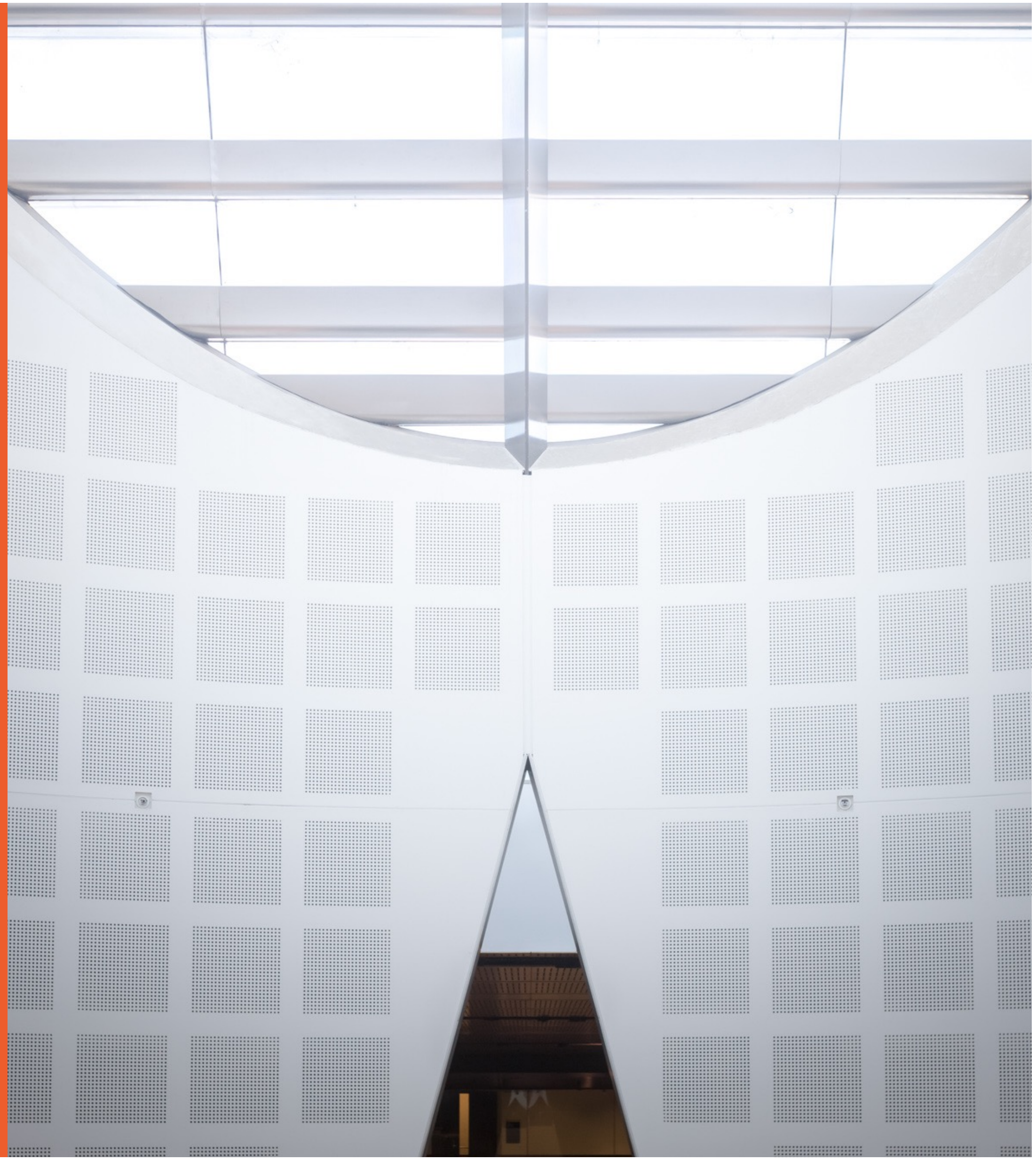
Software Design and Construction 2 SOFT3202 / COMP9202

Design Patterns & Software Verification

Prof Bernhard Scholz

School of Computer Science

THE UNIVERSITY OF SYDNEY



Agenda

- Design Verification
- OCL
- Alloy
- Test Driven Development

Design Verification

- Quest: how to test that the design is correct?

- For code: use unit/integration/system/acceptance tests and regression tests for changes.

- For design: nothing to execute; testing a design is ***hard***

- How can we detect design flaws early?
  - Manually, i.e., pencil and paper proofs, reflection, inspection, etc.
  - Automatically (=better)

Example: UML Diagrams

- UML Diagrams
  - structural/behavioral/interaction
- Example of Structural UML :

- Class Diagram sets in relation two sets
  - Team & Employee
  - Has a multiplicity constraint (composition)
- Verify by manually inspection the relationship:
  - Can there be a team with no employees?
  - Can an employee be in two teams?
  - Can there be an employee without a team?
- Check whether this model fits its purpose?

Formal Specification Languages

- Not all application semantics is expressible in UML diagrams
  - Limited expressiveness
  - Example
    - A team member must be older than 18 years, and requires an academic degree
    - Not expressible in UML

- Formal Design Specification Languages
  - Specify design formally
  - Specify constraints formally
  - Express requirements formally
  - Check whether design meets requirements
- **Formal** means requirements are defined using formal syntax and semantics

Adapted from

Formal Specification Languages

- **OCL** (from 1997; OMG 2012)
  - Textual language expressing constraints for a design
- **Alloy** (Jackson 2002)
  - Textual language for design that can be formally checked up to a certain problem size
- **Z** (Spivey 1992)
- **B** (Abrial 2009)
- **VDM** – Vienna Development Method (Björner and Jones 1978)

