09 - UML Class Diagrams

February 6, 2023

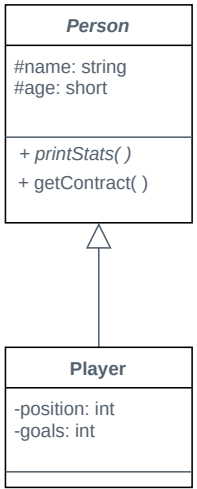
COMP2404

Darryl Hill

Contents

Class Diagrams

- The purpose of this section is to introduce you to UML Class Diagrams.
- First step into the larger world of documentation.
- When we are writing a program there are many types of documentation.
- Before we look at class diagrams, we will go over the reasons and importance of documentation
- Also how class diagrams fit in to the "Documentation-sphere".



Documention

Someone who works at Google once described their workweek as:

- One day of coding
- Two days of writing tests for that code
- Two days of writing documentation for that code

Why is documentation so important?

Try using a language or library with non-existent, trivial, or outdated documentation

- Using these is a horrible experience
- A lot of guesswork and trial and error

Documentation

Contrast that with a language like Java

- Has a documenting utility built into the JVM
- Javadocs can be “compiled”
- Very easy to look up a library or class

This is the experience you should want to give other developers

- Or perhaps yourself, later on, when you’ve forgotten how you wrote your own code

Documentation

What if your lead programmer gets a better job?

- No one knows how to finish the product
- Production slows down or stops as everyone tries to figure out where the lead left off

Types of Documentation

The software development life cycle

- Requirements analysis
  - Figuring out what you are building.
- Design
  - Making a plan.
- Implementation
  - Implementing the plan.
- Testing
  - Did this all work?

Each of these requires separate/different documentation

Types of Documentation

Requirements analysis (you will do this in COMP3004)

- Functional requirements
  - Specifies what the system will do
- Non-functional requirements
  - User-visible constraints on the system
- Use cases
  - Model the interactions between the user and the system
  - (particularly useful)
- High-level object model
  - A model of the interactions of the objects in your application
  - Major entity, boundary and control objects
- Dynamic model
  - System behaviour over time.

Types of Documentation

Design

- Subsystem decomposition
  - Specification of the high-level subsystems
- Detailed object model
  - Low level objects required to implement the system interfaces

Implementation

- Program comments, intelligent naming conventions
- Tutorials

Testing

- Test plan
- Test cases

Program Comments

Code should be self-documenting whenever possible

- Ideally should read like a story
- "first this happens, then this, if this happens" etc
- Keeping functions and classes with simple, single purposes and appropriate names will help with this
- `Product::getPriceWithTax();`
  - is self-documenting (but you may still add more details using comments)

Comments should describe

- Purpose of every class
- How to use (even when it seems obvious)
- Details of complex or critical class members

Test Cases

Test cases should describe

- What portion of the program you are testing
- The test input
- The expected output

Unit tests are usually pass / fail.

Integration and system testing - may have to describe the expected output or result.

- Interaction of parts is generally more complex.

Class Diagrams

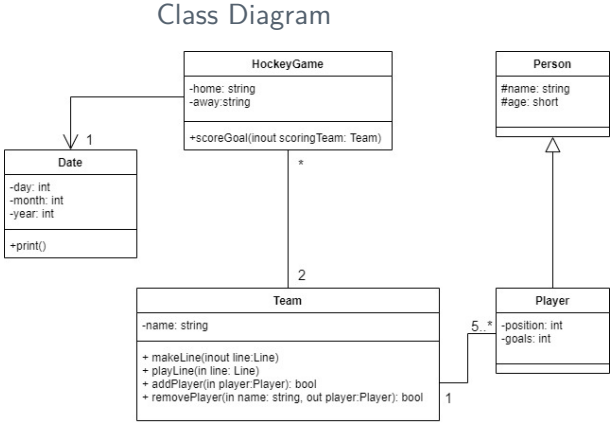
Unified Modelling Language (UML)

- Used during the design phase to help document OO design
- Family of notations used to represent OO models
- Communication tool for developers
- Programming language independent
  - A design expressed in UML can be implemented in any OO language

