[L7P2]

Putting up defenses to protect our code

Assertions

Assertions are used to confirm assumptions about the program state. An assertion can be used to express something like 'when the execution comes to this point, the variable v cannot be null'. This implies that if v is null at that point, the code has a bug. If the runtime detects such an 'assertion failure' (think of it as an 'assumption failure'), the runtime will typically take some drastic action such as terminating the execution with an error message. This is because an assertion failure is a confirmation of the code having a bug and the sooner the execution stops, the safer it is.

```
int timeout = Config.getTimeout();      //line 1
assert timeout > 0 ;                    //line 2
setTimeout(timeout);                    //line 3
```

In the Java code given above, the assertion in line 2 verifies the assumption that timeout returned by Config.getTimeout() is greater than 0. The only reason for it to return a zero or lower value in line 1 is the presence of a bug in the system.

It is recommended that assertions be used liberally in the code. If required, assertions can be disabled without modifying the code. For example, 'java -enableassertions HelloWorld' will run HelloWorld with assertions enabled while 'java -disableassertions HelloWorld' will run it without verifying assertions.

Notes:

1. The assertions mentioned above have a different purpose from the Assertions used in unit testing frameworks such as JUnit. Unit testing assertions are located *outside* the functional program and used for explicit testing of the program *before* the system is put into use. The assertions discussed here are located *inside* the functional program and they verify assumptions *while* the system is running.
2. Some runtime environments disable assertions by default. This could create a situation where the developer thinks all assertions are being verified as true while in fact they are not being verified at all. Therefore, remember to enable assertions when you run the program if you want them to be in effect.

Sidebar: Exceptions vs assertions

Exceptions and assertions are two complementary ways of handling errors in software but they serve different purposes.

- The raising of an exception indicates an unusual condition created by the user (e.g. user input an unacceptable input) or the environment (e.g., a file needed for the program is missing).
- An assertion failure indicates the programmer made a mistake in the code (e.g., a null value is returned from a method that is not supposed to return null under any circumstances).

Both assertions and exceptions should be used.

Logging

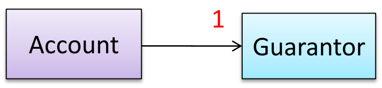
Logging can be useful for troubleshooting problems that exceptions and assertions could not prevent (due to the failure to cater for certain situations). A good logging system records some system information regularly. When bad things happen to a system, their associated log files may provide indications of what went wrong and action can then be taken to prevent it from happening again. This is the same reason why airplanes have 'black boxes'. While it is relatively easy to implement a customized logging system (e.g. by inserting file I/O statements to write to a log file), most programming environments come with logging systems that allow more sophisticated forms of logging. For example, being able to enable and disable logging easily or to change the logging intensity/verbosity (i.e. how much information to record).

Given below is a sample Java code that uses Java's default logging mechanism. When running the code, the logging level can be set to WARNING so that log messages specified as INFO level will not be written to the log file at all.

```
import java.util.logging.*;

public class Foo{
    // Obtain a suitable logger.
    private static Logger logger = Logger.getLogger("Foo");

    public void bar() {
        // log a message at INFO level
        logger.log(Level.INFO, "going to start processing");

        try{
            processInput();
        } catch (Exception ex) {
            //log a message at WARNING level
            logger.log(Level.WARNING, "processing error", ex);
        }
        logger.log(Level.INFO, "end of processing");
    }
}
```

Defensive programming

A defensive programmer codes under the assumption "if we leave room for things to go wrong, they *will* go wrong". Therefore, a defensive programmer proactively tries to eliminate any room for things to go wrong. The following example illustrates this. Consider two entities, Account and Guarantor, with an association as shown in the following diagram:



```
class Account {
    Guarantor guarantor;
    void setGuarantor(Guarantor g){
        guarantor = g;
    }
}
```

Here, the association is compulsory i.e. an Account object should always be linked to a Guarantor. One way to implement this is to simply use a reference variable as above. However, what if someone else in the team used the Account class as:

```
Account  a = new Account();
a. setGuarantor(null);
```