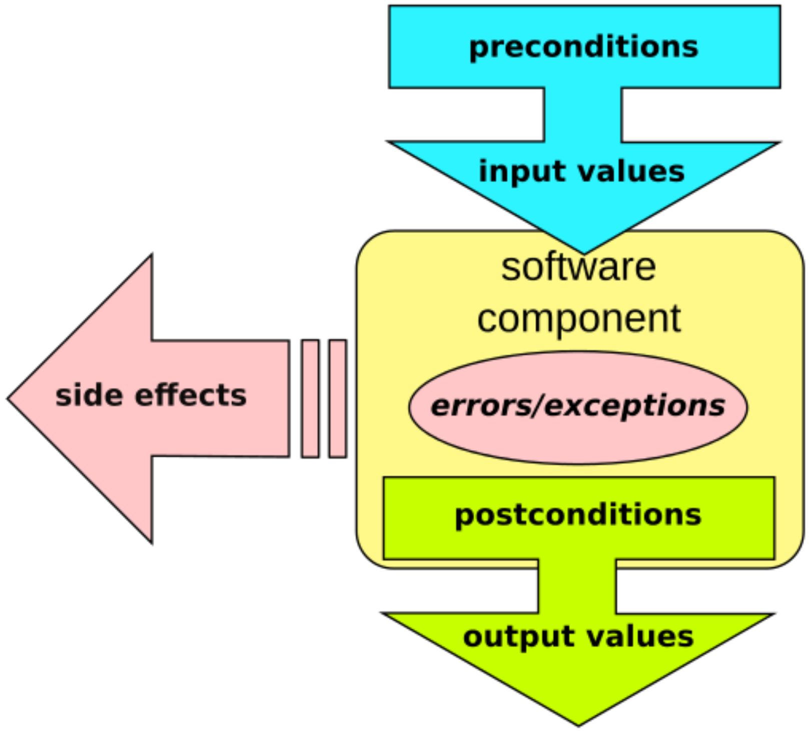
Design by Contract (DbC)

- 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

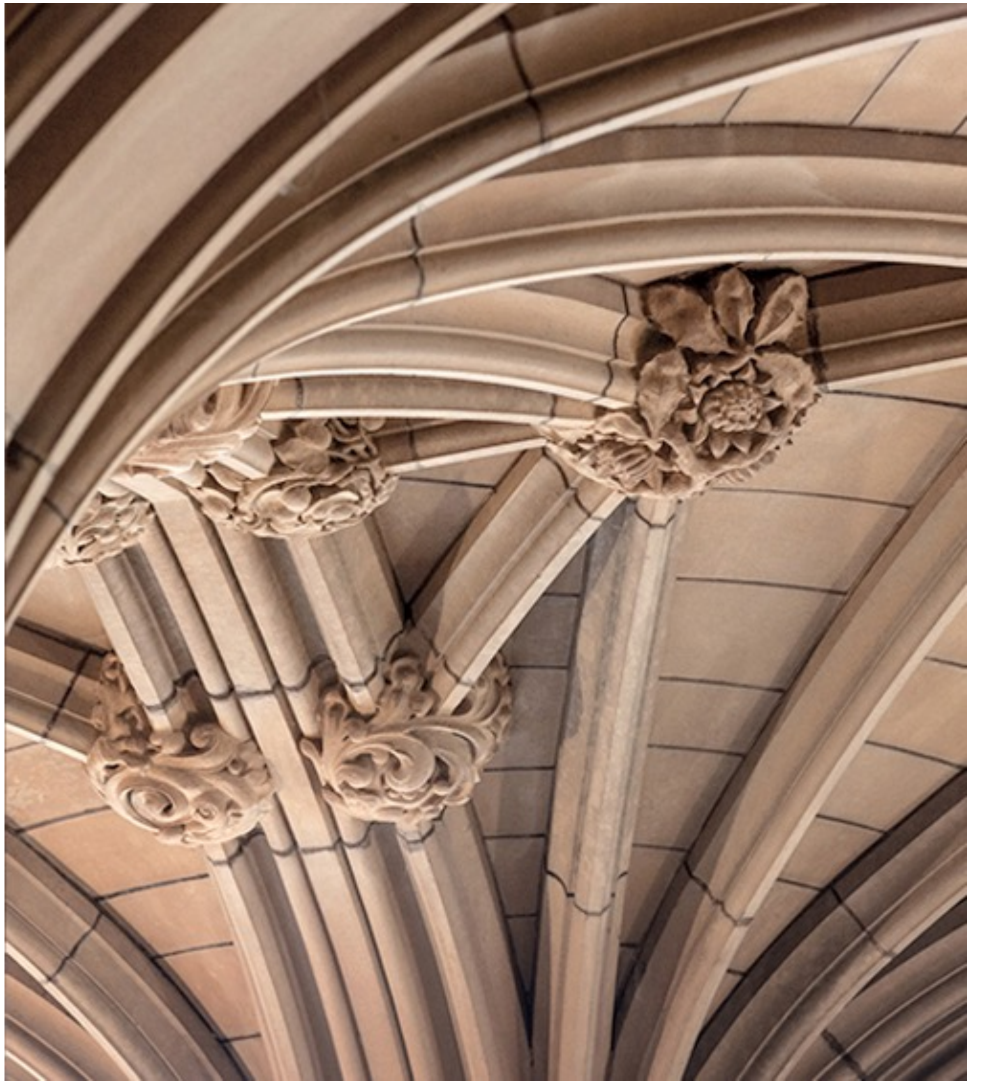
Execution of Contracts

- Where can contracts/requirements be checked?

- At design time:
  - Contracts/requirements are checked formally
  - Formal verification tools for running these checks
  - E.g., Alloy expresses design/constraints/requirements formally

- At runtime:
  - Pre/Post conditions and invariants are lowered to code-level in form of assertions.
  - Requirements are checked at runtime via testing (weak approach)

Object Constraint Language (OCL)



Object Constraint Language (OCL)

- UML diagrams not expressive enough

- Formal language for expressing constraints in SW designs

- Part of the UML standard

- Declarative
  - No side effects
  - No control flow

Example – Tournament Class

Tournament
- maxNumPlayers: int
+ getMaxNumPlayers():int + getPlayers(): List + acceptPlayer(p:Player) + removePlayer(p:Player) + isPlayerAccepted(p:Player):boolean

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)

```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() {…}
```

