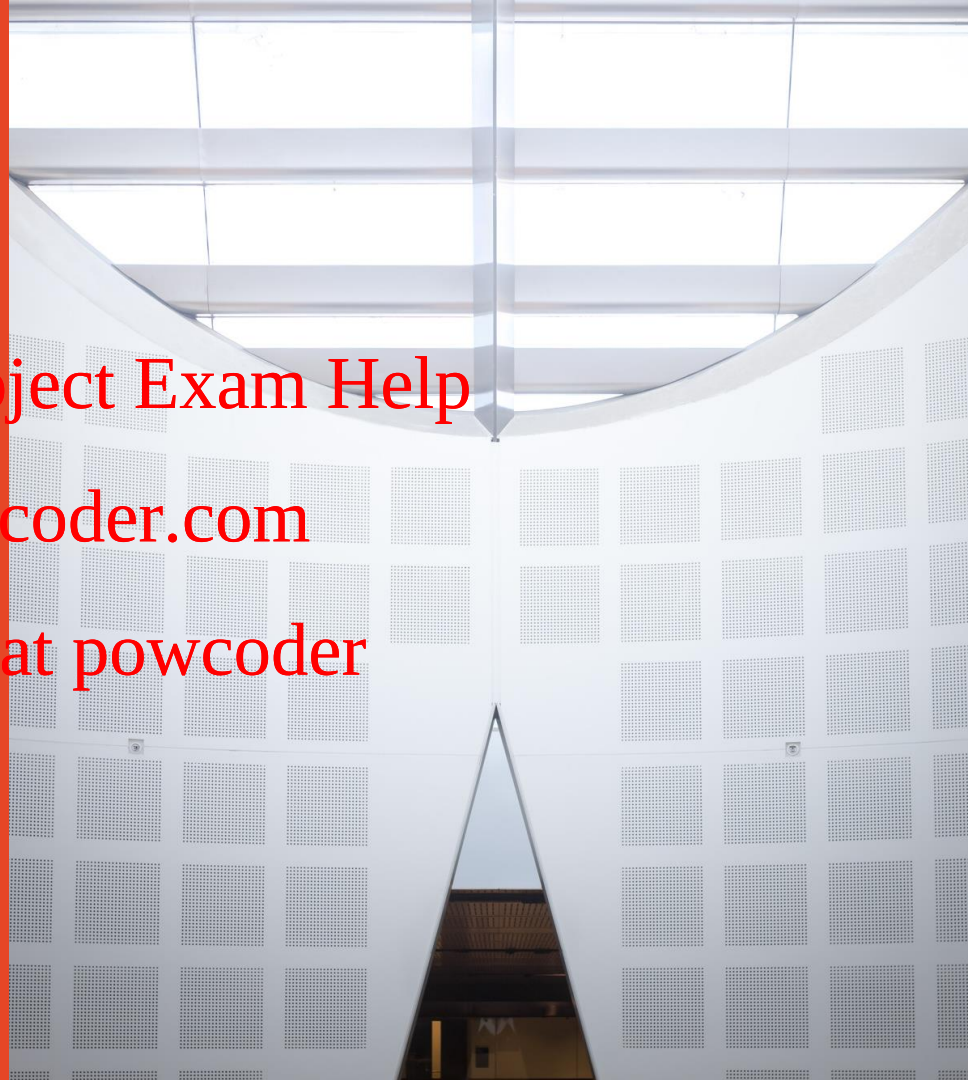
Software Design and Construction 2
SOFT3202 / COMP9202

Software Verification
Theory and Examples

Dr. Basem Suleiman

School of Information Technologies

THE UNIVERSITY OF SYDNEY



Copyright Warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Agenda

- Software Verification Theory
- Design by Contract
  - Pre-conditions and post-conditions
  - Class invariants

Software Validation and Verification

Theory

THE UNIVERSITY OF SYDNEY

Software Testing – Revisit

- Software process to
  - Demonstrate that software meets its requirements (*validation testing*)
  - Find incorrect or undesired behavior caused by defects/bugs (defect testing)
    - E.g., System crashes, incorrect computations, unnecessary interactions and data corruptions
- Part of software verification and validation (V&V) process

