Object Oriented Design and Analysis
CPE 372

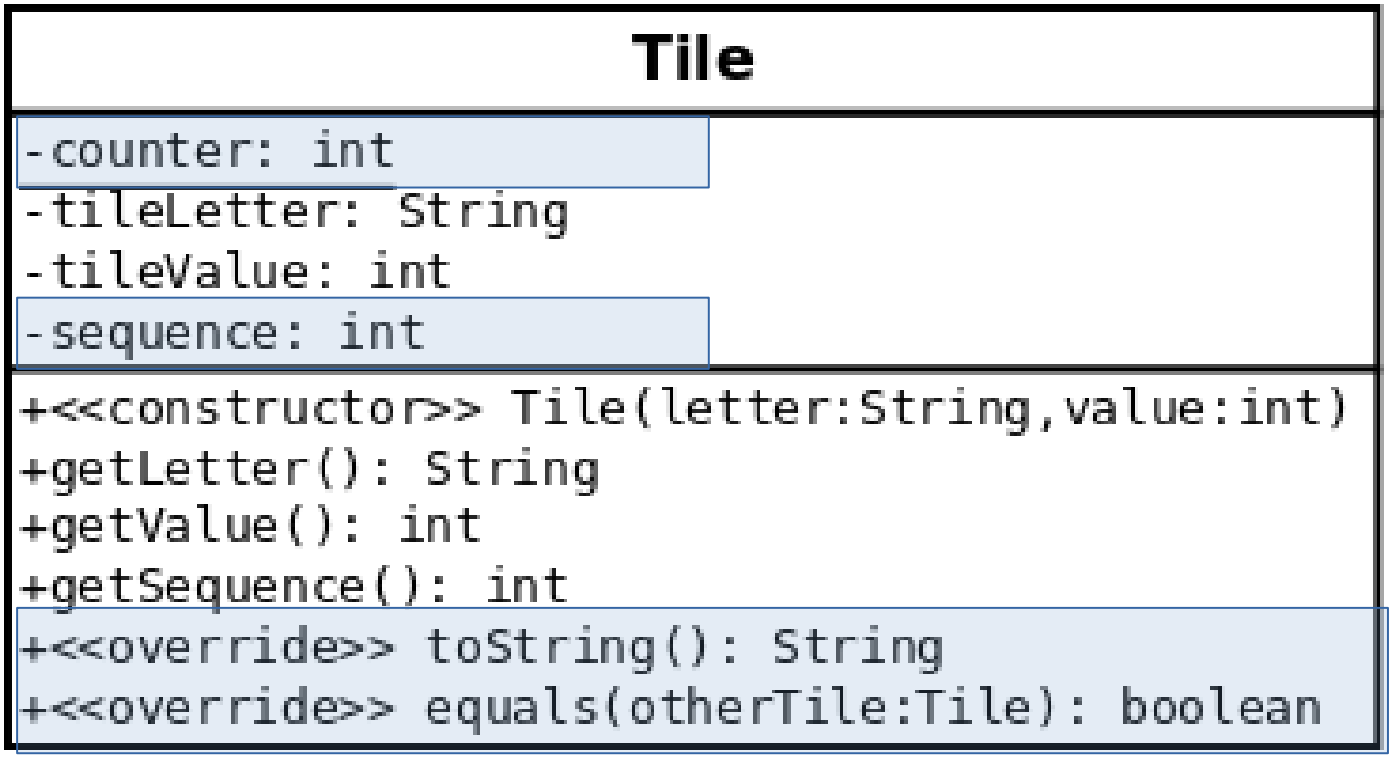
Lecture 8

Design Patterns 1

Dr. Sally E. Goldin
Department of Computer Engineering
King Mongkut's University of Technology Thonburi
Bangkok, Thailand