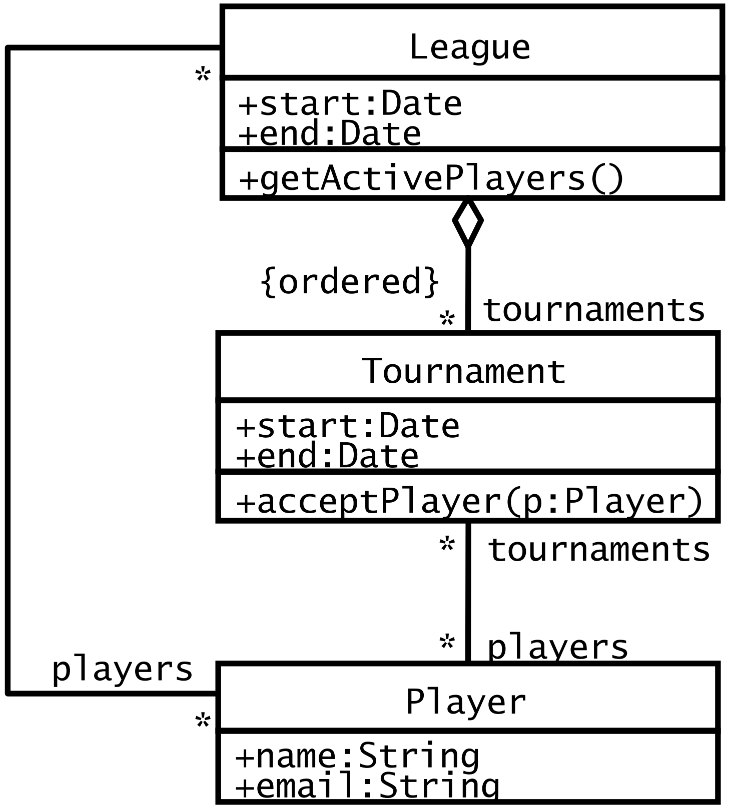
```java
/** The acceptPlayer() operation
 * assumes that the specified
 * player has not been accepted
 * in the Tournament yet.
 * @pre !isPlayerAccepted(p)
 * @pre getNumPlayers()<maxNumPlayers
 * @post isPlayerAccepted(p)
 * @post getNumPlayers() =
 *     @pre.getNumPlayers() + 1
 */
public void acceptPlayer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

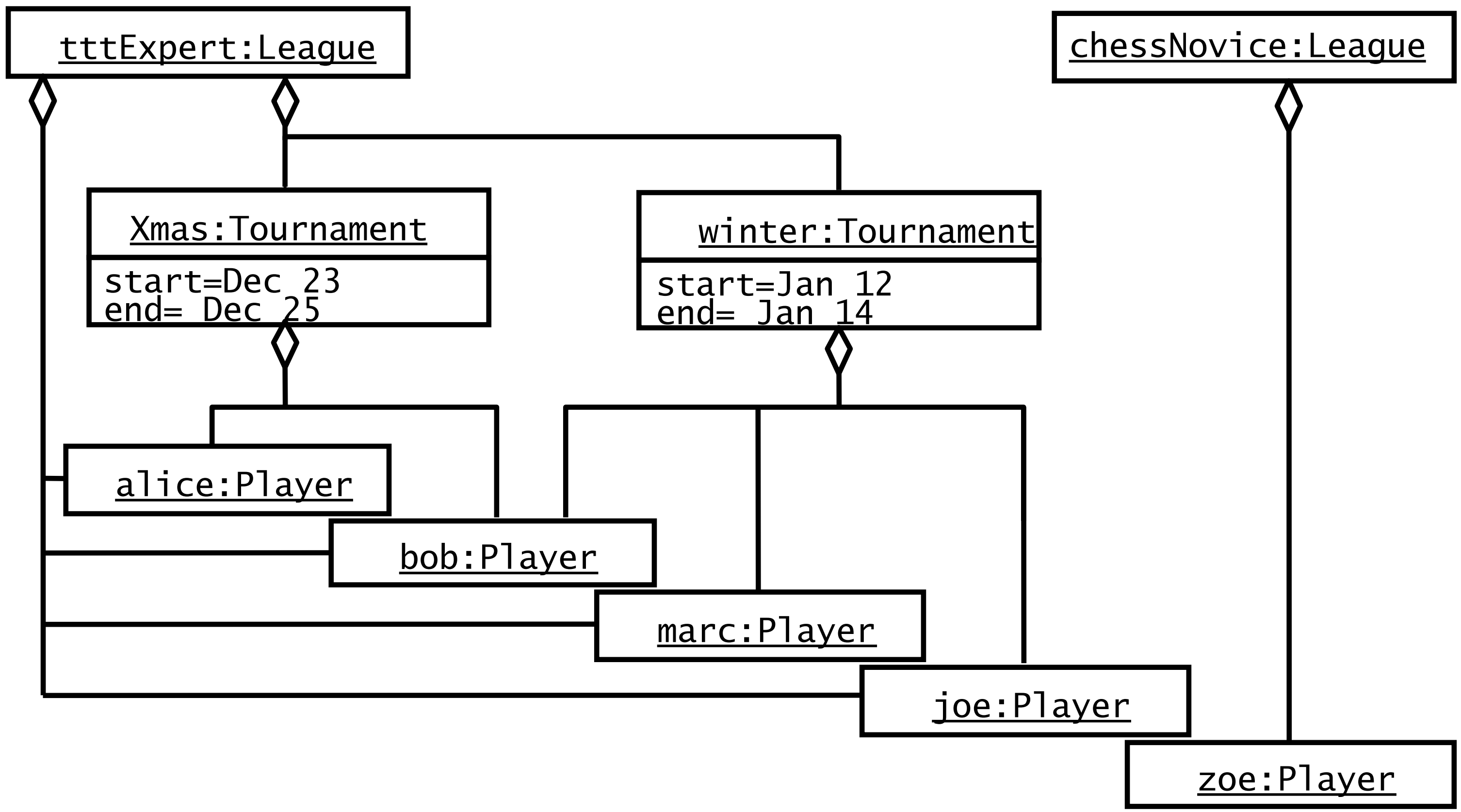
Example from ARENA: League, Tournament and Player



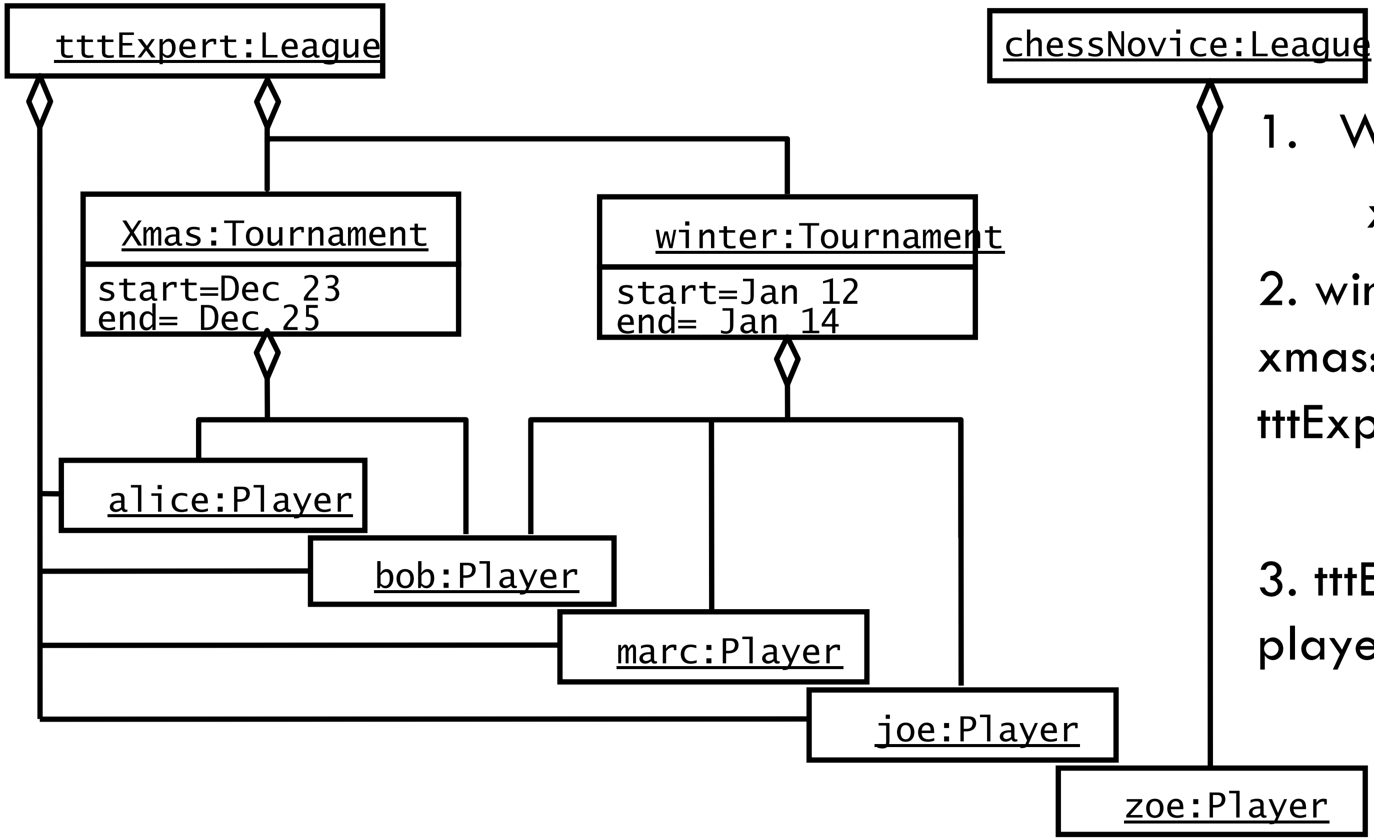
Constraints:

