Section A

Problem Description

Suppose that you are writing part of a video game in which various entities will battle against one another by casting spells. Entities are represented by an abstract class, **Entity**. There are two kinds of entity: *magicians* and *transport units*, represented by classes **Magician** and **TransportUnit** respectively.

Every entity has a *name* and a non-negative number of *life points*. The life points of an entity decrease as the entity is hit by enemy spells. While an entity has a positive number of life points, the entity is considered alive. When an entity's life points becomes zero the entity is considered dead.

A transport unit is a *composite* entity. In other words, a transport unit consists of various other entities: magicians and other transport units. Throughout this exercise you can assume that a transport unit does not contain itself (directly or indirectly). You do **not** need to add assertions to check for this.

Entities are hit by *spells* which are cast by *spell casters.* This is achieved via an instance method of **Entity**, **`applySpell`**, which takes a parameter of type **SpellCaster**. **SpellCaster** is an *interface* that specifies a method **`getStrength`,** which returns an integer. When **`applySpell`** is invoked on an entity that has *n* life points, the entity's life points are reduced by the strength of the spell caster, or by *n*, whichever is less. **In addition**, if the entity is a transport unit *t*, life points are deducted from entities contained (directly or indirectly) in *t* as follows:

- An entity with *n* life points contained *directly* in *t* has its life points reduced by 50% of the strength of the spell caster, up to a maximum of *n*
- An entity with *n* life points contained in a transport unit contained in *t* has its life points reduced by 25% of the strength of the spell caster, up to a maximum of *n*
- An entity with *n* life points contained in a transport unit contained in a transport unit contained in *t* has its life points reduced by 12.5% of the strength of the spell caster, up to a maximum of *n*
- *etc.*

The **`applySpell`** method returns the total number of life points that were deducted.

To illustrate this, consider the object diagram shown in Figure 1. If method **`applySpell`** is called on **TransportUnit** t1 with a parameter of type **SpellCaster** with strength 400, **`applySpell`** returns 1318. The resulting life points are:

t1:	2100	**t2**:	150	**m1**:	100	**m2**:	0
m3:	400	**m4**:	0	**m5**:	850	**m6**:	200

Note that entities contained directly in t1 get 200 life points deducted (50% of 400), while entities contained in t2 get 100 life points deducted (25% of 400). Note also that m4 has only 18 life points deducted, since deducting further life points would lead to a negative number of life points.

All entities provide a method **`minimumStrikeToDestroy`** that returns the lowest strength needed by a **SpellCaster** in order to reduce the life points of the entity, and of all entities contained (directly or indirectly) within the entity, to zero.