Scrabble Assignment

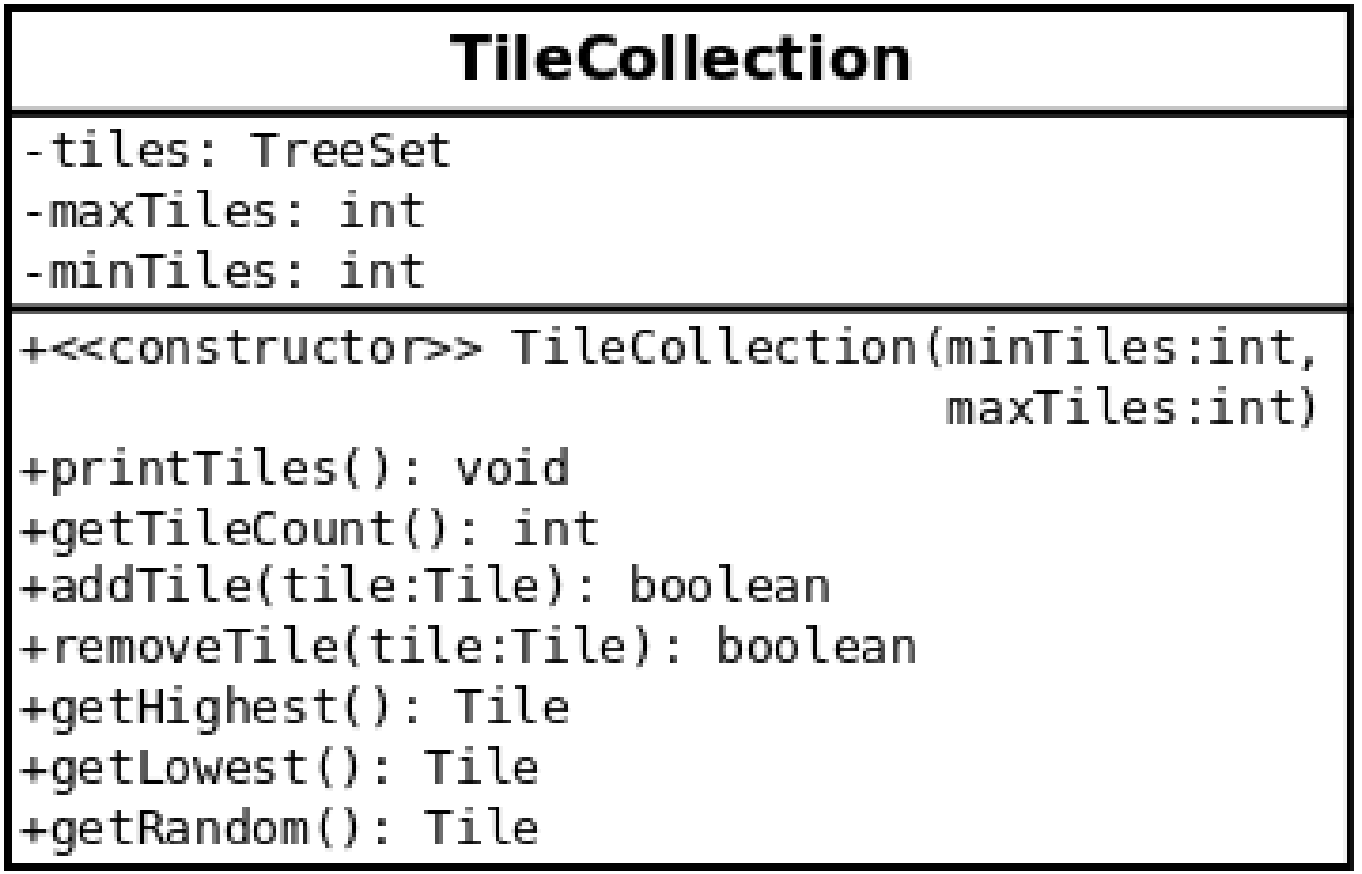
How should you approach a problem like this?
I started with the simplest class,
building from the bottom up.