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



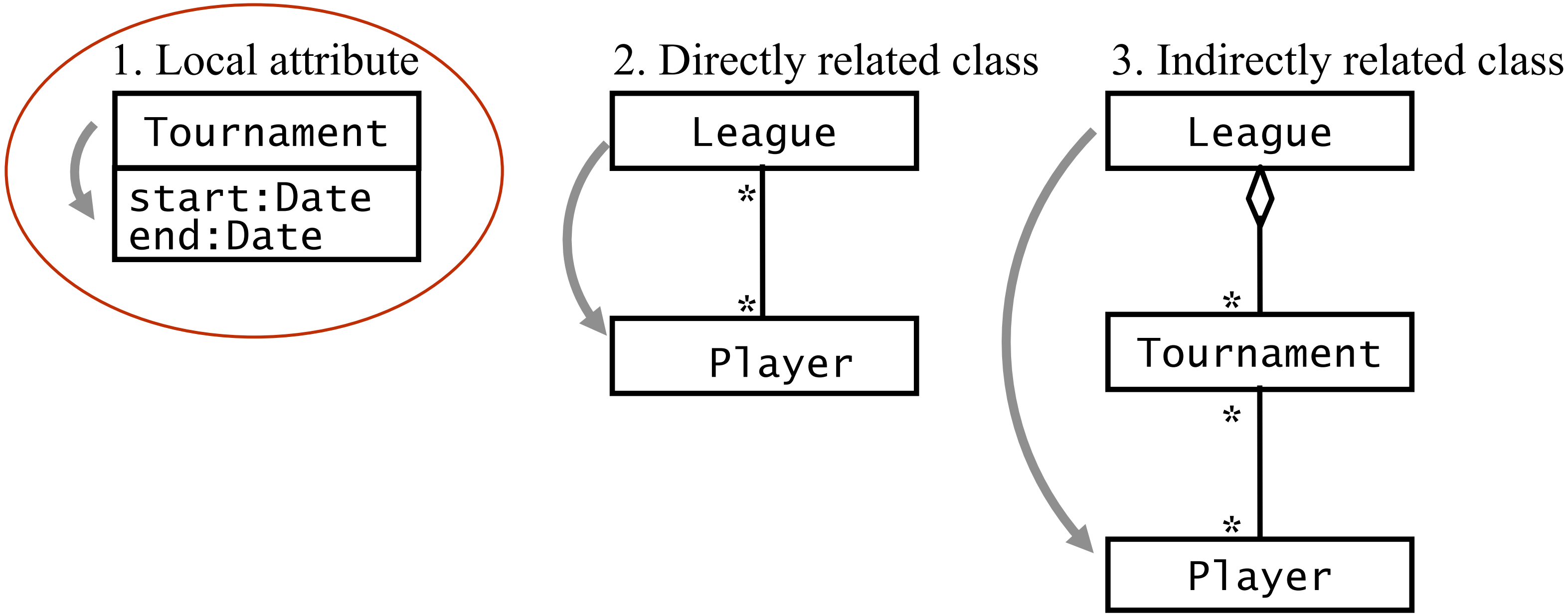
1. Winter:Tournament lasts 2 days

   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

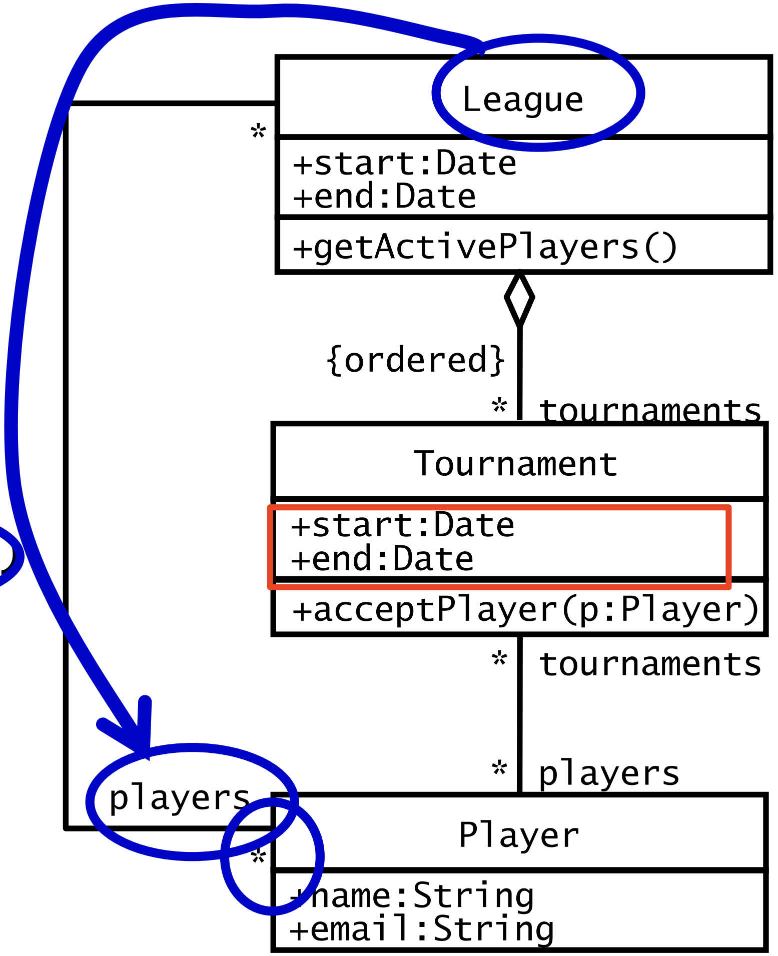
Specifying the Model Constraints in OCL

Local attribute navigation

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

Directly related class navigation

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



OCL 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

Example: OCL Quantifiers Example

- 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))
```