This results in an Account without a Guarantor! In a real banking system, this could have serious consequences! The code here did not try to prevent such a thing from happening. To proactively enforce the multiplicity constraint, a solution is offered as follows:

Make the variable private to prevent the value being changed from outside the class.

```
class Account {
      private Guarantor guarantor;
      pubic Account(Guarantor g){
         if(g==null) stopSystemWithMessage("multiplicity violated. Null Guarantor ");
         guarantor = g;
      }
      public setGuarantor (Guarantor m){
         if(g==null) stopSystemWithMessage("multiplicity violated. Null Guarantor");
         guarantor = g;
      }
      …
}
```

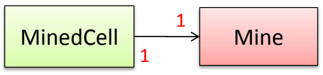
It is not necessary to be 100% defensive all the time. The degree of code defensibility depends on many factors such as:

- How critical is the reliability of the system?
- Will the code be used by programmers other than the author?
- The level of programming language support for defensive programming

The following are some more examples of defensive programming.

Enforcing 1-to-1 associations

Consider the association given below. Here, a MinedCell cannot exist without a Mine and vice versa. The only way to enforce this is by simultaneous object creation. However, in Java and C++, only one object can be created at a time. Given below are two alternatives. Both options violate the multiplicity for a short period of time.



This option forces us to keep a Mine without a MinedCell (until the MinedCell is created).

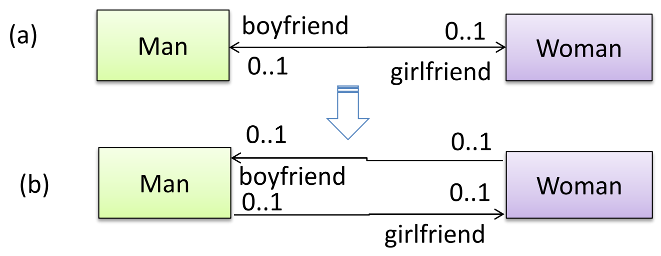
```
class MinedCell {
      private Mine mine;
      pubic MinedCell(Mine m){
         if(m==null) error;
         mine = m;
      }
      …
}
```

```
class MinedCell {
      private Mine mine;
      pubic MinedCell(){
         mine = new Mine();
      }
      …
}
```

This options is more defensive because the Mine is immediately linked to a MinedCell.

Enforcing referential integrity

A bidirectional association in the design (shown in (a)) is usually emulated at code level using two variables (as shown in (b)).



```
class Man {
        Woman girlfriend;
        void setGirlfriend(Woman w){
          girlfriend = w;
        }
        …
}
```

```
class Woman {
        Man boyfriend;
        void setBoyfriend(Man m){
          boyfriend = m;
        }
        …
}
```

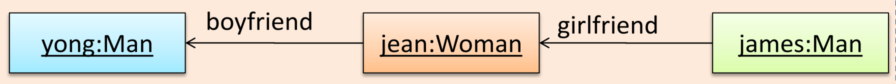
The two classes are meant to be used as shown in (c) below. Now see what happens if the two classes were used as in (d) below. Now James' girlfriend is Jean, while Jean's boyfriend is not James. This situation results as the code was not defensive enough to stop this "love tangle". In such a situation, *referential integrity* has been violated. It simply means there is an inconsistency in object references.

(c)

```
Woman jean; Man james;
…
james.setGirlfriend(jean);
jean.setBoyfriend(james);
```

(d)

```
Woman jean; Man james, yong;
…
james.setGirlfriend(jean);
jean.setBoyfriend(yong);
```