Class Diagrams

Types of UML diagrams:

- UML class diagram
  - Attributes and operations
  - Associations between classes
- UML activity diagram / sequence diagram
  - Behaviour of the program
  - Interaction between classes
- UML state machine diagrams

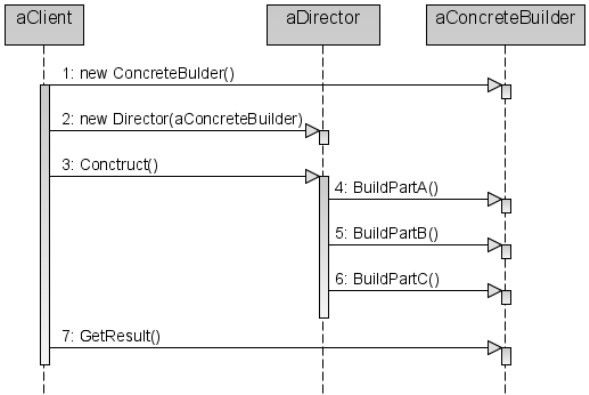


Sequence Diagrams

Types of UML diagrams:

- UML activity diagrams
  - Similar to a flow-chart
  - Models behaviour
- UML sequence diagrams
  - Timeline of when objects/functions are in use
  - Lifetime of temporary objects

Sequence Diagram



State Machine Diagrams

UML state machine diagrams

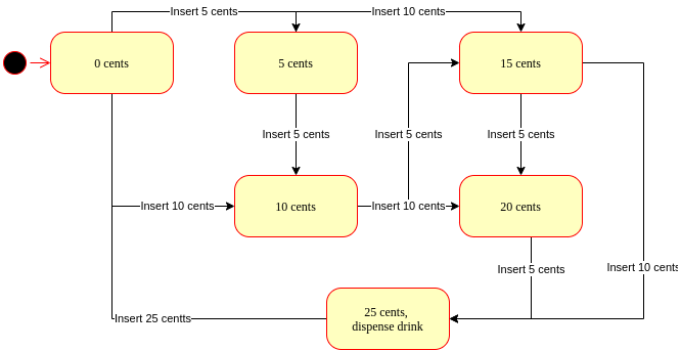
- Many programs can be modelled as state machines
- Or have components that are state machines

A stealth game might have states

- Enemies have seen / are aware of you
- Enemies not aware

Behaviour of enemies depends on the state

State Machine for Pop Machine



Class Diagrams

There are many different UML specifications.

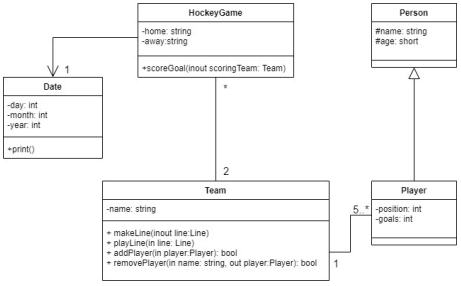
*****These slides are considered to be your UML Class Diagram specification*****

We will use a subset of the most common elements of UML Class Diagrams.

We will be modelling:

- Classes.
- Associations between classes.

Class Diagram

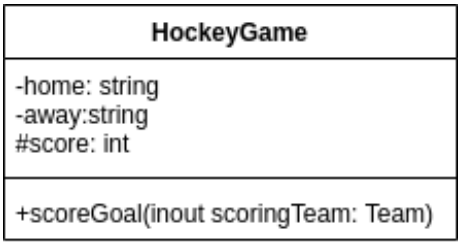


Class Diagrams

Classes consist of:

- Attributes
  - Generic terms for instance variables, data members, etc
- Operations
  - Generic term for member functions, methods, etc
- Access specifiers for attributes and operations
  - \- private
  - # protected
  - \+ public