- 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 Summary

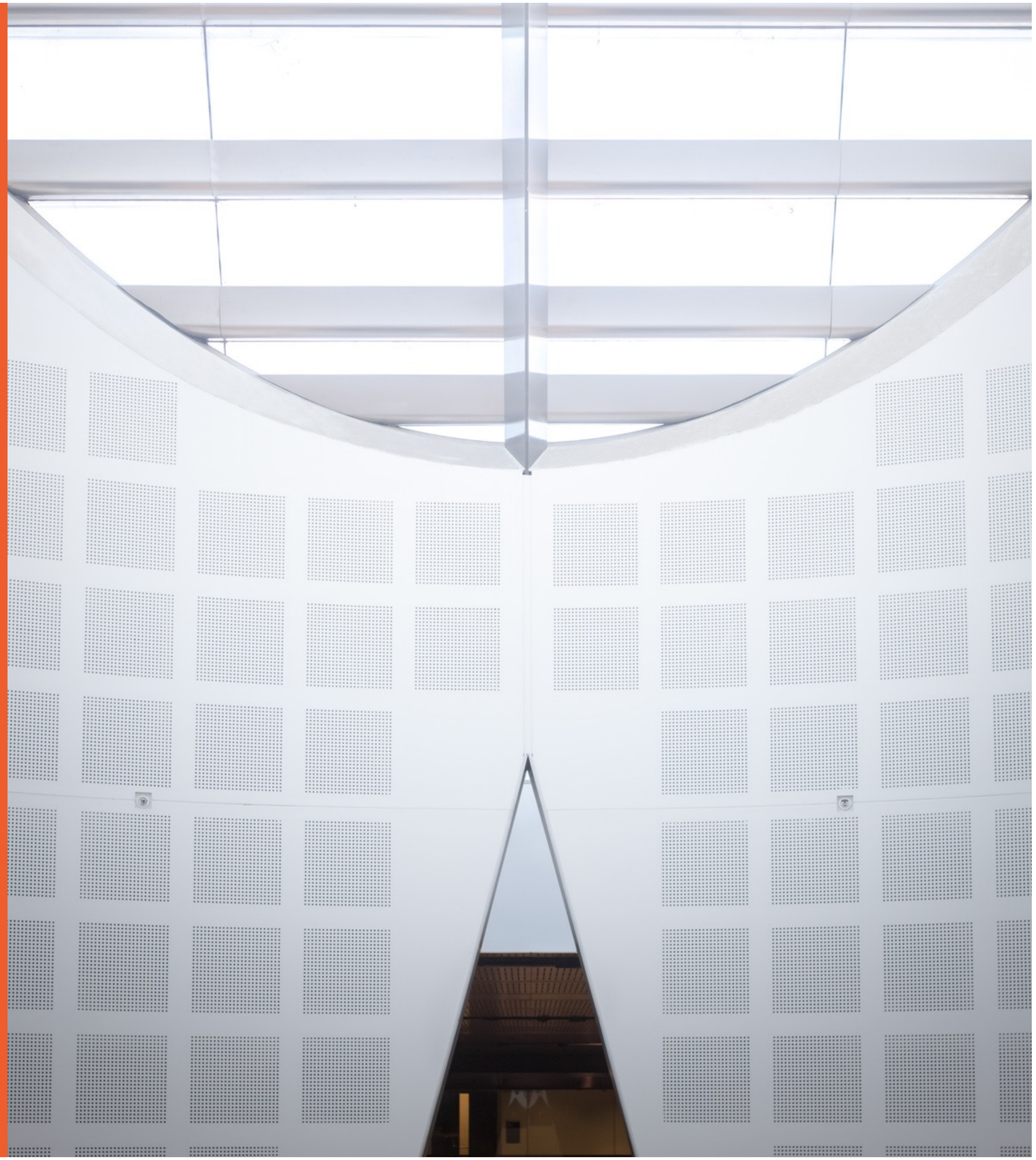
- OCL is a design language
  - Part of UML
  - Declarative
  - Growing community

- OCL cannot be executed directly
  - Formalize your constraints / contracts

- How to use it?
  - Translate OCL to assertions in your code
  - Limited number of OCL tools for checking design/code translation/etc.

Alloy

THE UNIVERSITY OF SYDNEY



Alloy

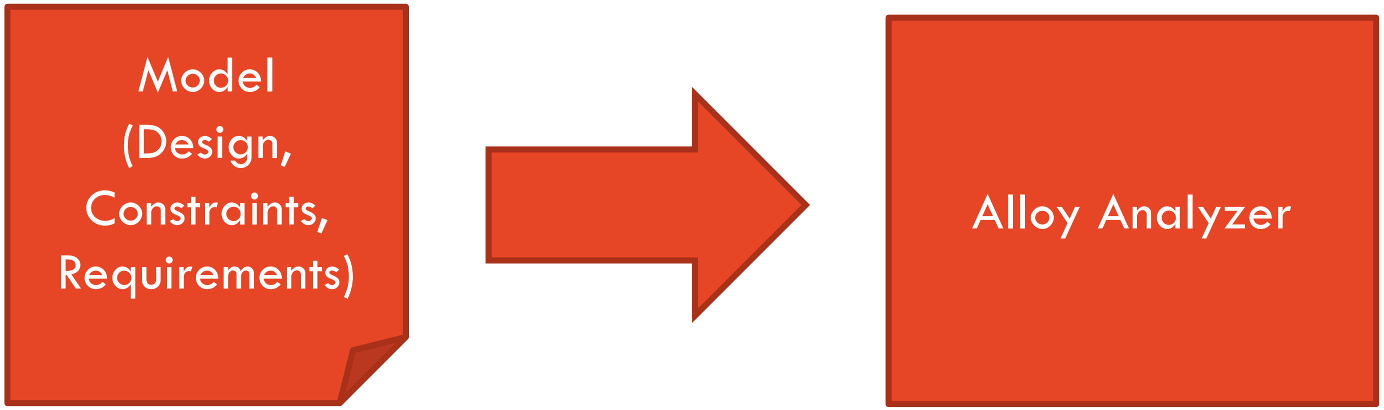
- Modelling Language for Design, Constraints, and Requirements
- Tool that checks the correctness up to a certain problem size
- Assumption: small problem sizes reveal most corner cases
- Relational logic
  - Alloy uses the same logic for describing designs, constraints, and requirements
  - for-all and exists-some quantifiers of first-order logic
  - operators of set theory and relational calculus.
- Modelling software designs with sets and relations
- Restrictions:
  - only first-order structures, no sets of sets, no relations over sets.