Software Verification and Validation

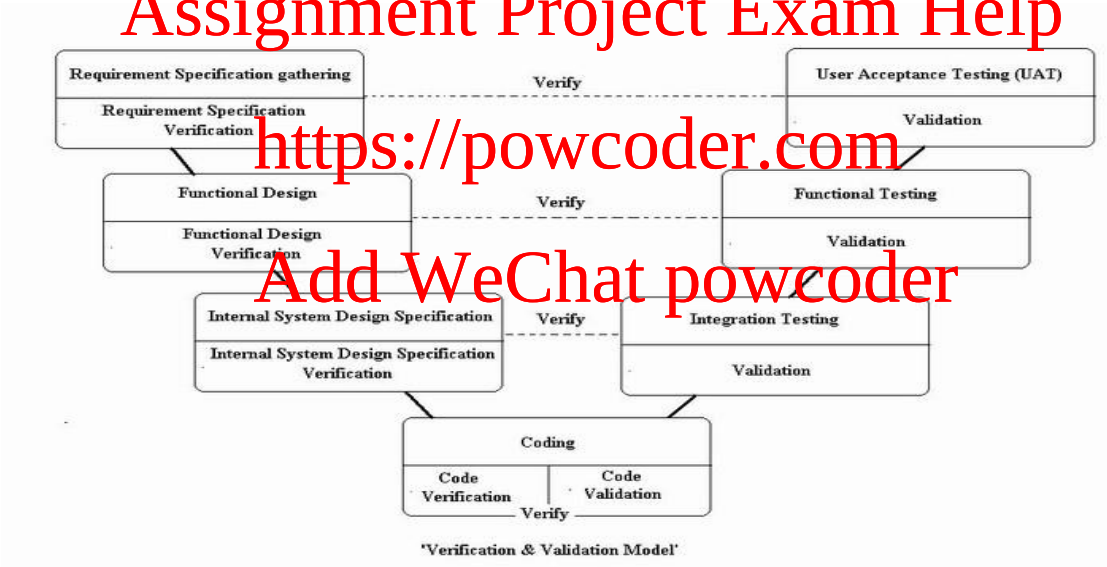
- Software testing is part of software Verification and Validation (V&V)
- The goal of V&V is to establish confidence that the software is "*fit for purpose*"
- Software Validation
  - Are we building the right product?
  - Ensures that the software meets customer expectations/needs
- Software Verification
  - Are we building the product right?
  - Ensures that the software meets its stated functional and non-functional requirements

Software V&V

- Software Verification
  - Concerned with the software requirements/specifications
  - Can be ambiguous
- Software Validation
  - Customer/end user's expectation
  - Disambiguate the requirements

V-Model

- Link each phase of the SDLC with its associated testing phase
- Each verification stage relates to a validation stage



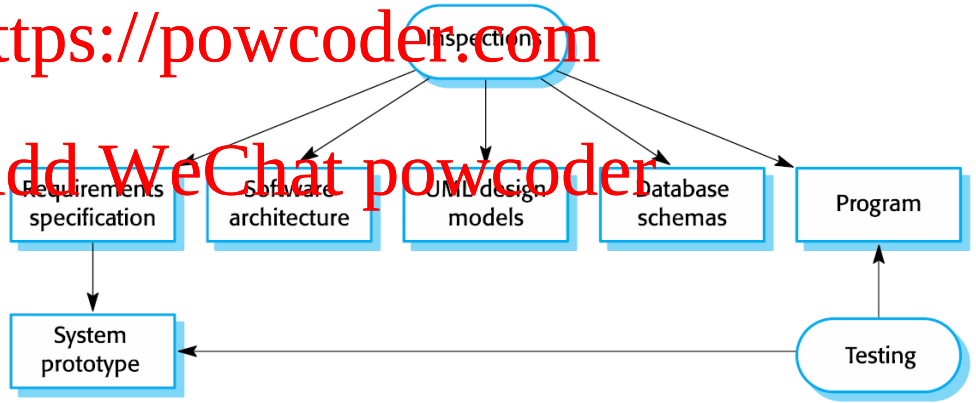
https://www.buzzle.com/editorials/4-5-2005-68117.asp

Software Verification

- Artifact/specification verification
  - Review the resulted software artifacts to verify that specifications are met
- Check the output of each software development phase against the input specification (specs.)
  - Design specifications against requirement specifications
    - Detailed design correctly implement the requirements (F & NF)
  - Construction (code) against the design specs.
    - Source code/user interfaces correctly implements the design specs.
- Include inspections, reviews, walkthrough