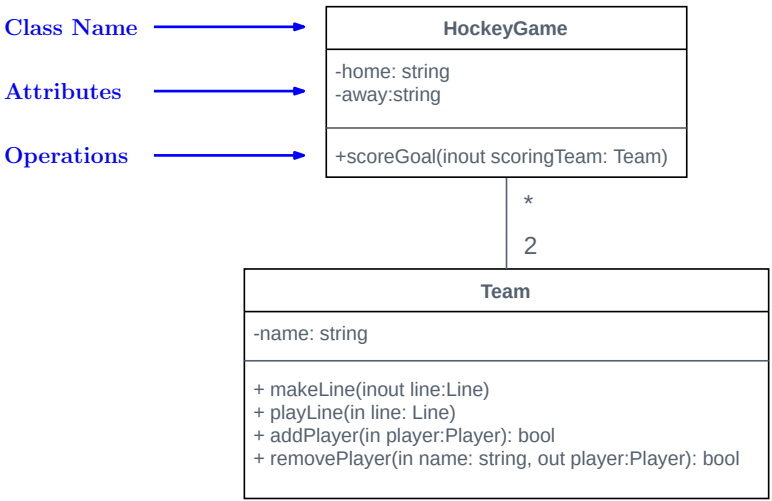
Class



UML Specification

We will make our classes with 3 sections:

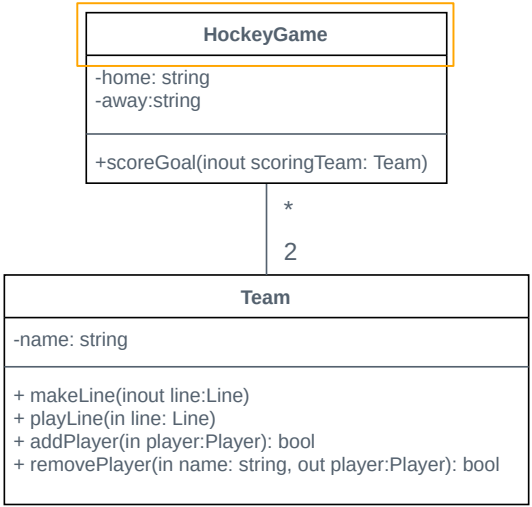
- **Class Name** - name of the class
- **Attributes** - member variables
- **Operations** - member functions



UML Specification

Class Name - name of the class

- If the class is abstract, use *italics*

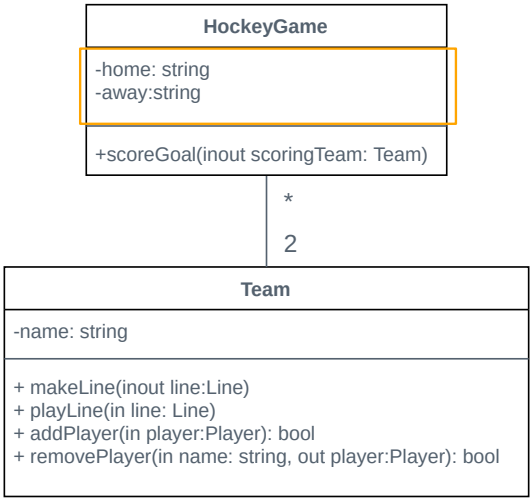


UML Specification

Attributes - member variables

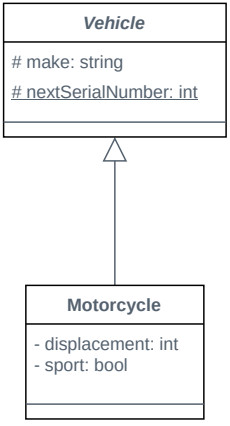
Syntax:

- start with access modifier `-, #, +`
- followed by the attribute **name**
- followed by **:** followed by **type**



UML Specification

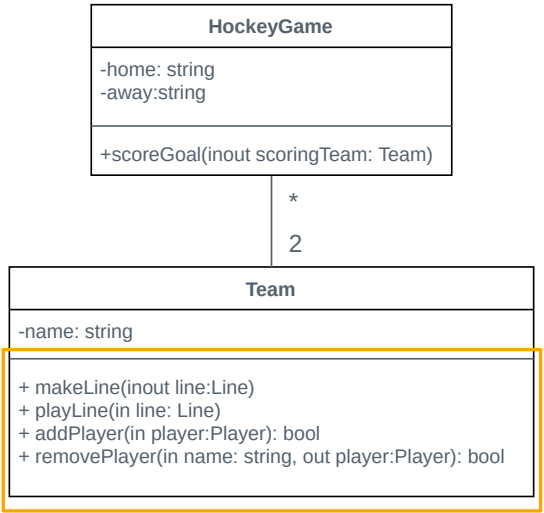
- Abstract classes and methods are shown in ***italics***
- Static members are <u>underlined</u>
- Friendship associations are never shown
  - (Friendship is specific to C++, not OO in general)