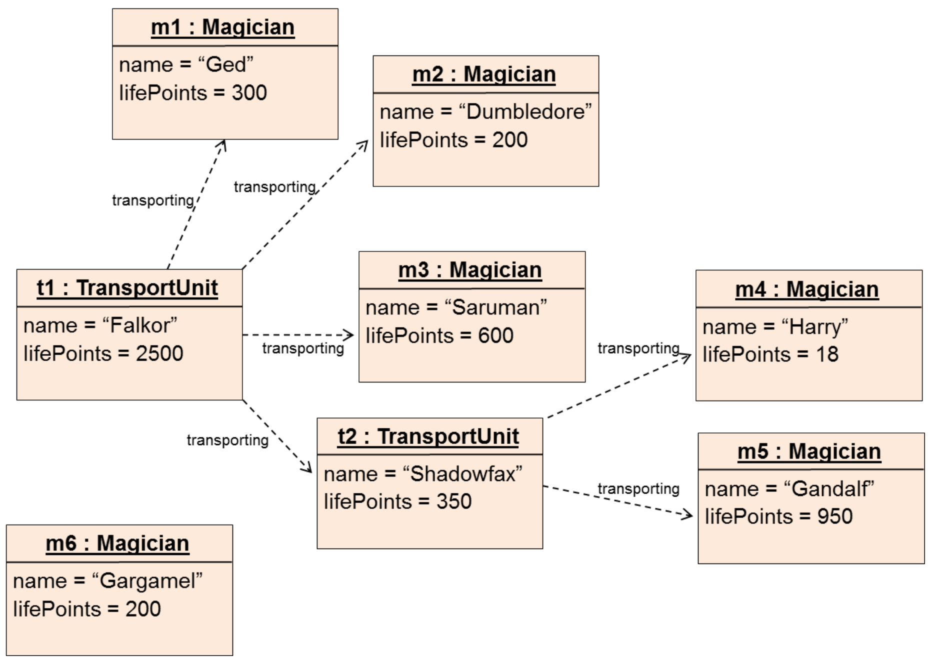


Figure 1: Object diagram showing various **Magician** and **TransportUnit** objects. The *transporting* arrows indicate the entities which a **TransportUnit** contains.

For the object diagram of Figure 1, **`minimumStrikeToDestroy`** returns 3800 when applied to t1. To see this, observe that a spell cast with strength 3800 will reduce t1's life points by up to 3800, m1, m2, m3 and t2's life points by up to 1900 (50% of 3800), and m4 and m5's life points by up to 950 (25% of 3800). This is sufficient to reduce all life points to zero. A lower strength would not reduce m5's life points to zero.

Files Provided

You will find a **SectionA** directory in your Lexis home directory with a **src** subdirectory. Inside **src** is a **videogame** package containing the following Java files:

- **Entity.java**
- **GameReport.java**
- **Magician.java**
- **SpellCaster.java**
- **TransportUnit.java**

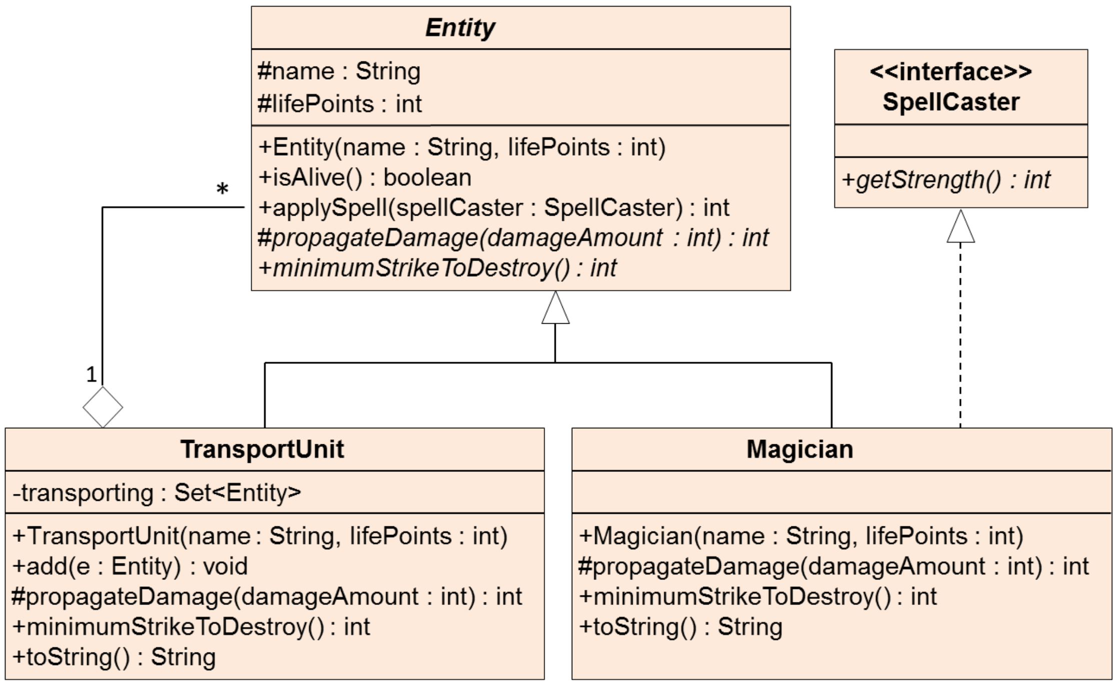
You should edit these files as explained below. To enable auto-testing, you should not change the signatures or visibility levels of any of the provided methods or fields. You may feel free to add additional methods and classes as you see fit.