One way to prevent this situation is to implement the two classes as shown below. Note how the referential integrity is maintained.

```
public class Woman {
  private Man boyfriend;

 public void setBoyfriend(Man m) {
      if(boyfriend == m){
          return;
      }
      if (boyfriend != null) {
          boyfriend.breakUp();
     }
     boyfriend = m;
     m.setGirlfriend(this);
 }
   public void breakUp() {
      boyfriend = null;
 }
 …
}
```

```
public class Man{
  private Woman girlfriend;

 public void setGirlfriend(Woman w) {
      if(girlfriend == w){
          return;
      }
      if (girlfriend != null) {
          girlfriend.breakUp();
      }
      girlfriend = w;
      w.setBoyfriend(this);
}
  public void breakUp() {
      girlfriend = null;
  }
 …
}
```

When the following line is executed,

```
james.setGirlfriend(jean);
```

it is ensured that `james` break up with any current girlfriend before he accepts `jean` as the girlfriend. Furthermore, the code ensures that `jean` breaks up with any existing boyfriends and accepts `james` as the boyfriend.

The design-by-contract approach

Suppose an operation is implemented with the behavior specified precisely in the API (preconditions, post conditions, exceptions etc.). When following the defensive approach, the code should first check if the preconditions have been met. Typically, exceptions are thrown if preconditions are violated. In contrast, the *Design-by-Contract* (DbC) approach to coding assumes that it is the responsibility of the caller to ensure all preconditions are met. The operation will honor the contract only if the preconditions have been met. If any of them have not been met, the behavior of the operation is "unspecified".

Languages such as Eiffel have native support for DbC. For example, preconditions of an operation can be specified in Eiffel and the language runtime will check precondition violations without the need to do it explicitly in the code. To follow the DbC approach in languages such as Java and C++ where there is no built-in DbC support, assertions can be used to confirm pre-conditions.

Worked examples

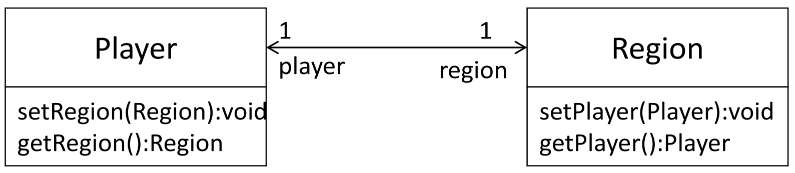
[Q1]

Imagine that we now support the following feature in our Minesweeper game.

Feature id: *multiplayer*

Description: a minefield is divided into mine regions. Each region is assigned to a single player. Players can swap regions. To win the game, all regions must be cleared.

Given below is an extract from our class diagram.



Minimally, this can be implemented like this.

```
class Player{
        Region region;
        void setRegion(Region r) { region = r;}
        Region getRegion() {return region;}
}
//Region class is similar
```