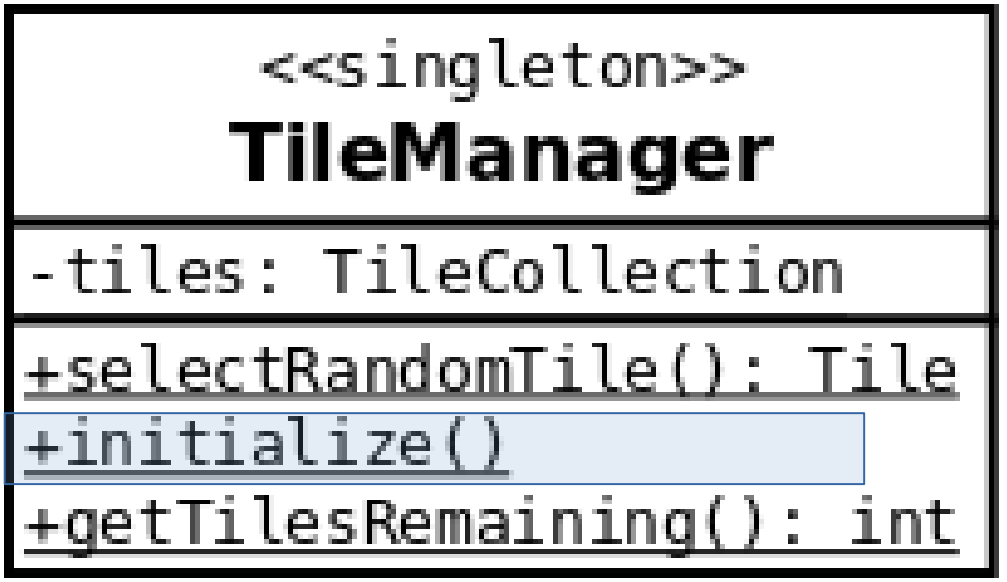
= new attributes or operations

Next class: ***TileCollection***

TileCollection is needed for several other classes



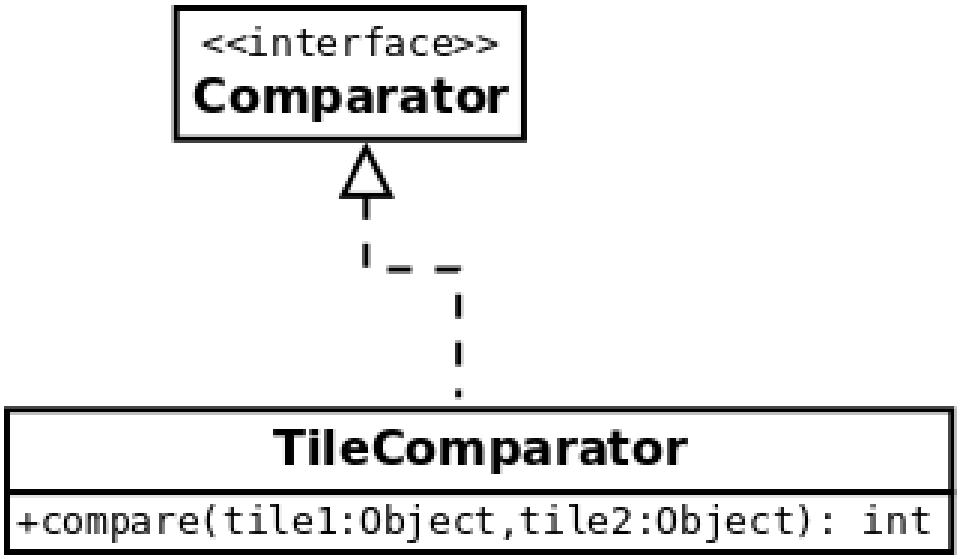
Next class: ***TileManager***



Added the initialize() method to populate the tiles collection with the standard set of Scrabble tiles

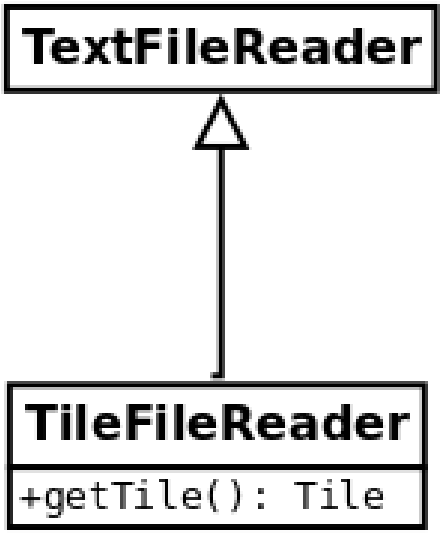
No arguments to this method.
Keep the details of initialization hidden so they can be easily changed

TileManager required two new classes



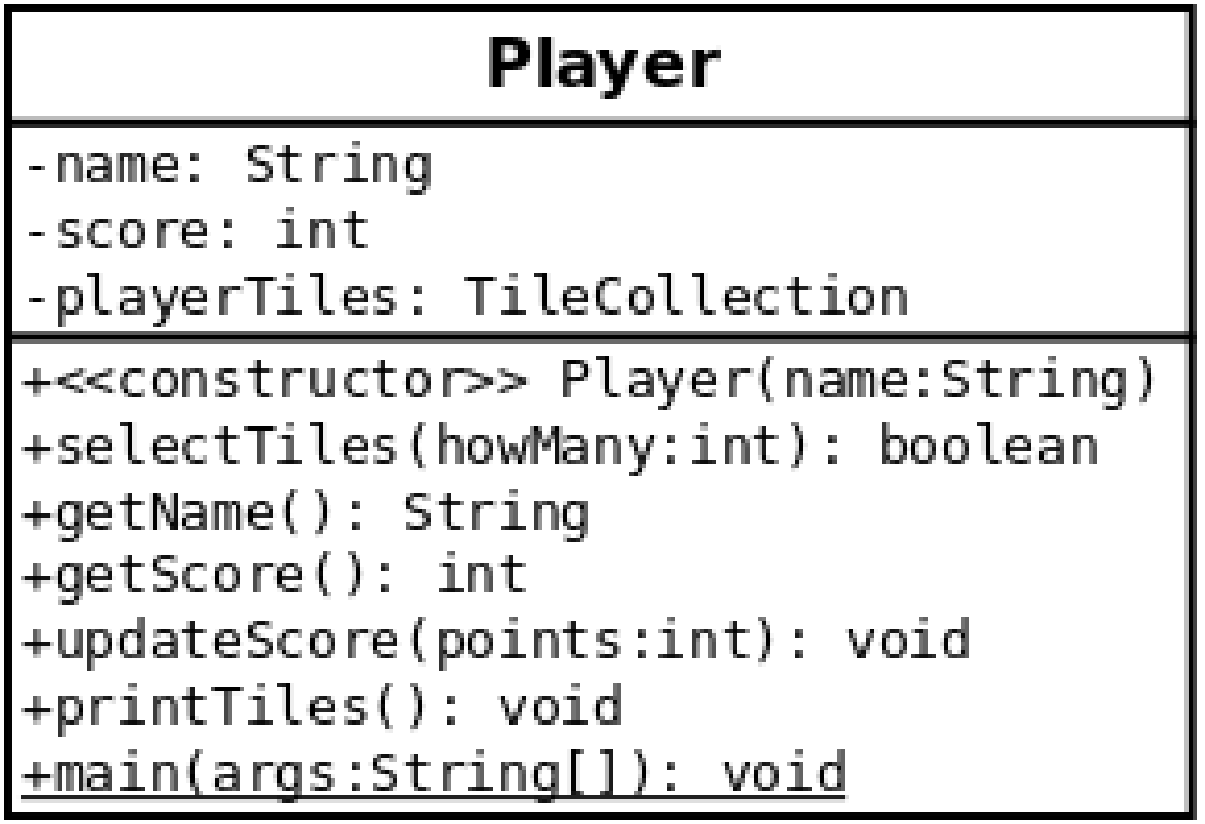
Provides a custom ordering of ***Tiles*** for the ***TreeSet***

Deals with the problem of “duplicate” ***Tiles***



Used in initialize() to read tile information from text file

Finally, created ***Player***



main() is used to execute the Select Tiles use case

Just one use case!