Static Verification

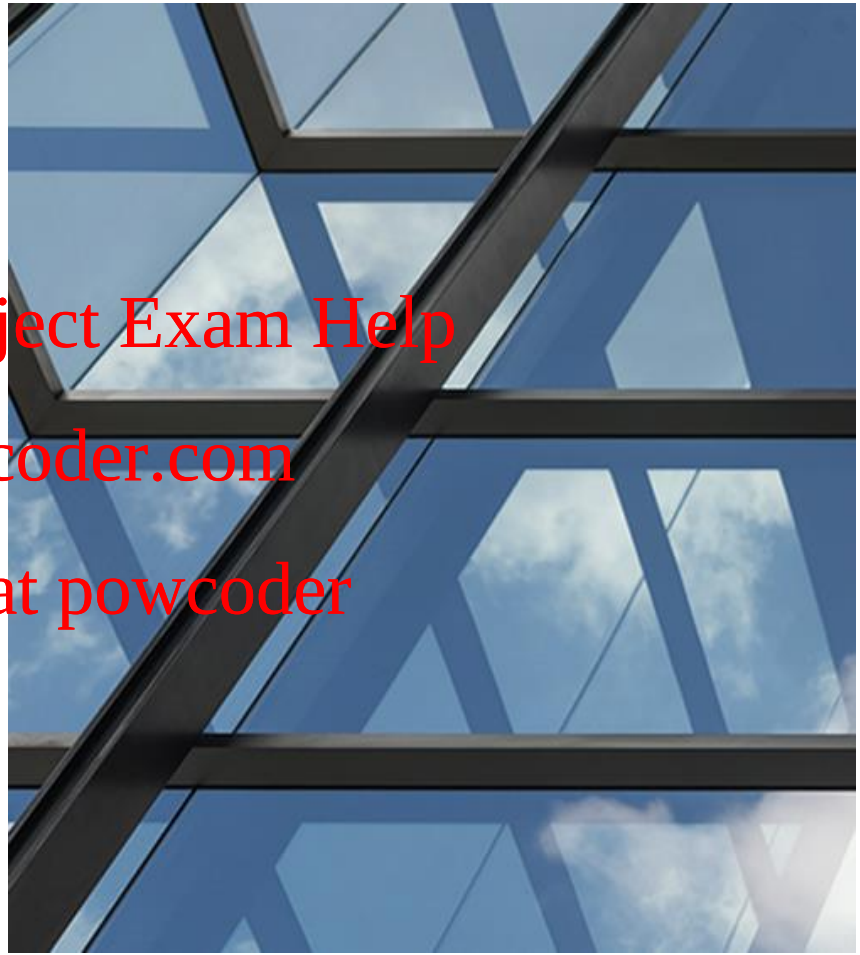
- Static Verification
  - Static system analysis to discover problems
  - May be applied to requirements, design/models, configuration and test data
- Reviews
  - Walk through
  - Code inspection



Software Validation

- Artifact/specification validation
  - Validate the requirements from the end user perspective
- Check that the needs of all stakeholders (users, managers, investors) are met
  - Validate the output of main development stages from the stakeholder's point of view
    - Validate if the requirements represent the will and goals of the stakeholders
- Include Unit testing, integration/functional testing, and user acceptance testing

Design by Contract



THE UNIVERSITY OF SYDNEY

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

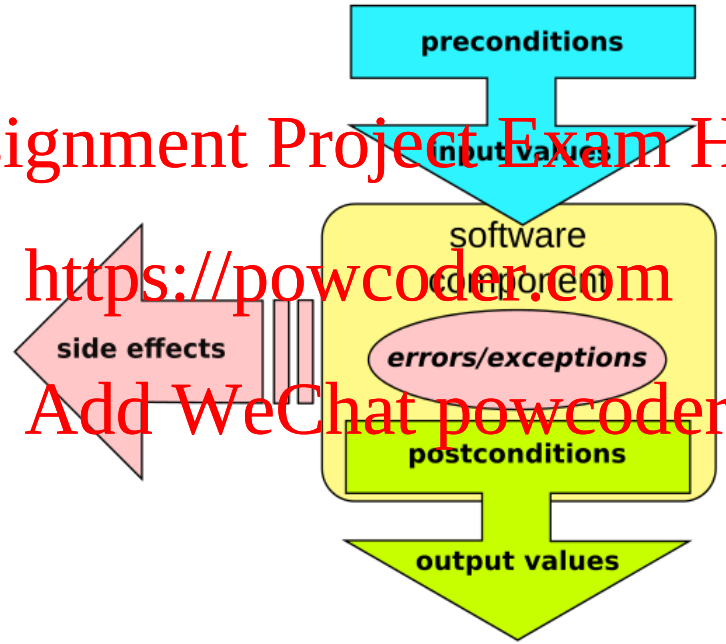
Object-Oriented Contract

- Specifies the expected services that can be provided if certain conditions are satisfied
- Services = "*Obligations*" and conditions = "*Rights*"
- Breach of a contract results in generating an exception

Design by Contract (DbC)