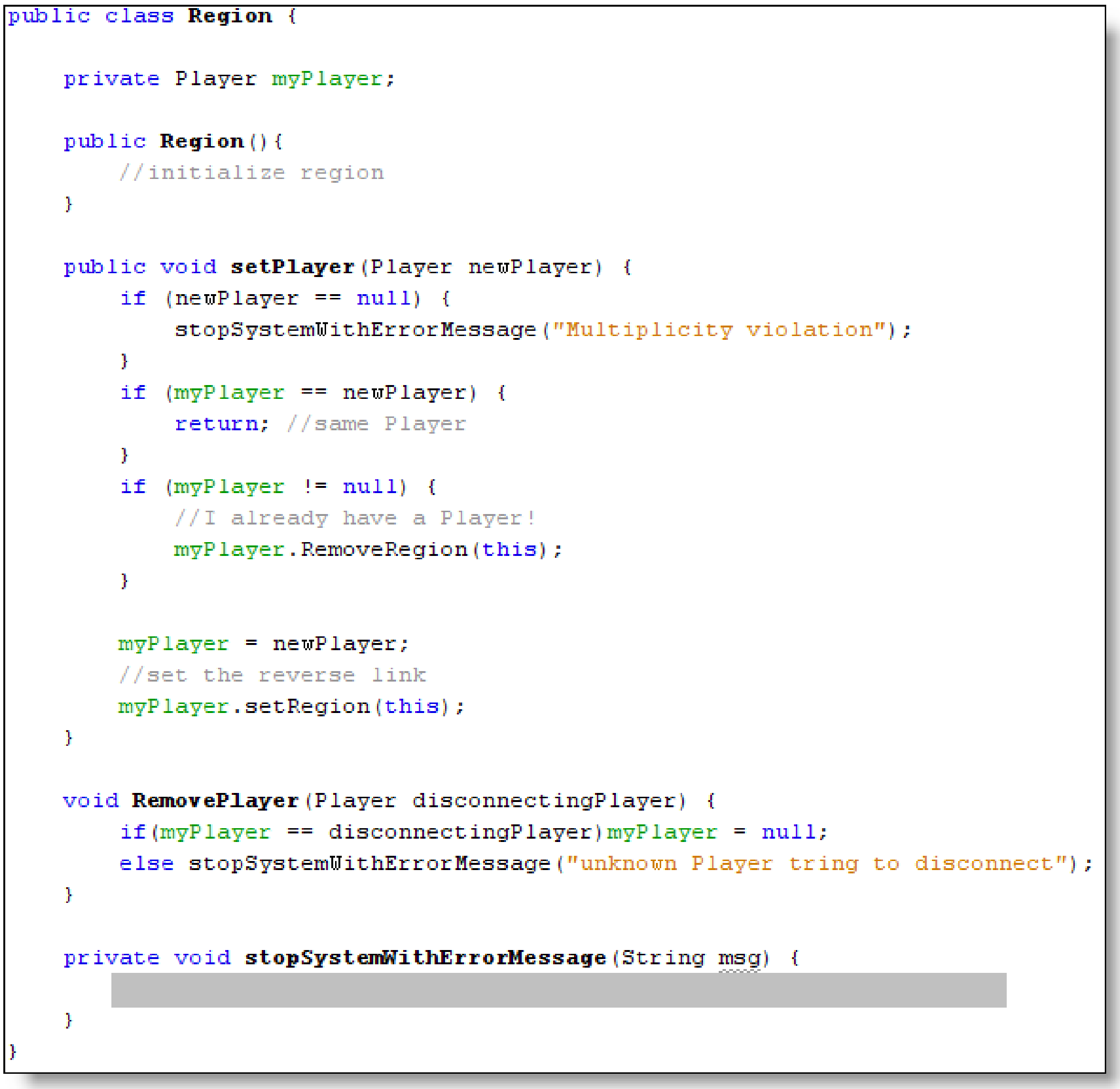
However, this is not very defensive. For example, a user of this class can pass a null to either of the methods, thus violating the multiplicity of the relationship.

Implement the two classes using a more defensive approach. Take note of the bidirectional link which requires us to preserve referential integrity at all times.