But many methods will be reused in other use cases

- Take a turn will use:
  removeTile() (***TileCollection***)
  selectTiles() (***Player***)
  updateScore() (***Player***)

- Determine first player will use:
  selectTiles() (***Player***)

Clearly we will need to design and implement other classes
However, we can build on what we have already

Introducing Software Design Patterns

"Software patterns are reusable solutions to recurring problems that occur during software development...As programmers gain experience, they recognize the similarity of new problems to problems they have solved before." *Mark Grand, Patterns in Java, Volume 1 (1998)*

Writing and testing the code for the Scrabble exercise took me about 2 hours.

How long did it take you?

Why discuss patterns?

- Quick and concise way to describe common solutions
- Shared language for developers to communicate about their designs
- Shortcut to experience for novice programmers

History of Software Design Patterns

Original ideas came from from architecture

C. Alexander – *A Pattern Language: Towns, Buildings, Construction* (Oxford University Press, 1977

First applied to UI design by Ward Cunningham & Kent Beck

"Using Pattern Languages for Object-Oriented Programs", OOPSLA-87

Gang of Four (GoF) – Erich Gamma, Richard Helm, John Vlissades, Ralph Johnson

Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley, 1994)

Design patterns are:

Reusable

The same approach can be applied in many programs

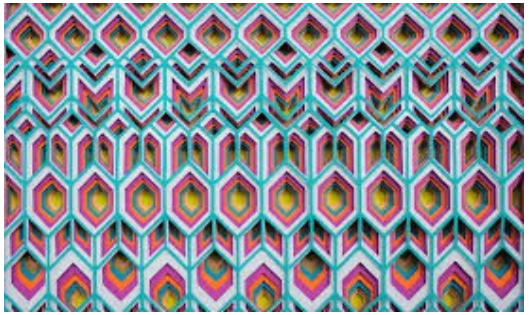
Abstract

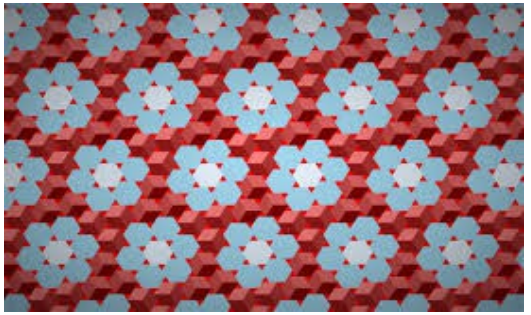
A pattern is a generalization which does not provide actual code

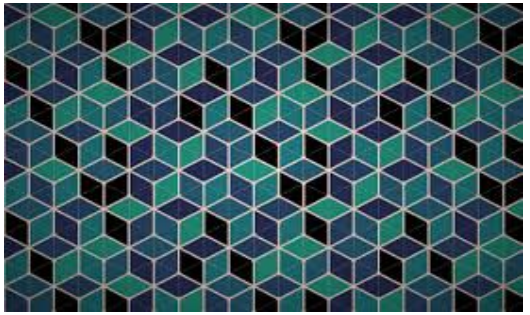
Subject to interpretation

There is no single agreed-upon list of patterns

Two developers might disagree about the identity of a pattern in a particular case







Singleton Pattern

Problem

A class manages and modifies resources which are common to the application as a whole. We want to avoid conflicts that might occur if multiple instances of the class can access the resources.

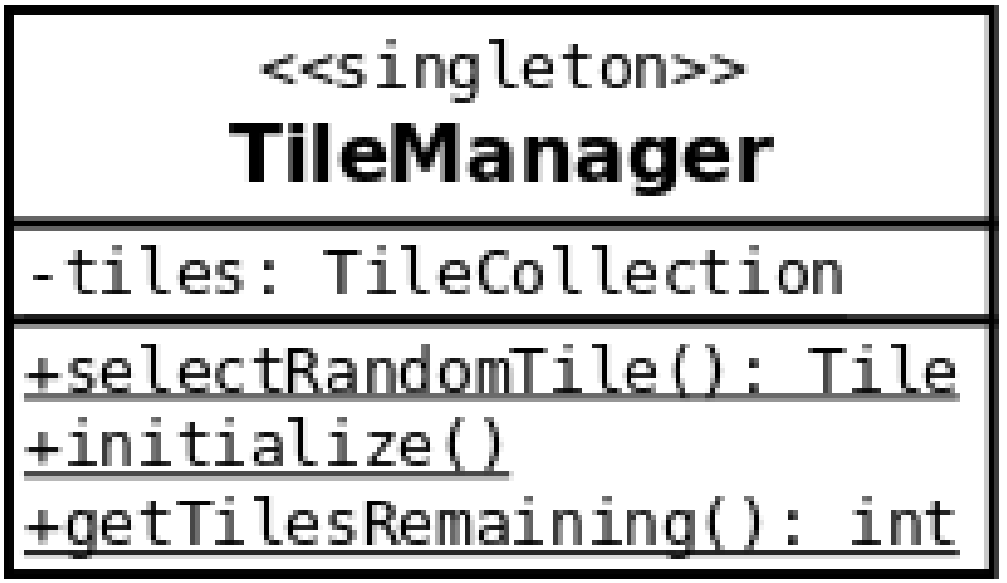
Solution

Guarantee that there can never be more than one instance of the class at any time

Example

TileManager holds and controls the remaining pool of available tiles. We use the Singleton pattern so there will never be more than one instance trying to select, remove or add tiles in the collection.