Alloy's Verification Process

- Process

  - Express the structural components and the constraints on components.

  - Alloy Analyzer tells if constraints are satisfied and, if so, what instances satisfy the constraints.



Alloy's Model Language

- Components are modelled as sets
- Basic set operations
  - union (+), difference (-), intersection (&), join (.), etc.
- Express component structure coarse-grained (=unconstrained)
- Refine components with constraints to check whether the design is working

Signatures

- Introduce a set of objects and some fields
- Fields relate to other objects
- Signatures can be seen as components/object classes
- Format

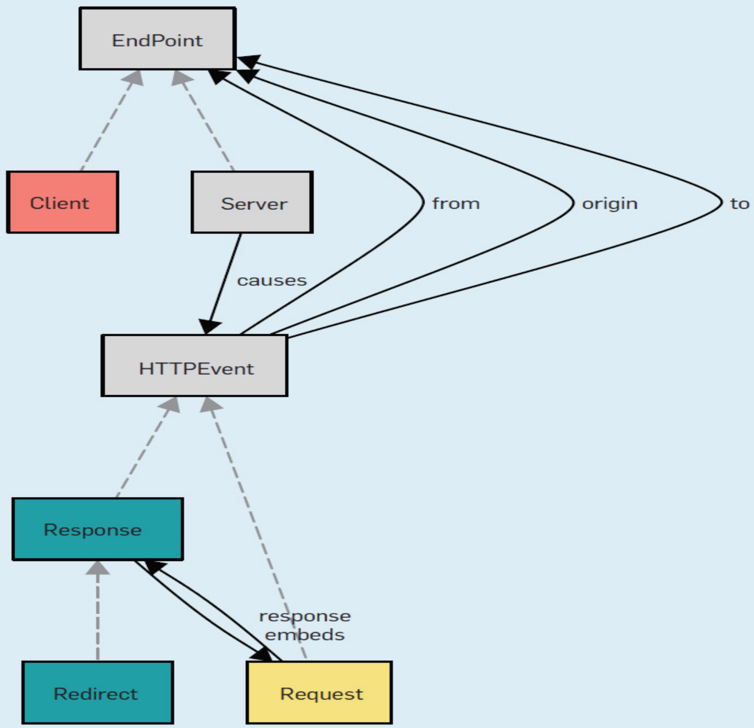
```
sig <sig-name> extends <super> {
    <fields> …
}
```

- Fields has the format

```
<name> , … : <multiplier> <sig-name>
```

Example: Signature

```
abstract sig EndPoint { }

sig Server extends EndPoint {
  causes: set HTTPEvent
  }

sig Client extends EndPoint { }

abstract sig HTTPEvent {
  from, to, origin: EndPoint
  }

sig Request extends HTTPEvent {
  response: lone Response
  }

sig Response extends HTTPEvent {
  embeds: set Request
  }

sig Redirect extends Response {
  }
```



From **Communications of the ACM, September 2019, Vol. 62 No. 9, Pages 66-76**

Example (cont'd)

- *Server* represents the set of server nodes, and has a field *causes*
- If no multiplier is specified, we assume a 1:1 relationship
  - Example: HTTP event has exactly one *from* endpoint, one to endpoint, and one *origin* endpoint
- The `lone` multiplier specifies at most one
- The `set` multiplier specifies multiple elements

Constraints

- Facts
  - Things that must hold
  - Format: `fact <name> { … }`
- Predicates
  - Defines re-usable predicates (like functions)
  - Format: `pred <name> (<parameters>) { … }`

Example: Constraints

```
fact Directions {
    Request.from + Response.to in Client
    Request.to + Response.from in Server
    }

fact RequestResponse {
    all r: Response | one response.r
    all r: Response | r.to = response.r.from and r.from = response.r.to
    all r: Request | r not in r.^(response.embeds)
    }

fact Causality {
    all e: HTTPEvent, s: Server | e in s.causes iff
        e.from = s or some r: Response | e in r.embeds and r in s.causes
    }

fact Origin {
    all r: Response, e: r.embeds |  e.origin = r.origin
    all r: Response | r.origin = (r in Redirect implies response.r.origin else r.from)
    all r: Request | no embeds.r implies r.origin in r.from
    }

pred EnforceOrigins (s: Server) {
    all r: Request | r.to = s implies r.origin = r.to or r.origin = r.from
    }
```

- Direction fact: every request is from, and every response is to, a client; every request is to, and every response is from, a server