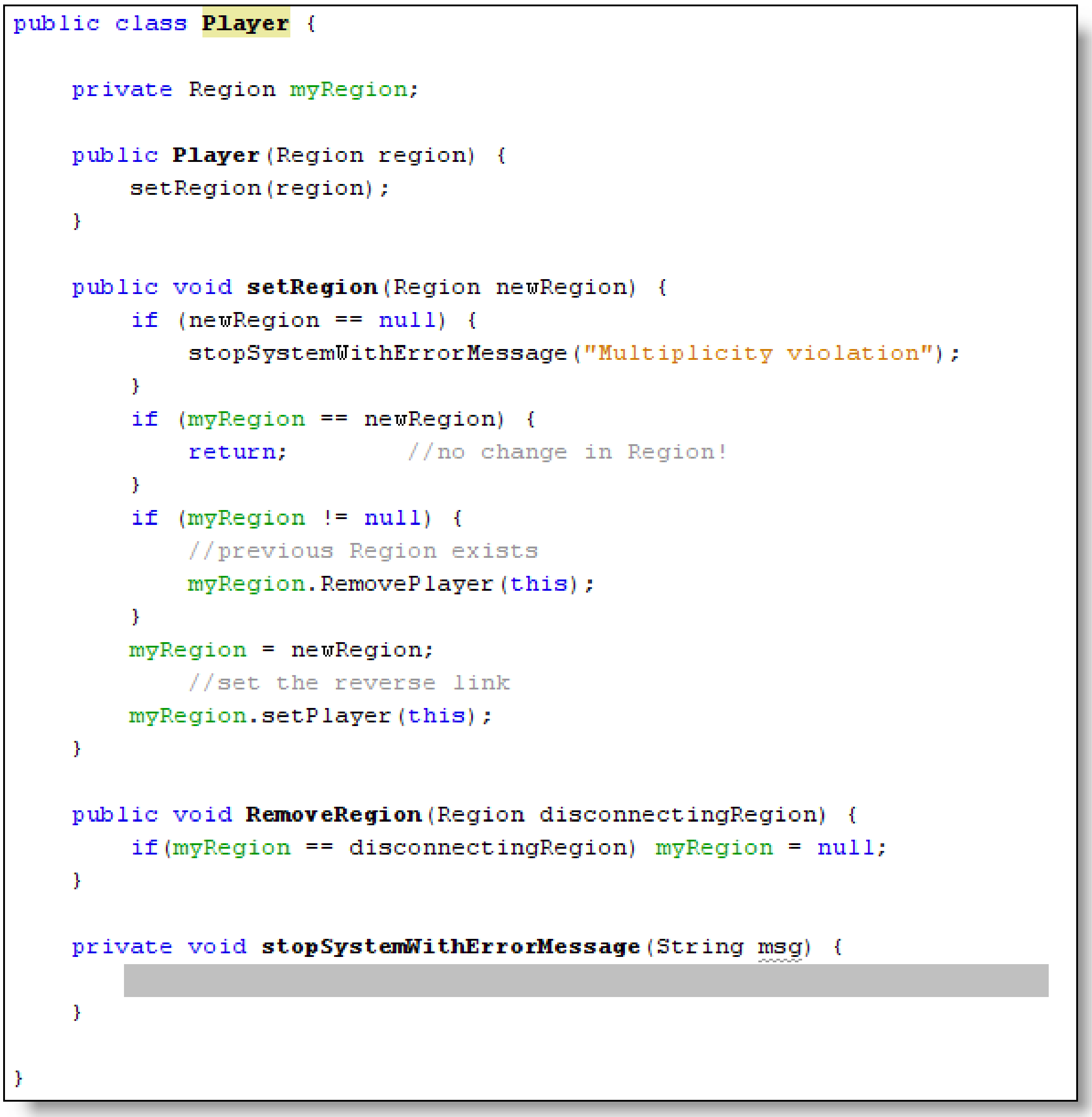
[A1]
In this solution, we assume Regions can be created without Players (note that we cannot be 100% defensive all the time). The usage will be something like this:

```
Region r1 = new Region();
Player p1 = new Player(r1);
Region r2 = new Region();
Player p2 = new Player(r2);
p1.setRegion(r2);
r1.setPlayer(p2);
```

Here are the two classes. Get methods are omitted as they are simple. Note how much extra effort we need to be defensive.



```
public class Region {

    private Player myPlayer;

    public Region(){
        //initialize region
    }

    public void setPlayer(Player newPlayer) {
        if (newPlayer == null) {
            stopSystemWithErrorMessage("Multiplicity violation");
        }
        if (myPlayer == newPlayer) {
            return; //same Player
        }
        if (myPlayer != null) {
            //I already have a Player!
            myPlayer.RemoveRegion(this);
        }

        myPlayer = newPlayer;
        //set the reverse link
        myPlayer.setRegion(this);
    }

    void RemovePlayer(Player disconnectingPlayer) {
        if(myPlayer == disconnectingPlayer)myPlayer = null;
        else stopSystemWithErrorMessage("unknown Player tring to disconnect");
    }

    private void stopSystemWithErrorMessage(String msg) {

    }
}
```