- A software design approach for program correctness
- Known as contract programing, programming by contract, design-by-contract programming
- Definition of formal, precise and verifiable interface specification for software components
  - Pre-conditions, postconditions and invariants (contract)

Design by Contract (DbC)



By Fabuio [CC0], from Wikimedia Commons, https://commons.wikimedia.org/wiki/File:Design_by_contract.svg

Contracts in OO Design

Object-oriented contract

- Describes the services that are provided by an object if certain conditions are fulfilled
  - services = "obligations", conditions = "rights"
- An OO-contract for each service specifically describes:
  - The *conditions* under which the service will be provided
  - A *specification* of the result of the service that is provided
- The remedy for breach of an OO-contract is the generation of an exception

Object-Oriented Contract –

- Examples:
  - A letter posted before 18:00 will be delivered on the next working day to any address within Australia
  - For the price of $7 a letter with a maximum weight of 80 grams will be delivered anywhere in Australia within 4 hours of pickup

- Exercise: identify the conditions/rights and obligations/services of the above delivery services.

Modeling Constraints with Contracts

Modeling Constraints with Contracts

- Example of constraints in Arena:
  - An already registered player cannot be registered again
  - The number of players in a tournament should not be more than maxNumPlayers
  - One can only remove players that have been registered

- We model them with contracts
- Contracts can be written in OCL

Modeling OO-Contracts – Formal Specifcation

- 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

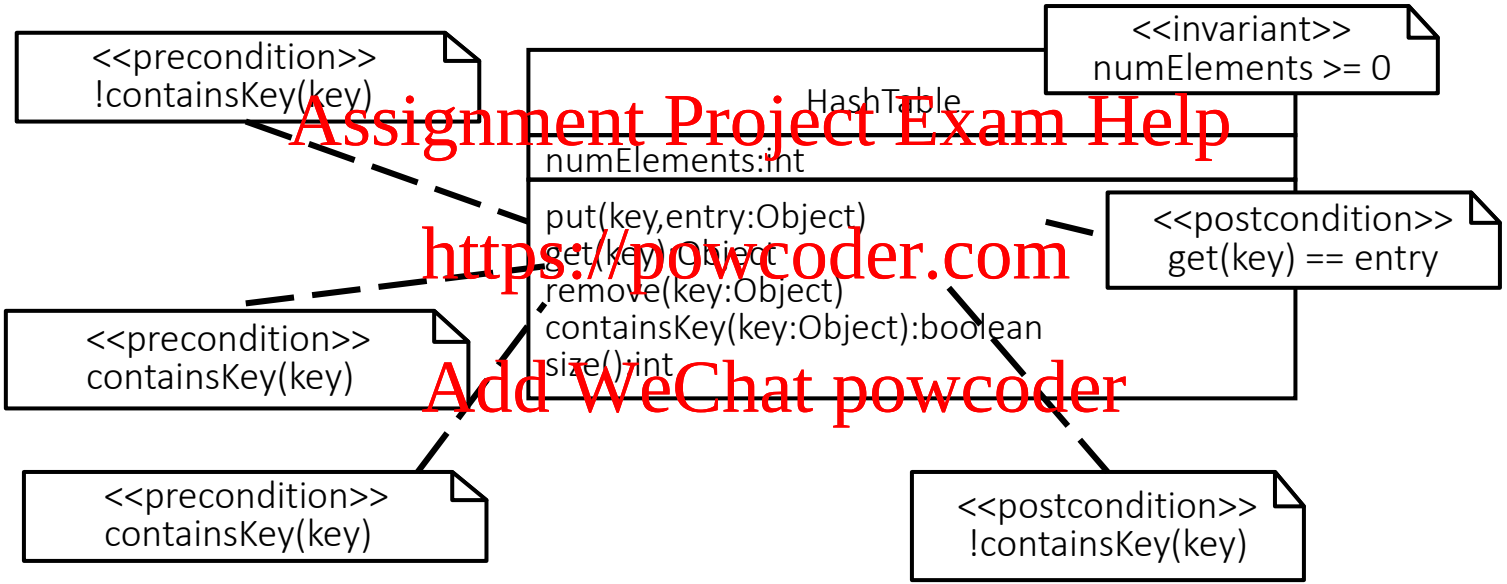
Contracts and Formal Specification

- A Contract share same assumptions about the class
- Three 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.

