute or operation can be expressed with 3 basic navigation types.