If you work using Eclipse, you will find it already set up with a **SectionA** project for the source files, however the project is slightly out of sync. To sync up, do the following:

- Click on the **SectionA** project and then press **F5** or click **File->Refresh**
- Click on the menu **Project->Clean** and in the new window that pops up, click **Ok**

After this you can work on the project directly in Eclipse.

Note that the source code you are initially provided with does not compile. This is due to the missing **SpellCaster** interface, which you must write as described below.



Entity
#name : String
#lifePoints : int
+Entity(name : String, lifePoints : int)
+isAlive() : boolean
+applySpell(spellCaster : SpellCaster) : int
#propagateDamage(damageAmount : int) : int
+minimumStrikeToDestroy() : int
<<interface>>
SpellCaster
+getStrength() : int
*
1
TransportUnit
-transporting : Set<Entity>
+TransportUnit(name : String, lifePoints : int)
+add(e : Entity) : void
#propagateDamage(damageAmount : int) : int
+minimumStrikeToDestroy() : int
+toString() : String
Magician
+Magician(name : String, lifePoints : int)
#propagateDamage(damageAmount : int) : int
+minimumStrikeToDestroy() : int
+toString() : String

Figure 2: UML class diagram showing the inheritance and compositional relationships between **Entity**, **TransportUnit**, **Magician** and **SpellCaster**

Your Task

Figure 2 shows the relationships between the **Entity** abstract class, its subclasses **Magician** and **TransportUnit**, and the **SpellCaster** interface. Your task is to implement the class relationships and missing functionality in Java. Where appropriate you should use assertions to check preconditions for methods.

1) Write the interface **SpellCaster** in **SpellCaster.java (2 marks)**

2) Implement the stub methods `isAlive` and `applySpell` in the **Entity** class in **Entity.java**
   - `isAlive` should return true if and only if the entity on which it is invoked is alive
   - `applySpell` should call the `propagateDamage` method, passing the strength of the **SpellCaster** parameter as an argument to `propagateDamage`, and should return the same result as `propagateDamage`

   (4 marks)

3) Write the class **Magician** in **Magician.java**. **Magician** should implement the **SpellCaster** interface and extend the **Entity** abstract class
   - **Magician** must provide implementations of any methods specified by the **SpellCaster** interface. The strength of a magician is double the magician's life points
   - **Magician** must provide implementations of the abstract methods `propagateDamage` and `minimumStrikeToDestroy` specified by **Entity**. The `propagateDamage` method should simply reduce the magician's life

points according to the **`damageAmount`** parameter (which is required to be non-negative), and return the number of life points that were deducted. The **`minimumStrikeToDestroy`** method should operate as described in the problem description above

- Override **`toString`** to provide a string representation of a **Magician**. This should be the magician's name, followed by the magician's life points in brackets. For example, referring to the object diagram of Figure 1, **`m1.toString()`** should return the string "**`Ged(300)`**"

(11 marks)

4) Write the class **TransportUnit** in **TransportUnit.java**. **TransportUnit** should extend the **Entity** abstract class
   - A **TransportUnit** should have a field representing the set of entities contained in the transport unit
   - A **TransportUnit** should contain no entities when constructed
   - The method **`add`** should take an **Entity** parameter, and add this entity to the transport unit
   - **TransportUnit** must provide implementations of the abstract methods **`propagateDamage`** and **`minimumStrikeToDestroy`** specified by **Entity**. The **`propagateDamage`** method should reduce the transport unit's life points according to the **`damageAmount`** parameter (which is required to be non-negative), then call **`propagateDamage`** recursively on all entities contained in the transport unit, with an appropriately reduced amount of damage. The method should return the total number of life points deducted as a result of this. The **`minimumStrikeToDestroy`** method should operate as described in the problem description above
   - Override **`toString`** to provide a string representation of a **TransportUnit**. This should be the unit's name, followed by the unit's life points in brackets, followed by "**`transporting: [`**", followed by string representations of all the entities in the transport unit separated by "**`, `**", followed by "**`]`**". For example, referring to the object diagram of Figure 1, **`t2.toString()`** should return the string:

     "**`Shadowfax(350) transporting: [Harry(18), Gandalf(950)]`**"

     although the order of "**`Harry(18)`**" and "**`Gandalf(950)`**" may be different as the members of a transport unit are represented using an unordered set

(28 marks)

5) Implement the **`main`** method in class **GameReport**, in **GameReport.java**. Your method should:
   - Create objects t1, t2, m1, m2, m3, m4, m5 and m6 as shown in Figure 1
   - Print a line of output consisting of the string representation of t1
   - Apply a spell to t1 cast by m6
   - Print a line of output consisting of the string representation of t1
   - Create a new **Magician** object m7 (with a name of your choice) such that m7's strength equals the minimum strike required to destroy t1

- Apply a spell to t1 cast by m7
- Print a line of output consisting of the string representation of t1
- Assert that t1, t2, m1, m2, m3, m4 and m5 are all dead
- Assert that m6 and m7 are alive

If you have implemented the various classes and methods correctly, your main method should produce the following lines of output (though the order in which members of a transport unit are displayed may be different):

```
Falkor(2500) transporting: [Ged(300), Dumbledore(200), Saruman(600),
Shadowfax(350) transporting: [Harry(18), Gandalf(950)]]

Falkor(2100) transporting: [Ged(100), Dumbledore(0), Saruman(400),
Shadowfax(150) transporting: [Harry(0), Gandalf(850)]]

Falkor(0) transporting: [Ged(0), Dumbledore(0), Saruman(0), Shadowfax(0)
transporting: [Harry(0), Gandalf(0)]]
```

(5 marks)

Total for Section A: 50 marks

Bonus Marks

Up to five bonus marks are available. *Bonus marks cannot take your score for Section A above 50*, but they can make up for marks you may lose elsewhere.

- We have specified that a magician's strength should be double the magician's life points. Represent this design decision using an integer constant, in such a way that if we wished to change all magicians' strengths to be a different integer multiple of life points, we could do so very easily
- Similarly, we have specified that the damage inflicted on entities within a transport unit should be half the damage inflicted on the unit. Represent this design decision using an integer constant in such a way that if we wished to change the decay of damage points through all transport units to use a different integer divisor we could do so very easily
- You should find that the implementations of `toString()` in **TransportUnit** and **Magician** have some similarities. Remove this code duplication by overriding `toString()` in Entity and locating the common part of the toString() implementation here
- You should find that the implementations of `propagateDamage()` in **TransportUnit** and **Magician** have some similarities. Remove this code duplication by adding an appropriate new method to **Entity**, locating the common code in this new method, and calling the new method from `propagateDamage()` in both **Magician** and **TransportUnit**

Section B

Problem Description

Tic-tac-toe (also known as **noughts and crosses**), is a game played between two players (**X** and **O**). A game is played on a 3x3 board with **X** and **O** taking turns in making moves. A move is made when a player places his/her mark on an empty position on the board. To win a player needs to place three marks in either a horizonal, vertical or diagonal row, A game terminates when either one of the players has won or the board is full with no winner. For this exercise we will number the board positions **1..9**. *Figure 1* shows an example game won by the first player X.