Formal Specification

- A contract is called a formal specification, if the invariants, rights and obligations in the contract are unambiguous

Expressing Constraints in UML Models



Use Contracts in Requirements Analysis

- Many constraints represent domain-level information
- Why not use them in requirements analysis?
  - Increase requirements precision
  - Yield more questions for the end user
  - Clarify the relationships among several objects
- Constraints are sometimes used during requirements analysis, however there are trade-offs

Object Constraint Language (OCL)

Object Constraint Language (OCL)

- Formal language for expressing constraints over a set of objects and their attributes

- Part of the UML standard

- Used to write constraints that cannot otherwise be expressed in a diagram

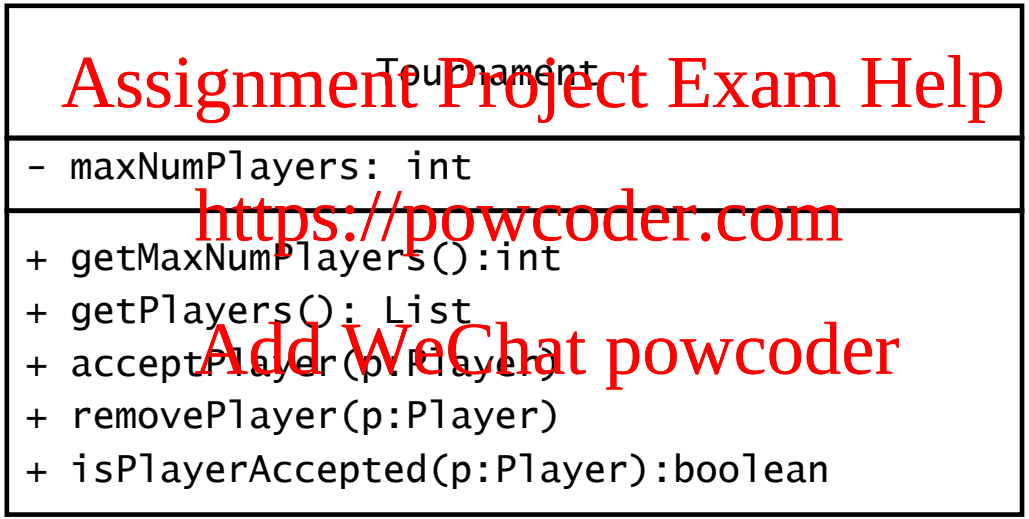
- Declarative
  - No side effects
  - No control flow

- Based on Sets and Multi Sets

OCL Basic Concepts

- OCL expressions
  - Return **True** or **False**
  - Are evaluated in a specified *context*
  - All *constraints* apply to *all instances*

Example – Tournament Class



OCL Simple Predicates

"The maximum number of players in any tournament should be a positive number."


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


Notes:

- OCL uses the same dot notation as Java

OCL Preconditions – Examples

"The *acceptPlayer(p)* operation can only be invoked if player *p* has not yet been accepted in the tournament."


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


Questions:

- What is the context the pre-condtion?
- What is "isPlayerAccepted(p)"?

OCL Postconditions – Example

"The number of accepted player in a tournament increases by one after the completion of acceptPlayer()"


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


Notes:

- self@pre: the state of the tournament before the invocation of the operation
- self: denotes the state of the tournament after the completion of the operation

OCL Contract for acceptPlayer() in Tournament

```
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() = @pre.getNumPlayers() + 1
```

OCL Contract for removePlayer() in Tournament

```
context Tournament::removePlayer(p) pre:
    isPlayerAccepted(p)

context Tournament::removePlayer(p) post:
    not isPlayerAccepted(p)

context Tournament::removePlayer(p) post:
    getNumPlayers() = @pre.getNumPlayers() - 1
```

Java Implementation of Tournament class (Contract as a set of JavaDoc comments)

```
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() {…}
```

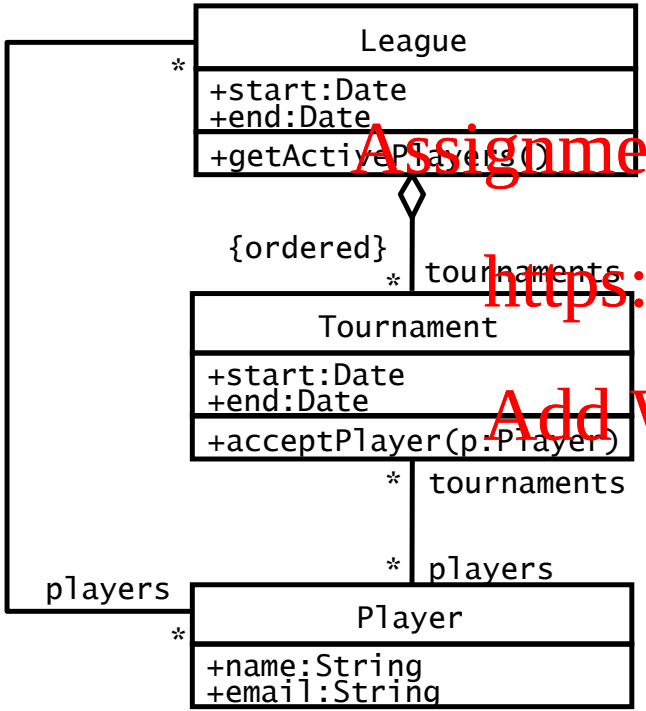
```
/** The acceptPlayer() operation
 * assumes that the specified
 * player has not been accepted
 * in the Tournament yet.
 * @pre !isPlayerAccepted(p)
 * @pre getNumPlayers()<maxNumPlayers
 * @post isPlayerAccepted(p)
 * @post getNumPlayers() =
 *      @pre.getNumPlayers() + 1
 */
public void acceptPlayer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

}
```