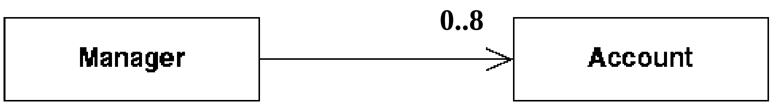


```java
public class Player {

    private Region myRegion;

    public Player(Region region) {
        setRegion(region);
    }

    public void setRegion(Region newRegion) {
        if (newRegion == null) {
            stopSystemWithErrorMessage("Multiplicity violation");
        }
        if (myRegion == newRegion) {
            return;         //no change in Region!
        }
        if (myRegion != null) {
            //previous Region exists
            myRegion.RemovePlayer(this);
        }
        myRegion = newRegion;
            //set the reverse link
        myRegion.setPlayer(this);
    }

    public void RemoveRegion(Region disconnectingRegion) {
        if(myRegion == disconnectingRegion) myRegion = null;
    }

    private void stopSystemWithErrorMessage(String msg) {

    }

}
```

Note that the above code stops the system when the multiplicity is violated. Alternatively, we can throw an exception and let the caller handle the situation.

[Q2]
For the Manager class shown below, write an addAccount( ) method that

- restricts the maximum number of Accounts to 8
- avoids adding duplicate Accounts



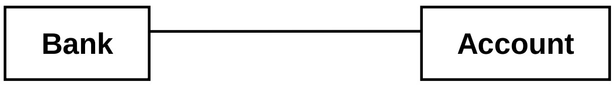
[A2]

```java
import java.util.*;
public class Manager {

    private ArrayList<Account> theAccounts ;
```

```
        public void addAccount(Account acc) throws Exception{
            if (theAccounts.size( ) == 8)
                throw new Exception ("adding more than 8 accounts");

            if (!theAccounts.contains(acc)) {
                theAccounts.add(acc) ;
            }
        }

        public void removeAccount(Account acc) {
            theAccounts.remove(acc) ;
        }
    }//end class
```

[Q3]

Implement this bidirectional association. Note that the Bank uses accNumber attribute to uniquely identify an Account object. Assume the Bank class is responsible for maintaining the links between objects.



[A3]

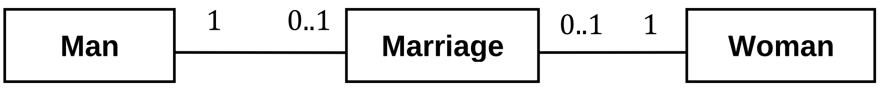
The code below contains a method in the Bank class to create an account; the bank field in the new account is thereby filled by the bank creating it.

We assume that once an Account has been assigned to one Bank, it cannot be assigned to a different Bank. Once the Account is removed from the Bank, it will not be used any more (hence, no need to remove the link from Account to Bank).

```
public class Account {
  private int accNumber ;
  private Bank theBank ;

  public Account(int n, Bank b) {
    accno = n ;
    theBank = b ;
  }
  public int getNumber() {
    return accNumber;
  }
  public Bank getBank() {
    return theBank ;
  }
}//end class-----------------------------------------

import java.util.*;

public class Bank {

    private HashMap<Integer,Account> theAccounts =
                              new HashMap <Integer,Account> () ;

    public void createAccount(int n) {
        addAccount(new Account(n, this)) ;
    }
    public void addAccount(Account a) {
        theAccounts.put(a.getNumber(), a);
    }
    public void removeAccount(int accNumber) {
        theAccounts.remove(accNumber);
    }
    public Account lookupAccount(int accNumber) {
        return theAccounts.get(accNumber);
    }
}//end class
```

[Q4]

Implement the classes with appropriate references and operations to establish the association among the classes. Follow the defensive coding approach. Let the Marriage class handle setting/removal of reference.