Singleton Implementation: 1



I made all the methods in ***TileManager*** be static (class) methods.

In this case we do not ever need to create an instance of the class.

Singleton Implementation: 2

Unfortunately we can still create instances!

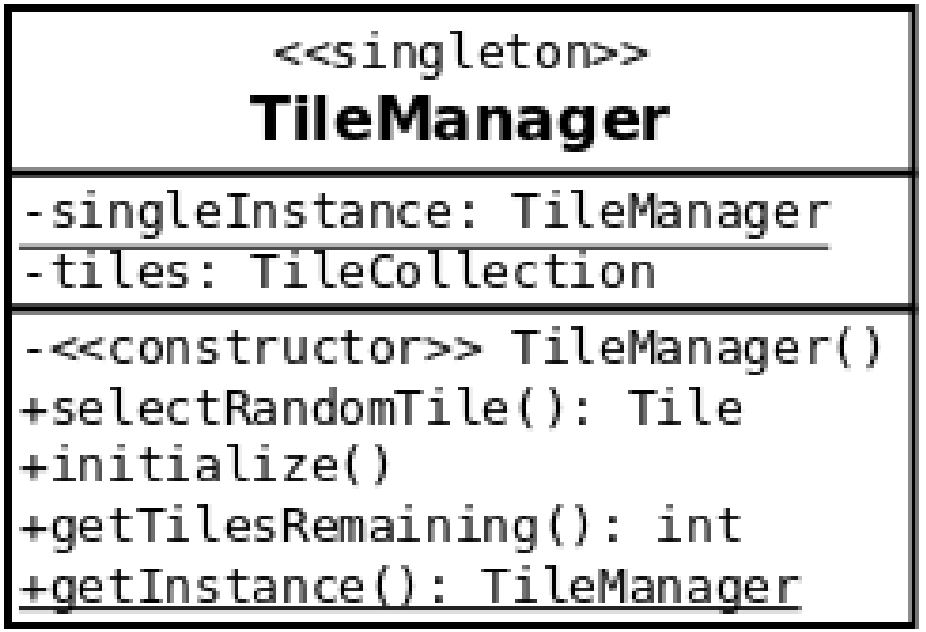
(See `TileManagerTester.java`)

This occurs because every class has a default constructor

Singleton Implementation: 3

The solution is to make the constructor **private**

If we want to avoid static methods, we can also create a getInstance() method



Singleton Implementation: 4

Code that uses ***TileManager*** would change as follows:

```
Tile t = TileManager.getInstance().selectRandomTile();
```

The getInstance method might look like this:

```
public static TileManager getInstance()
{
    if (singleInstance == null)
        singleInstance = new TileManager();
    return singleInstance;
}
```

Immutable Pattern

Problem

Instances of a class are shared by multiple objects. We want to avoid incompatible changes to those instances.

Solution

Make all data private. Initialize all data of the class instance in the constructor. Provide getter methods so other objects can retrieve instance data, but no methods to allow changing the data.

Example

Tile instances may be referenced by the ***TileManager***, by ***Player*** objects, by ***Word*** objects and by ***Square*** objects. The constructor creates a ***Tile*** instance with a letter and a score value, and assign a sequence number internally. After that, a ***Tile*** instance will never change, though it may be "owned" by different classes.

Delegation Pattern

Problem

Class A would like to reuse functionality defined by method M in class B. However, it does not make sense for A to be a subclass of B.

Solution

Instances of class A have a reference to an instance of class B. When method M is called in class A, that instance calls the corresponding method in class B.

Example

Player has a method printTiles() to print the tiles that the player currently holds. When that method is called, the ***Player*** instance calls the printTiles() method of its ***TileCollection*** (playerTiles).