Constraints can involve more than one class

How do we specify constraints on on a group of classes?

Starting from a specific class in the UML class diagram, navigate the associations in the class diagram to refer to the other classes and their properties (attributes and operations).

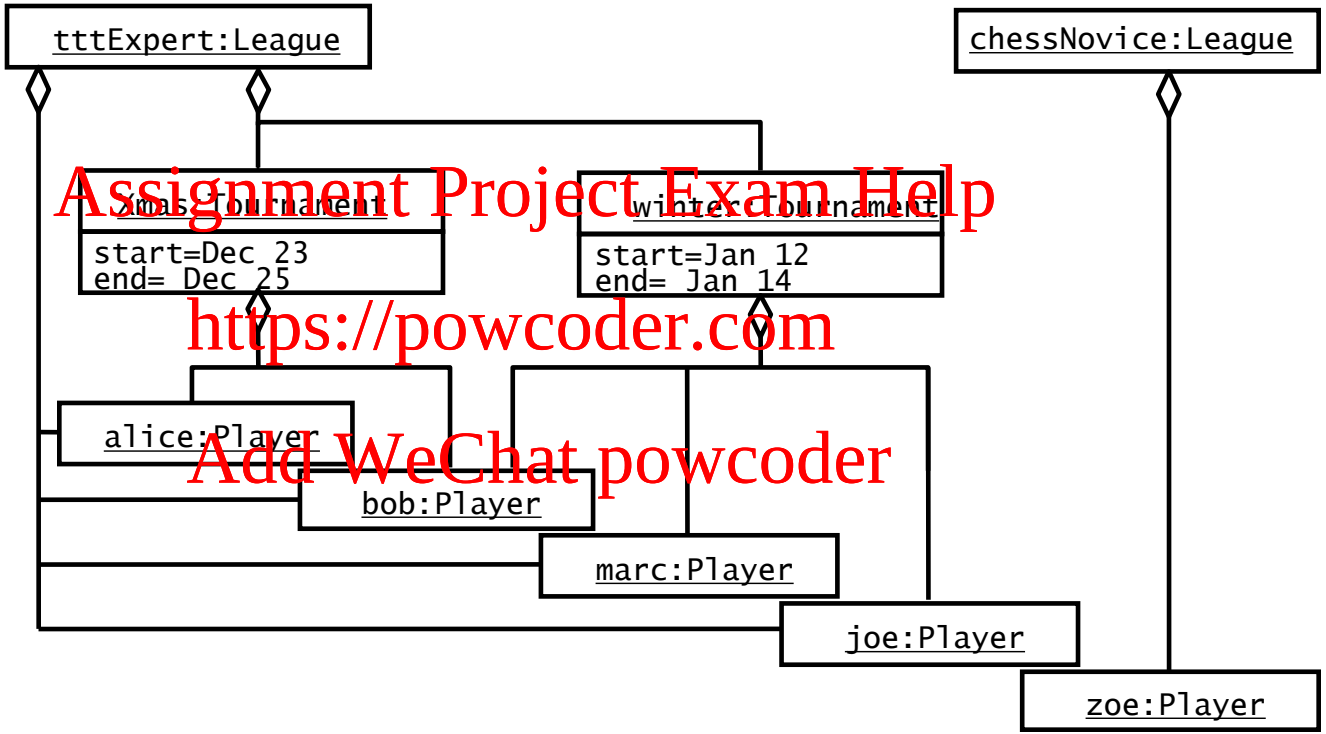
Example from ARENA: League, Tournament and Player