Requirements

- Define a design property to check

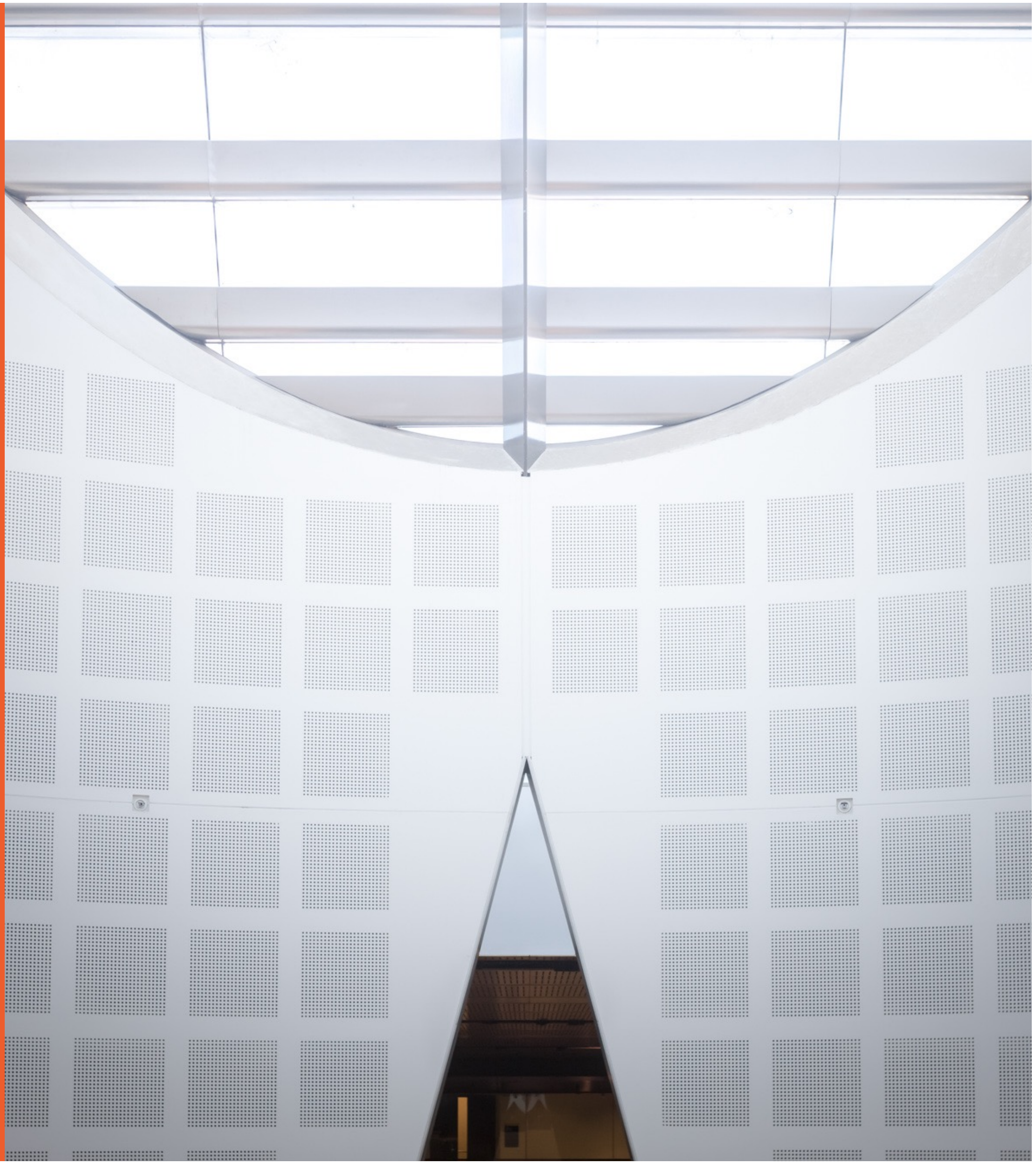
```
check {
    no good, bad: Server {
        good.EnforceOrigins
        no r: Request | r.to = bad and r.origin in Client
        some r: Request | r.to = good and r in bad.causes
    }
} for 5
```

- Checks only up to set size of 5 (grows exponentially!)

Summary

- Alloy is a modelling language
- Expresses Design, Constraints, and Requirements
- Checks the design fully automatically up to a certain set size
  - Verifies your design (not your program!!)
  - Small problem sizes will already reveal corner cases
- Is open-source and can be downloaded from here:
  - https://github.com/AlloyTools/org.alloytools.alloy
- More information:
  - https://cacm.acm.org/magazines/2019/9/238969-alloy/fulltext

Red, Green, Refactor



THE UNIVERSITY OF SYDNEY

Test-Driven Development

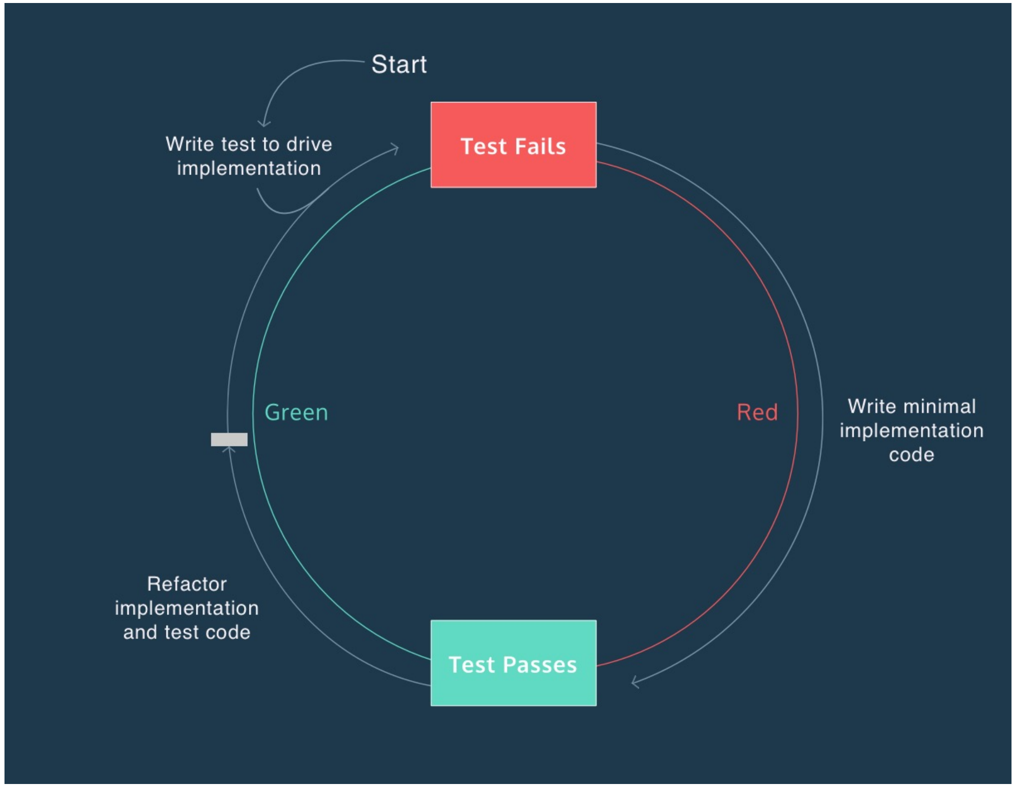
- Test-Driven Development (TDD)
- Write test cases first before design and development
- Design is evolved via refactoring
- Design → Test → Code vs. Design → Code → Test
- Tests drive the implementation
- Keep units small
  - Reduce debugging effort
  - Self-documenting tests