UML Specification

Operations - member functions
Syntax:

- access modifier `-, #, +`
- followed by operation **name**
- followed by **(parameters)** (separated by commas):
  - **in**, **out**, or **inout** parameter
  - parameter **name : type**
- followed by **:** followed by **return type**
- may be blank if returns `void`



UML Specification

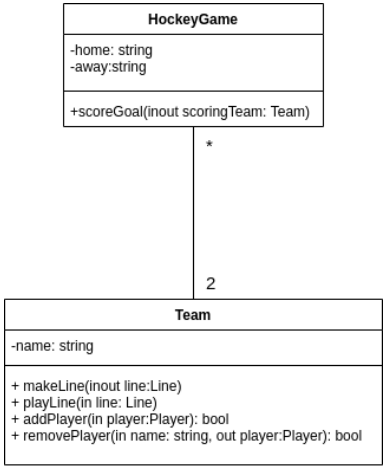
Associations model two main relationships between classes:

- Composition
- Inheritance

Composition

- "has-a" relationship
  - Container: the class the contains an instance of another class
  - Containee: the class contained within the other class

Composition



UML Specification

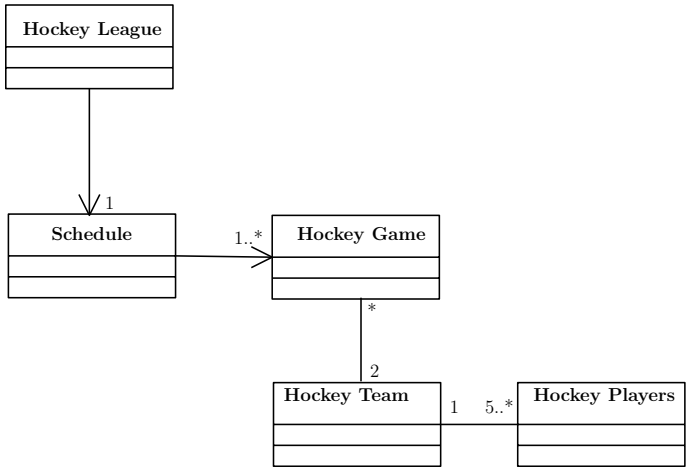
Characteristics of composition:

- Direction – which class contains which
  - Unidirectional – this is when class A contains one or more instances of class B
    - Line with arrow from A pointing to B
  - Bidirectional – class A contains one or more instances of class B AND class A contains one or more instances of class A
    - Line with no arrow
- Multiplicity
  - The number of containee instances possible in the container
  - Values: 0,1, many (*) or a range (1..* or 0..*)
  - Only applies to composition, never to inheritance!!

UML Specification

Composition

- Arrow point from container to containee
- If both contain each other, no arrows
- Always require multiplicity



UML Specification

Important convention:

- **We do not explicitly show *Collection* classes**
  - These are implied using the multiplicity indicators

Example

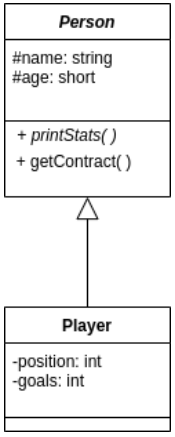
- `Library` contains many `Books`
- UML:
  - We DO NOT show the `BookArray` class
  - We DO show the `Library` as containing multiple `Books`
  - Composition relationship
  - Collection type is not important for design, important for implementation
- Program:
  - `Library` contains a `BookArray` object
  - `BookArray` contains `Books`

UML Specification

Inheritance

- "is-a" relationship
- Super-class, more generalized, also known as "parent", "grandparent", "ancestor"
- Sub-class, more specialized, also known as "child", "grandchild", "descendant"