[A4]

```
public class Marriage {
      private Man husband = null;
      private Woman wife = null;

      // extra information like date etc can be added

      public Marriage(Man m, Woman w) throws Exception {
            if (m == null || w == null) {
                  throw new Exception("no man/woman");
            }
            if (m.isMarried() || w.isMarried()) {
                  throw new Exception("already married");
            }
            husband = m;
            m.enterMarriage(this);
            wife = w;
            w.enterMarriage(this);
      }

      public Man getHusband() throws Exception {
        if(husband == null) {
               throw new Exception("error state");
        }else {
               return husband;
        }
    }

      public Woman getWife() throws Exception {
        if(wife == null) {
               throw new Exception("error state");
        } else {
               return wife;
        }
    }

      // removal of both ends of 'Marriage'
      public void divorce() throws Exception {
            if (husband==null || wife==null) {
                   throw new Exception("no marriage");
            }
          husband.removeFromMarriage(this);
```

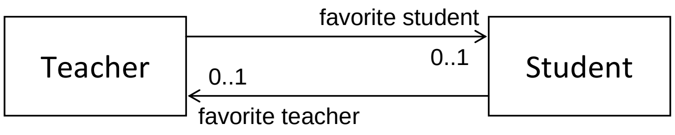
```
            husband = null;
            wife.removeFromMarriage(this);
            wife = null;
    }
}// end class
```

[Q5]

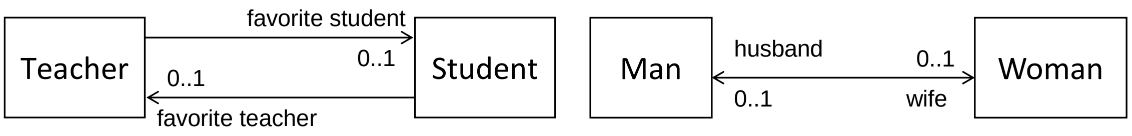
(a) Is the code given below a defensive translation of the associations shown in the class diagram? Explain your answer.

```
class Teacher{
  private Student favStudent;
  void setFavStudent(Student s){
    favStudent = s;
  }
}
```

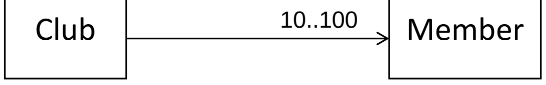
```
class Student{
  private Teacher favTeacher;
  void setFavTeacher(Teacher t){
    favTeacher = t;
  }
}
```



(b) In terms of maintaining referential integrity in the implementation, what is the difference between the following two diagrams?



(c) Show a defensive implementation of the remove(Member m) of the Club class given below.



[A5]

(a) Yes. Each links is mutable and unidirectional. A simple reference variable is suitable to hold the link.

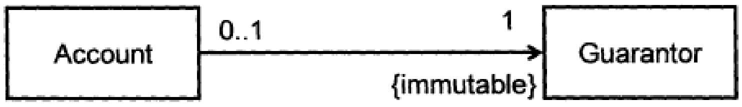
Teacher class can be made even more defensive by introducing a resetFavStudent() method to unlink the current favorite student from a teacher. In that case, setFavStudent(Student) method should not accept null. This approach is more defensive because it prevents a null value being passed to setFavStudent(Student) by mistake and being interpreted as a request to delink the current favorite student from the Teacher object.

(b) First diagram has unidirectional links. Second has a bidirectional link. RI is only applicable to the second.

(c)

```
void removeMember(Member m){
        if (m==null) throw exception(“this is null, not a member!”);
        else if(member_count == 10) throw exception(“we need at least 10 members to survive!”);
        else if(!isMember(m)) throw exception (“this fellow is not a member of our club!”);
        else members.remove(m); //members is a data structure such as ArrayList
}
```

[Q6]
Give a suitable defensive implementation to the Account class in the following class diagram. Note that “{immutable}” means once the association is formed, it cannot be changed.



A6]

```
class Account {
      private Guarantor myGuarantor; // should not be public

      public Account(Guarantor g){
            if (g==null) {
                  haltWithErrorMessage(“Account must have a guarantor”);
            }
            myGuarantor = g;
      }
      // there should not be a setGuarantor method
}
```

-- End of handout --