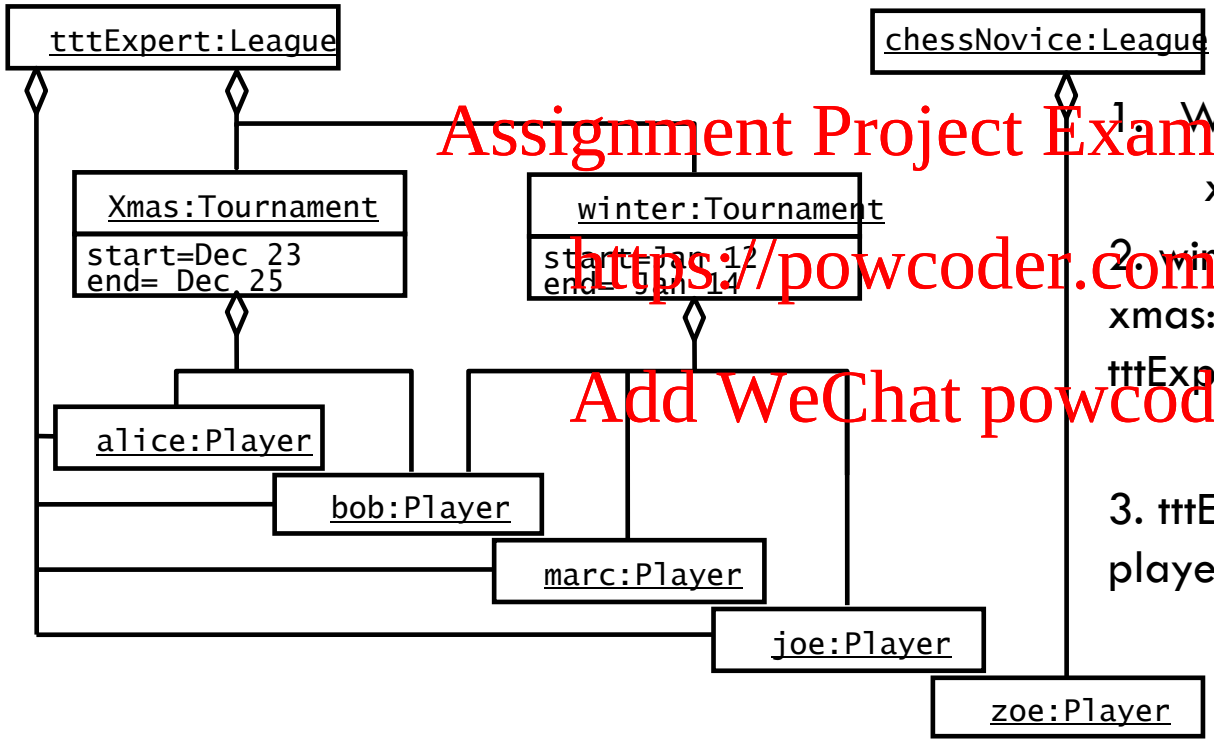
Constraints:

1. A Tournament's planned duration must be under one week.
2. Players can be accepted in a Tournament only if they are already registered with the corresponding League.
3. The number of active Players in a League are those that have taken part in at least one Tournament of the League.