Test-Driven Development

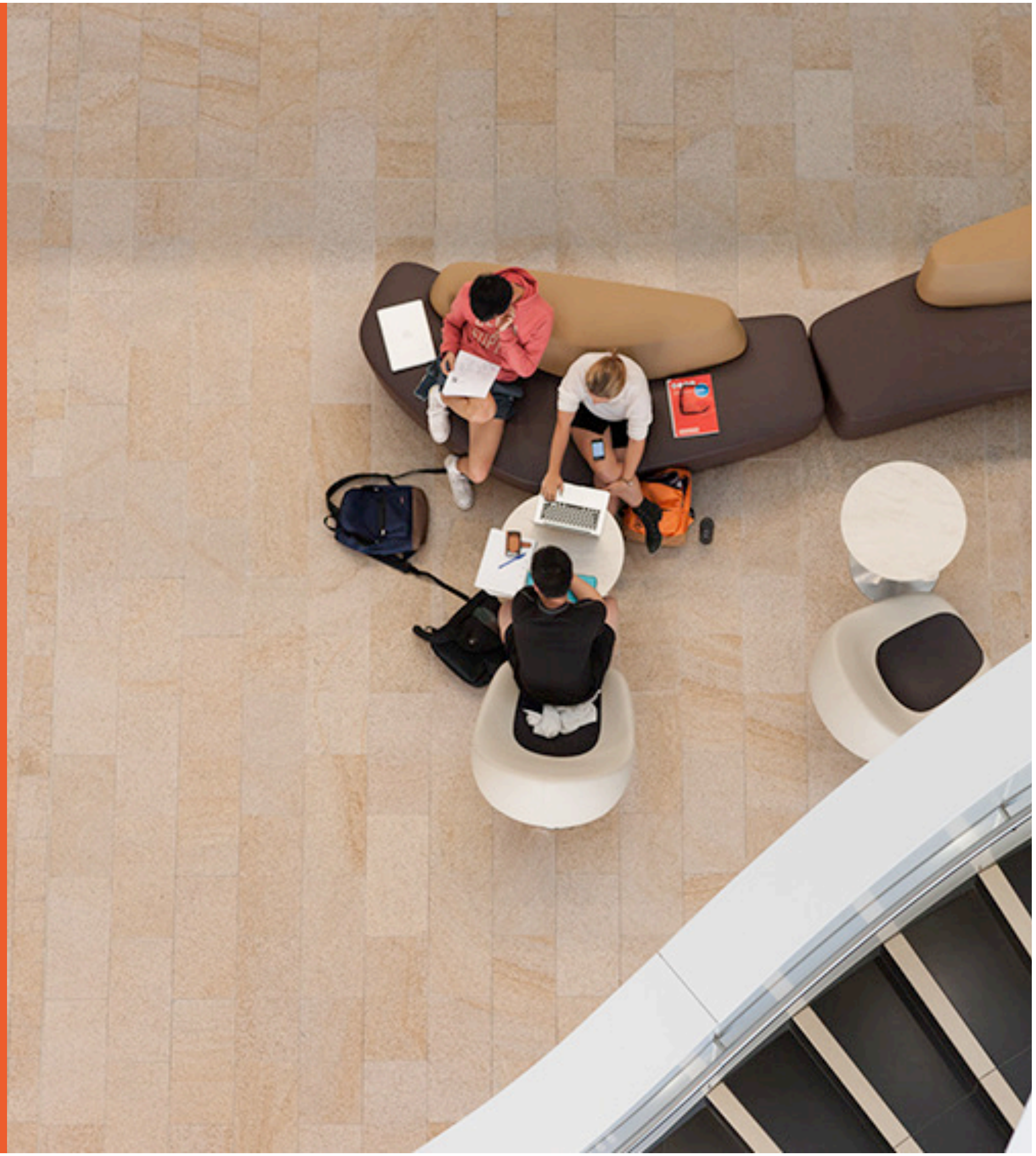
- Red – think about what you want to develop
  - Write a test that doesn't work; doesn't even compile at first
- Green – think about how to make your tests pass
  - Make test work; take short-cuts to make it work
- Refactor – think about how to improve your existing implementation
  - Eliminate all short-cuts & duplication to make the test work

Red, Green, Refactor

- Red Phase
  - Starting point
  - Find tests for implementation
  - Minimal implementation
- Green
  - Find solution that passes tests
- Refactor
  - Improve code/ more efficient



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

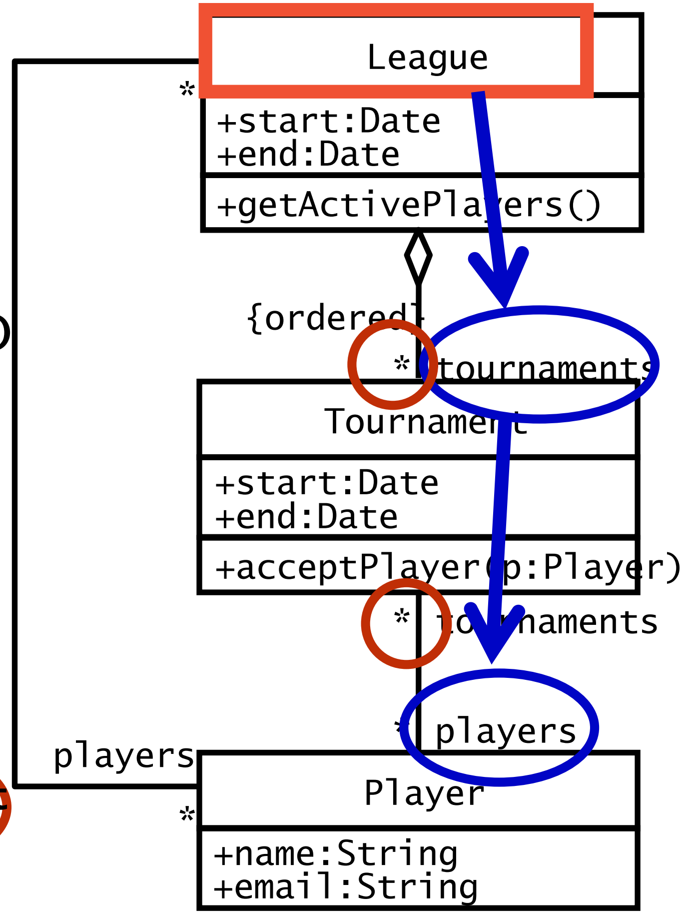
```
context Tournament inv:
 end - start <= Calendar.WEEK
```

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
```



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.*

References

- Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition).* Prentice Hall PTR, Upper Saddle River, NJ, USA.

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

- Martin Folwer, Patterns In Enterprise Software, [https://martinfowler.com/articles/enterprisePatterns.html]