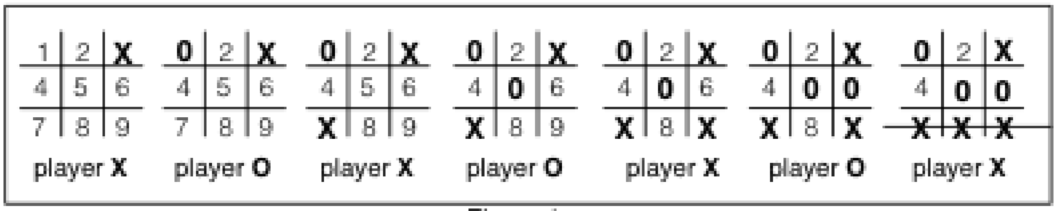


Figure 1

We want to write an *AI* version of this game where a user plays against the computer. The user can choose whether to go first as **X** or second as **O** (see the given **`main`** method in the **TicTacToe** class). For each move the user enters a position (**1..9**) and his/her mark is placed on the board. The computer then computes a *best move*, which is also marked on the board. The game continues until either one player wins or the game is a draw.

The program uses a *game tree*. This is implemented by the Abstract Data Type given in *Figure 2*, which is constructed using a generic linked list of **GameTreeNode.** A **GameTreeNode** includes a **GameTreeItem** containing the current state of play and a reference to a linked list of **GameTreeNode**, for example it could contain the boards of all possible next moves of the other player (and so on).

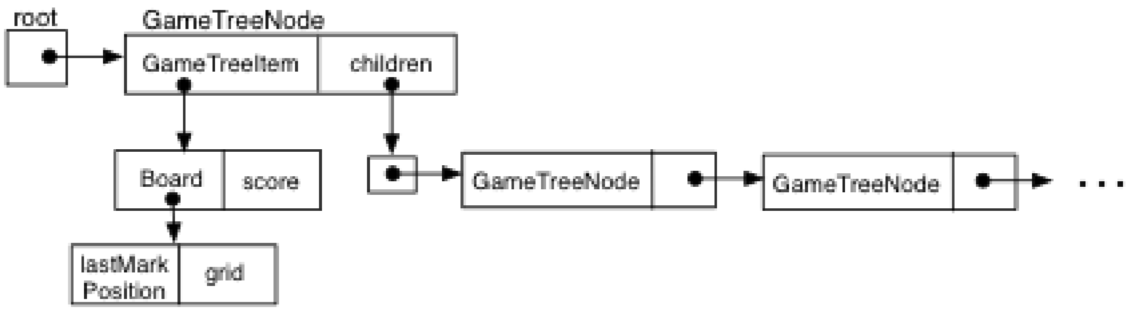


Figure 2: Abstract Data Type

The program computes a best move from the current board (method **`computerOptimalMove`** in the **TicTacToe** class) in the following way: a game tree is constructed with the current board as the root. This is then *expanded* so that it contains all possible future boards of the game, given the root board. Each board in the expanded game tree is then given a score and a next move is chosen randomly among the highest scoring immediate children of the root node.

The expand step is applied to a game tree containing only a root board, and uses the following algorithm:

- It checks that the current board is not finished (using the `isFinished` method in the **Board** class) and gets the mark of the next player (using the given `getTurn` method in the **Board** class),
- for each empty position in the board
  creates a **GameTreeNode** object, with the mark placed on that position,
  records in the board object this position as the last marked position,
  adds this **GameTreeNode** object to the list of children of the current board,
  and then (recursively) call the expand step on this (child) **GameTreeNode** object.

Part of an expanded game tree is given in Figure 3, where the double arrows show the order in which the expand step is recursively applied.