Instance Diagram: 2 Leagues 5 players

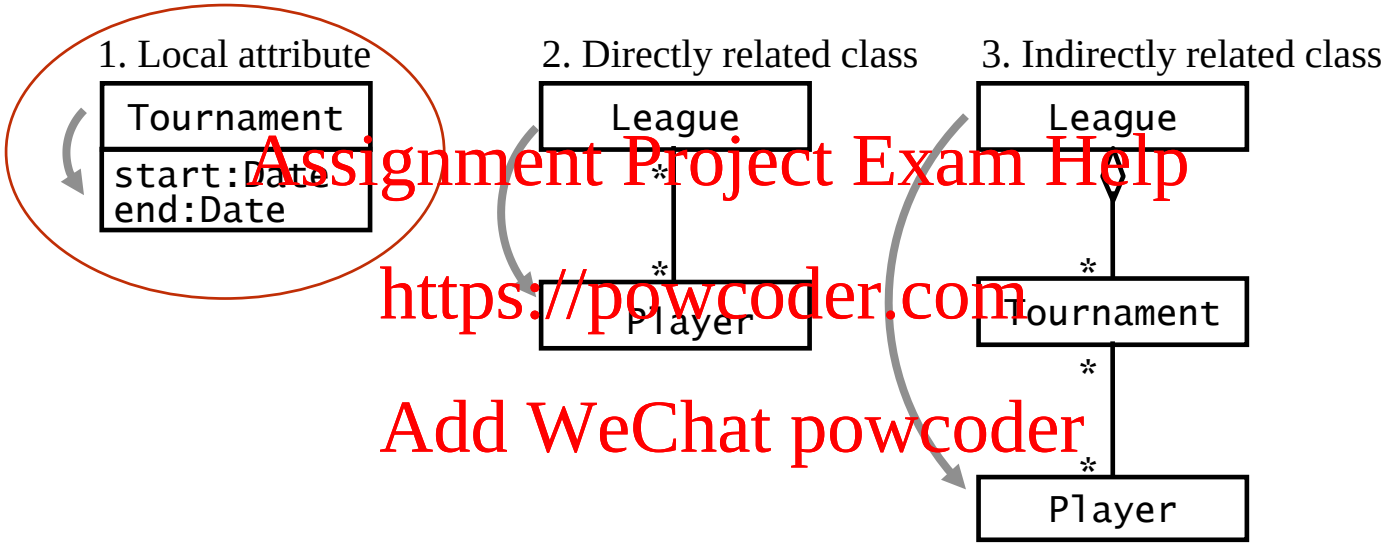


Instance Diagram: Review Constraints



1. Winter:Tournament lasts 2 days
   xmas:Tournament lasts 3 days
2. winter:tournament and xmas:tournament associated with tttExpert:League
3. tttExpertPlayer has 4 active players, ChessNovice:Legue has none

3 Types of Navigation through a Class Diagram



Any constraint for an arbitrary UML class diagram can be specified using only a combination of these 3 navigation types!

Local Attribute

context Tournament inv: self.end - self.start < 7

Specifying the Model Constraints in OCL

Local attribute navigation

```
context Tournament inv:
    end - start < 7
```

Directly related class navigation

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

OCL Sets, Bags and Sequences

- Sets, Bags and Sequences are predefined in OCL
  - Subtypes of Collection

- OCL offers a large number of predefined operations on collections. All of the form:


```
collection->operation(arguments)
```

OCL-Collection

- The **OCL-Type** Collection is the generic superclass of a collection of objects of Type T
- Subclasses of Collection are
  - Set: Set in the mathematical sense. Every element can appear only once
  - Bag: A collection, in which elements can appear more than once (also called multiset)
  - Sequence: A multiset, in which the elements are ordered
- Example for Collections:
  - Set(Integer): a set of integer numbers
  - Bag(Person): a multiset of persons
  - Sequence(Customer): a sequence of customers

OCL-Operations for OCL-Collections (1)

`size: Integer`
Number of elements in the collection

`includes(o:OclAny): Boolean`
True, if the element o is in the collection

`count(o:OclAny): Integer`
Counts how many times an element is contained in the collection

`isEmpty: Boolean`
True, if the collection is empty

`notEmpty: Boolean`
True, if the collection is not empty

The OCL-Type **OclAny** is the most general OCL-Type

OCL-Operations for OCL-Collections(2)

`union(c1:Collection)`

Union with collection **`c1`**

`intersection(c2:Collection)`

Intersection with Collection **`c2`** (contains only elements, which appear in the collection as well as in collection **`c2`** auftreten)

`including(o:OclAny)`

Collection containing all elements of the Collection and element **`o`**

`select(expr:OclExpression)`

Subset of all elements of the collection, for which the OCL-expression **`expr`** is true

Evaluating OCL Expressions

The value of an OCL expression is an object or a collection of objects.

- Multiplicity of the association-end is 1
  - The value of the OCL expression is a single object
- Multiplicity is 0..1
  - The result is an empty set if there is no object, otherwise a single object
- Multiplicity of the association-end is *
  - The result is a collection of objects
    - By default, the navigation result is a Set
    - When the association is {ordered}, the navigation results in a Sequence
    - Multiple "1-Many" associations result in a Bag

OCL Quantifiers

forAll

- *forAll (variable|expression)* is True if expression is True for all elements in the collection

exist

- *exists (variable|expression)* is True if there exists at least one element in the collection for which expression is True

OCL Quantifiers – forAll Example

Given the following OCL quantifier


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


Exercise: explain what this OCL attempts to verify.

OCL Quantifiers – Exists Example

Given the following OCL quantifier

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

References

- Ian Sommerville. 2016. Software Engineering (10th ed.) Global Edition. Pearson.
- Wikipedia, Software Verification and Validation, https://en.wikipedia.org/wiki/Software_verification_and_validation
- Object-Oriented Software Engineering: Using UML, Patterns, and Java, 3rd Edition, *Bernd Bruegge & Allen H. Dutoit, Pearson.*

W12 Tutorial: Practical Exercises

Design Pattern Assignment Demo

W12 Lecture: Specification Languages

THE UNIVERSITY OF SYDNEY

Specifying the Model Constraints: Using asSet

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
```