More about Delegation

Delegation may be the most fundamental design pattern

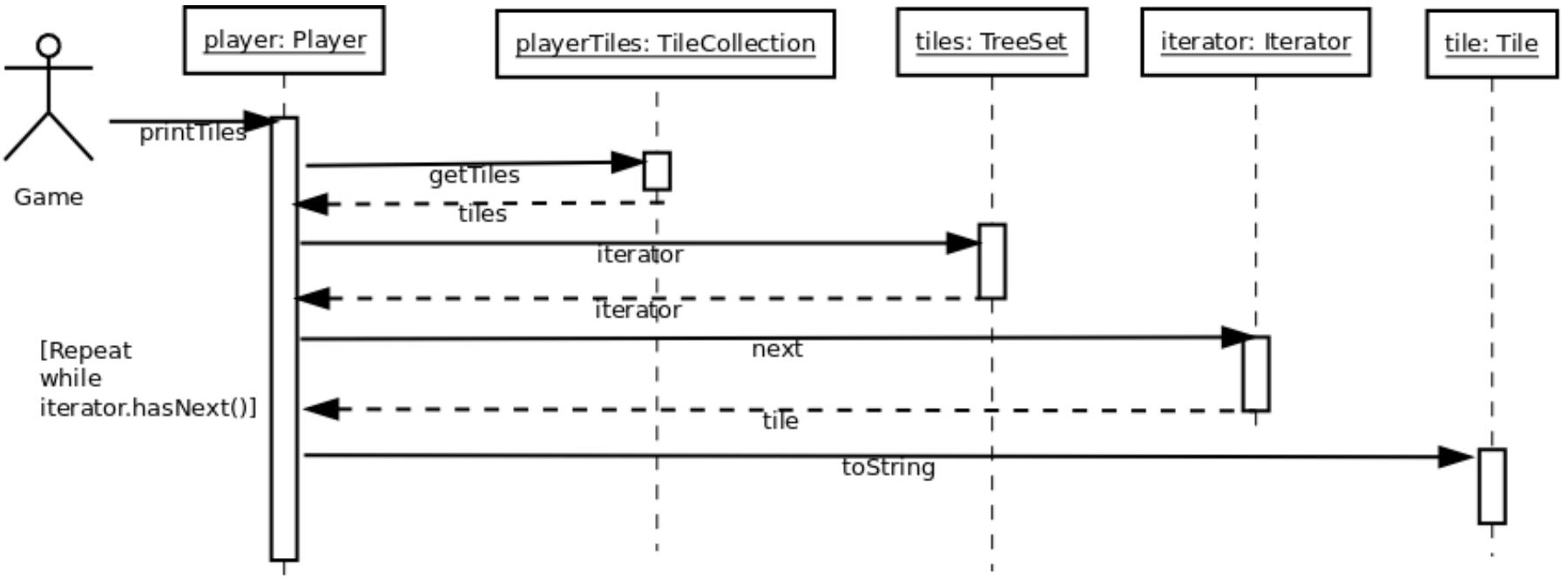
Many more complex patterns include delegation

(for instance the Façade pattern we will discuss soon)

Delegation helps reduce coupling.

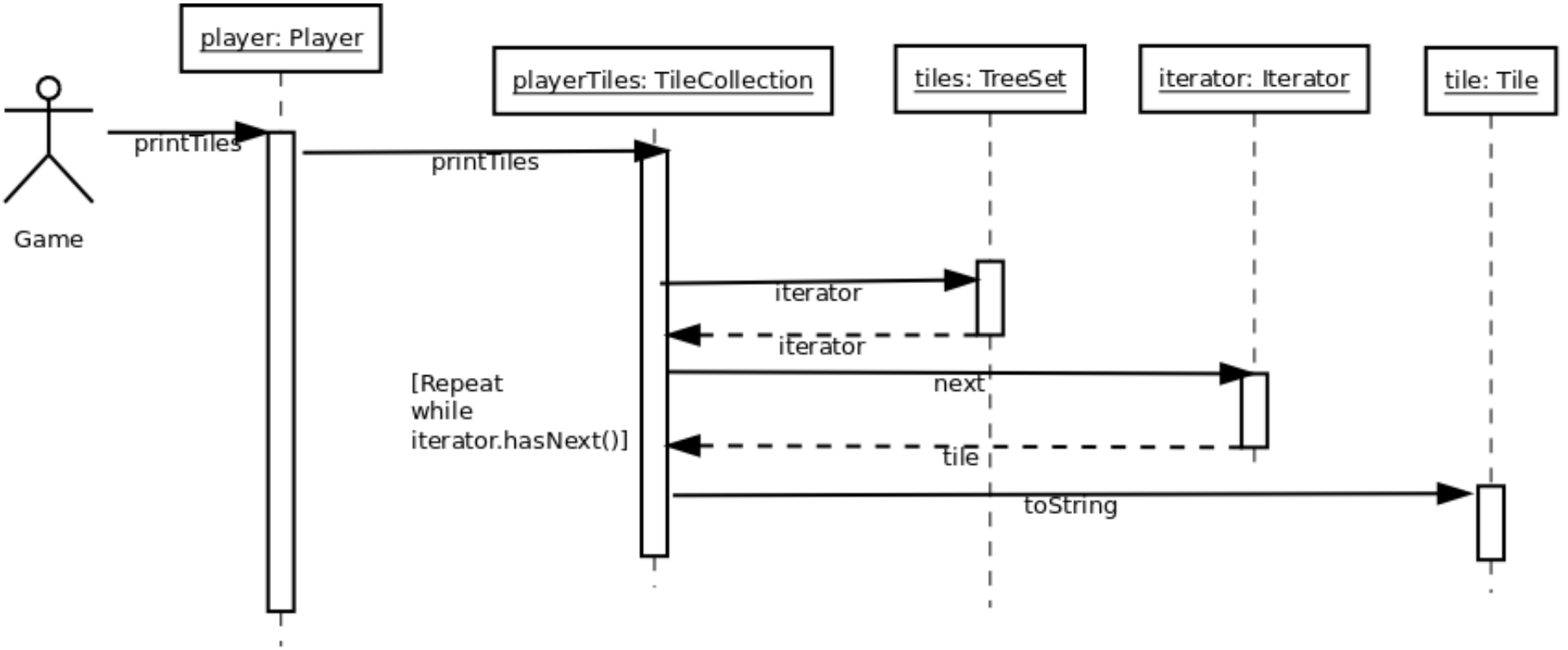
A particular class needs a reference only to its immediate delegate.