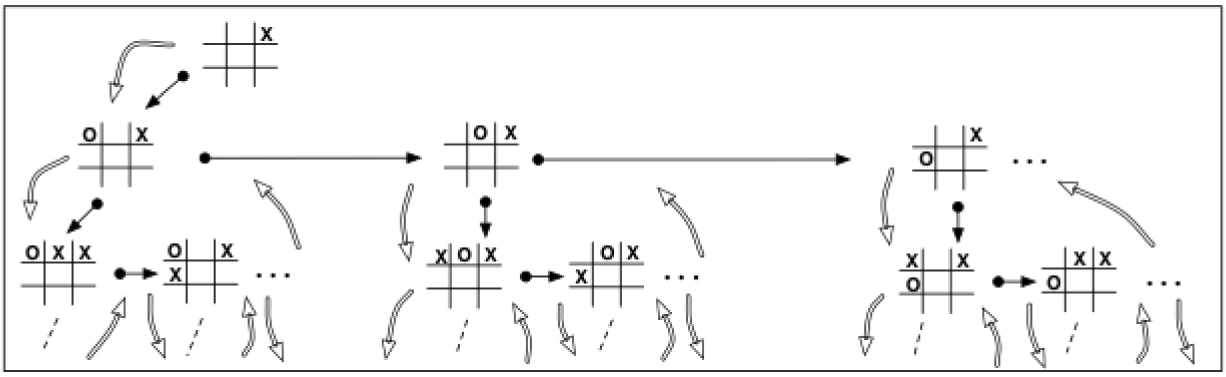


Figure 3: Partial Game Tree, starting from the root

Your task is to complete the provided implementation. In addition to this specification, you should carefully read the commented source files.

You Are Given

- For your convenience, a UML class-diagram describing the architecture of the system. Your implementation will build upon the given design.
- The class **Board** which defines a board and associated methods, including checking if a board is finished, setting a mark, getting the mark of the next player, and getting the mark of the winner.
- The class **TicTacToe** that has the main method for playing the game, and uses an auxiliary method for computing the best move for a given board.
- The interface **GameTreeInterface** that specifies the main access procedures for the Game Tree ADT.
- A partial implementation of the class **GameTree** that realises the interface **GameTreeInterface.**
- A class **GameTreeNode** that includes an inner class **GameTreeItem.**
- The interface **GenericListInterface<T>** that extends **Iterable<T>.**

The partial implementation of a class **GenericList<T>** that realises the interface **GenericListInterface<T>** and the inner class **ListIterator.**

You will find a **SectionB** directory in your Lexis home directory with a **src** subdirectory. Inside **src** is a **noughtsandcrosses** package containing Java files for the above classes and interfaces.

If you work using Eclipse, you will find it already set up with a **SectionB** project for the source files, however the project is slightly out of sync. To sync up, do the following:

- Click on the **SectionB** project and then press **F5** or click **File->Refresh**
- Click on the menu **Project->Clean** and in the new window that pops up, click **Ok**

After this you can work on the project directly in Eclipse.

Your Task

You are asked to write:

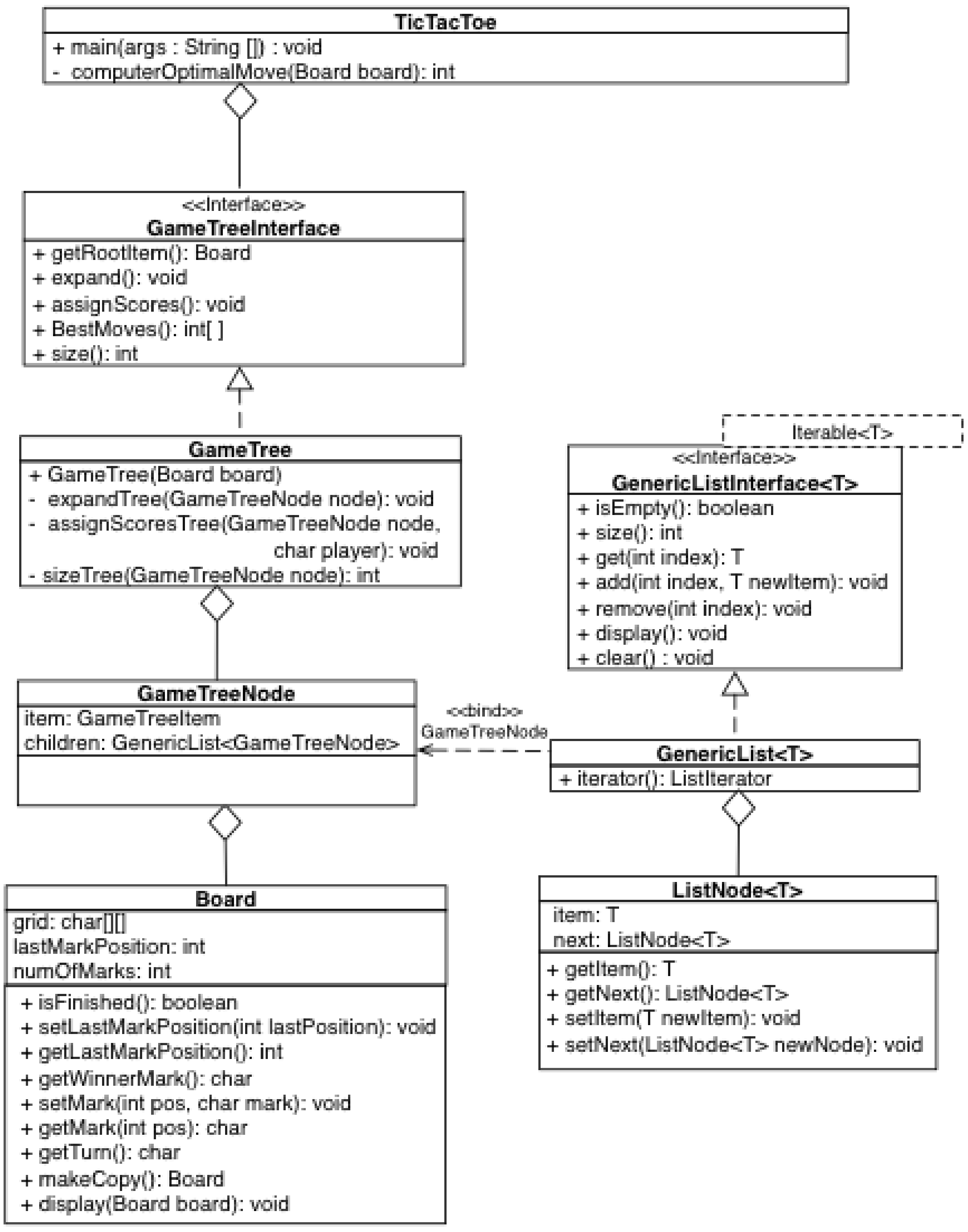
1) The method `add(int pos, T item)` for the class **GenericList<T>**.
(10 marks)

2) The recursive auxiliary method `sizeTree(GameTreeNode node)` in the class **GameTree**, that returns the number of boards stored in a game tree with the given node as root.
(15 marks)

3) The recursive auxiliary method `expandTree(GameTreeNode node)` in the class **GameTree**, that implements the expand step described above.
(25 marks)

Total for Section B: 50 marks

System Architecture


TicTacToe
+ main(args : String []) : void
- computerOptimalMove(Board board): int
<<Interface>>
GameTreeInterface
+ getRootItem(): Board
+ expand(): void
+ assignScores(): void
+ BestMoves(): int[ ]
+ size(): int
GameTree
+ GameTree(Board board)
- expandTree(GameTreeNode node): void
- assignScoresTree(GameTreeNode node, char player): void
- sizeTree(GameTreeNode node): int
GameTreeNode
item: GameTreeItem
children: GenericList<GameTreeNode>
Board
grid: char[][]
lastMarkPosition: int
numOfMarks: int
+ isFinished(): boolean
+ setLastMarkPosition(int lastPosition): void
+ getLastMarkPosition(): int
+ getWinnerMark(): char
+ setMark(int pos, char mark): void
+ getMark(int pos): char
+ getTurn(): char
+ makeCopy(): Board
+ display(Board board): void
Iterable<T>
<<Interface>>
GenericListInterface<T>
+ isEmpty(): boolean
+ size(): int
+ get(int index): T
+ add(int index, T newItem): void
+ remove(int index): void
+ display(): void
+ clear() : void
<<bind>>
GameTreeNode
GenericList<T>
+ iterator(): ListIterator
ListNode<T>
item: T
next: ListNode<T>
+ getItem(): T
+ getNext(): ListNode<T>
+ setItem(T newItem): void
+ setNext(ListNode<T> newNode): void