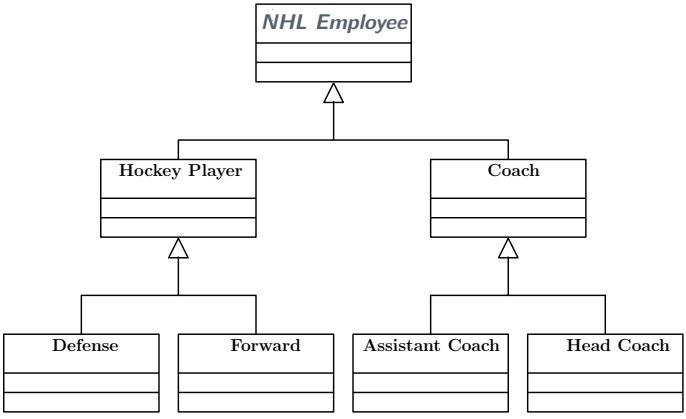
Inheritance



UML Specification

Inheritance

- Inheritance "triangle" points to super class



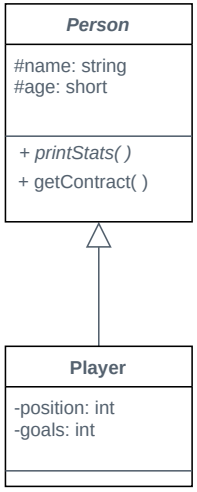
UML Specification

`Player` inherits from ***`Person`***:

- `name`, `age` attributes
- implements abstract operation `printStats()`
- overrides operation `getContract()`

These are not shown in the `Player` class in this UML

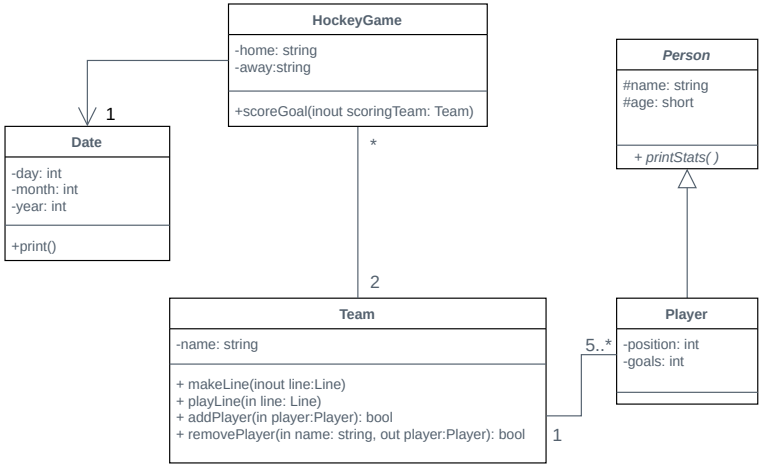
- We will accept either including them or not for the assignment
- Remember, UML is a communication tool
  - Including overridden functions in derived class may communicate a significant difference in implementation



UML Specification - DO's and DON'Ts

Do **NOT** show:

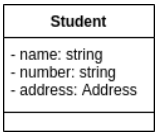
- constructor, destructor
- getter, setter
- collection classes



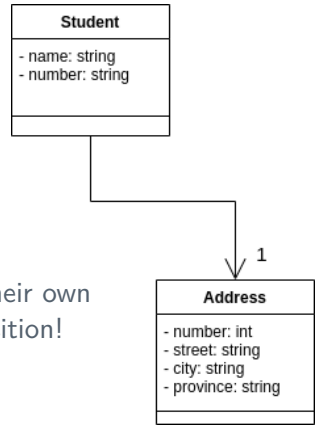
UML Specification - DO's and DON'Ts

Do **NOT** list object member variables with class attributes!

WRONG!!!



RIGHT!!!



DO show them as their own classes using composition!

UML Specification

These diagrams were made with **draw.io**, but you may use any UML drawing software

- **draw.io** can be used directly from your browser
- Also downloadable from
  - Windows Store
  - Ubuntu Software Center
  - Mac ¯_(ツ)_/¯
- On the VM, install snap, then use the command:
  - sudo snap install drawio