Without Delegation



The ***Player*** class needs to know about the internal implementation of the ***TileCollection*** class.
If these change, ***Player*** needs to change also.

With Delegation



Implementation details are encapsulated inside the ***TileCollection*** class. We could change how tiles were stored and accessed without any changes to ***Player***.

Facade Pattern

Problem

To provide system functionality, we need to call methods in a variety of related classes, sometimes in a specified order and subject to specific constraints. There are many dependencies between these classes and methods.

Solution

Create an intermediate class (the facade class, sometimes written “façade”) that handles all communication with the other classes. External classes wishing to access the services provided by these other classes only need to interact with the façade class.

“Façade” is a term from architecture

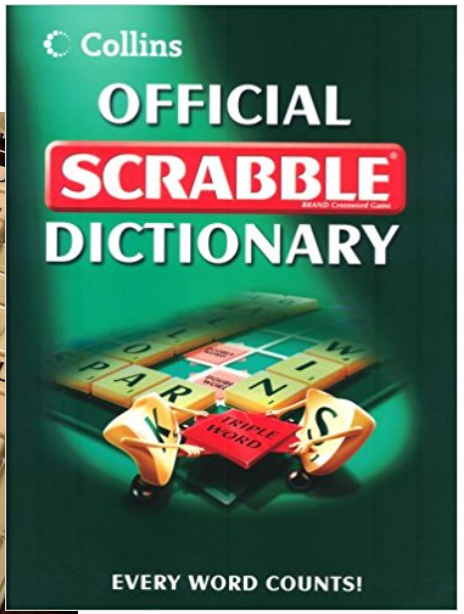
Façade of the Paris Opera House

Façades without any buildings behind them

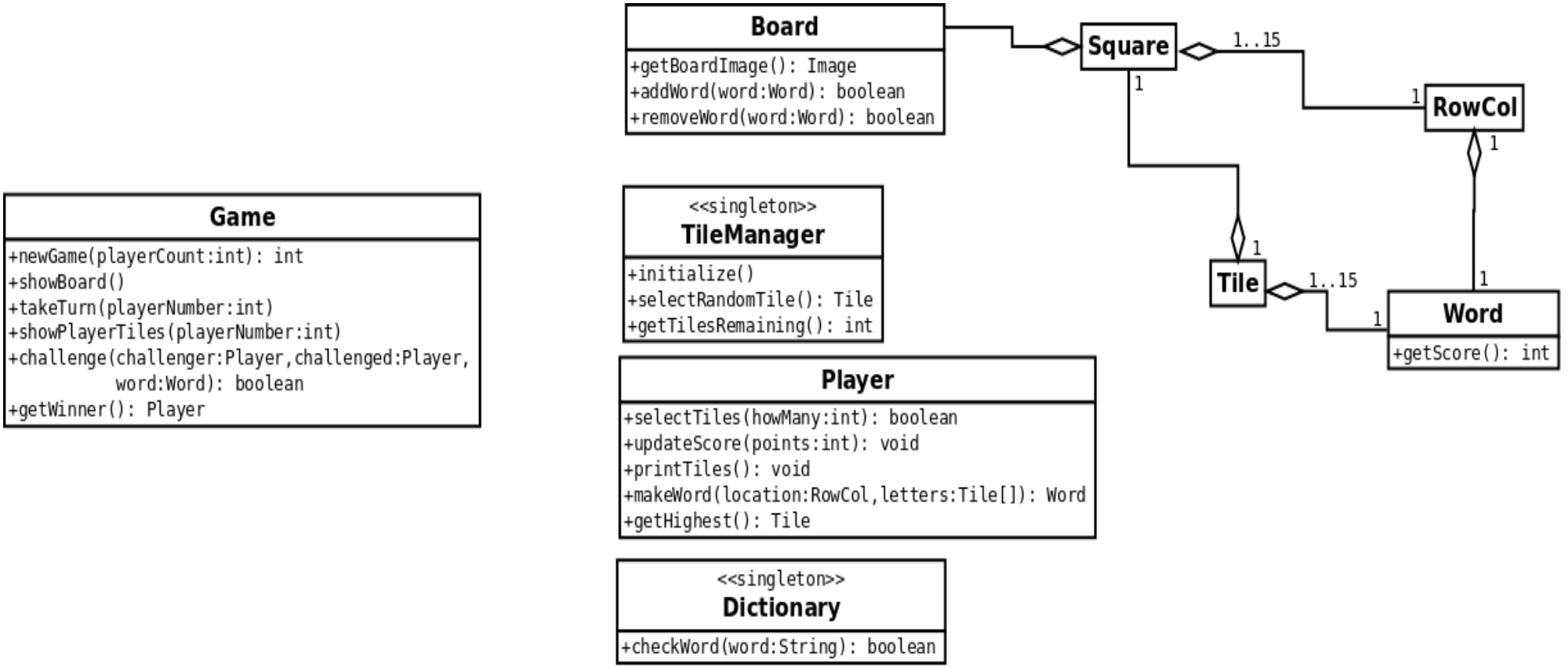
Example

As we continue implementing our Scrabble game, we will need to coordinate classes for the board, multiple players, words, the dictionary, etc.

We might create a ***Game*** class to serve as a facade between the user interface and the details of game play

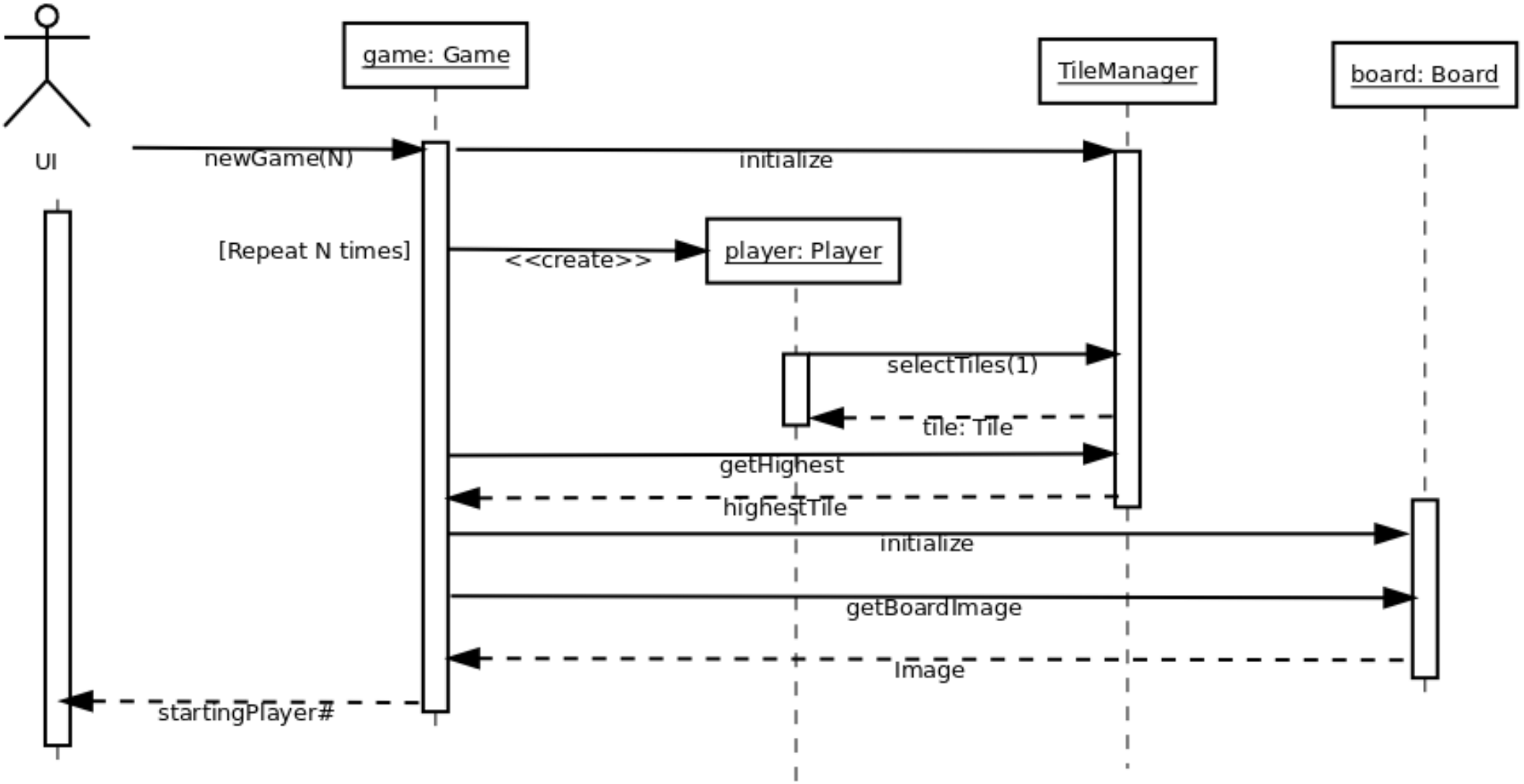


Partial Class Diagram



Game will hold references to a ***Board*** instance, the ***TileManager,*** the ***Dictionary***, and up to four ***Player*** instances

Sequence Diagram for New Game



- ***Game*** delegates most of the work to other classes
- ***Game*** hides the details of the other classes behind its facade
- If we change these classes the UI won't change at all

Assignments

- Read the brief article on design patterns here:

https://airbrake.io/blog/design-patterns/software-design-patterns-guide

- Based on the information in this lecture, plus the code in the **demos** directory for this lecture, create a ***Game*** class and implement the newGame() method.
  - This method should take an integer between 2 and 4 which is the number of players, and return the number of the player with the highest tile score after the initial selection
  - `Game.java` should include a main() to create an instance of ***Game***, call newGame(), and print the number of the player with the highest tile score
  - You will need to create a ***Board*** class but this can be a skeleton that does not do anything real. The getBoardImage() method can return this image:

    http://windu.cpe.kmutt.ac.th/ScrabbleBoard.jpg