OBJECTRON

SWOP 2012-2013

ITERATION 1

Contents

1 Introduction

For the course *Software Ontwerp: project*, you will develop a board game in which players move on a grid with the goal to reach a target before the opponents reach their targets.

In Section 2, we explain how the project is organized, discuss the quality requirements for the software you will develop and how we evaluate the solutions. In Section 3, we explain the problem domain of the application, followed by a diagram of the domain model in Section **??**. We describe the use cases in detail in Section 5.

2 General Information

In this section, we explain how the project is organized, what is expected of the software you will develop and the report you will write.

2.1 Team Work

For this project, you will work in groups of four. Each group is assigned an advisor from the educational staff. If you have any questions regarding the project, you can contact your advisor and schedule a meeting. When you come to the meeting, you are expected to prepare specific questions and have sufficient design documentation available. *If the design documentation is not of sufficient quality, the corresponding question will not be answered.* It is your own responsibility to organize meetings with your advisor and we advise to do this regularly. Experience from previous years shows that groups that regularly meet with their advisors produce a higher quality design.

If there are problems within the group, you should immediately notify your advisor. Do not wait until right before the deadline or the exam!

To ensure that every team member practices all topics of the course, a number of roles are assigned to team members at the start of each iteration (or shortly thereafter in case of the first iteration). A team member that is assigned a certain role will give the presentation or demo corresponding to that role at the end of the iteration. That team member, however, is *not supposed to do all of the work concerning his task*! But he must take a leading role in that activity, and be able to answer most questions on that topic during the evaluation.

The following roles will be assigned round-robin. The iterations during which a role is used is shown between square brackets.

Lead Designer [1…6] The lead designer has an active role in making the design of your software. In addition to knowing the design itself, he knows *why* these design decisions were taken, and what the alternatives are.

Lead Tester [1…6] The lead tester has an active role in planning, designing, and writing the tests for the software. Know which kinds of tests are needed (scenario, unit) for testing various parts of the software, which concrete tests are chosen (success cases, failure cases, corner cases,…), and which techniques are used (mocking, stubbing, …).

Lead Domain Modeler [3…6] The lead domain modeler has an active role in defining the domain model. He knows which parts of the domain are relevant for the application.

2.2 Iterations

The project is divided into six iterations. In the first iteration, you will implement the base functionality of the software. In subsequent iterations, new functionality will be added and/or existing functionality will be changed. For this year, the deadlines are:

- Iteration 1: 18/02/2013 - 01/03/2013 at 18:00
- Iteration 2: 04/03/2013 - 15/03/2013 at 18:00

- Iteration 3: 18/03/2013 - 29/03/2013 at 18:00
- Iteration 4: 15/04/2013 - 26/04/2013 at 18:00
- Iteration 5: 29/04/2013 - 10/05/2013 at 18:00
- Iteration 6: 13/05/2013 - 24/05/2013 at 18:00

2.2.1 Late Submission Policy

If the zip file is submitted N minutes late, with $0 <= N <= 240$, the score for all team members is scaled by $(240 - N)/240$ for that iteration. For example, if your solution is submitted 30 minutes late, the score is scaled by 87.5%, so the maximum score for the iteration is 1.75. If the zip file is submitted more than 4 hours late, the score for all team members is 0 for that iteration.

2.2.2 When Toledo Fails

If the Toledo website is down -- and *only* if Toledo is down -- at the time of the deadline, submit your solution as follows. First, make sure that the departmental subversion repository of your group is up to date. Then, send an email to both Prof. Bart Jacobs and your project advistor to report the problem and provide the url of your repository. We will then do a checkout based on the *timestamp* of the deadline. This means that any changes after the deadline will not be seen by us.

Even if you use a different repository during your project, we will only do a checkout from departmental repositories in case of problems with Toledo. Therefore, be sure to test that you have properly configured access to your repository.

2.3 The Software

The focus of this course is on the *quality* (maintainability, extensibility, stability, readability,...) of the software you write. We expect you to use the development process and the techniques that are taught in this course. You are also required to provide extensive class and method documentation, as taught in previous courses.

When designing and implementing your system, you should use a *defensive programming style*. This means that the *client* of the public interface of a class cannot bring the objects of that class, or objects of connected classes, in an inconsistent state. You can optionally use tools such as *Contracts For Java*, and *NonNullCheckWeaver* to enabled runtime checking of pre- and postconditions. These tools can help you to reduce the workload for testing and debugging. Instructions to use these tools are given in Appendix B and Appendix C.

Unless explicitly stated in the assignment, you do not have to take into account persistent storage, security, multi-threading, and networking. If you have doubts about other non-functional concerns, please ask your advisor.

2.4 User Interface

We provide a library that allows you to draw the grid of the game board and draw objects at certain grid locations. You are encouraged to use this library, but it is not mandatory. You are allowed to create your own visualization code, but the amount of time allocated for each iteration does *not include* the workload for such custom visualization code. You will not gain any points for making a nicer user interface. You will, however, lose many points if the quality (design, testing, ...) of your software is lacking because you spent too much time on the user interface.

2.5 Testing

All functionality of the software should be tested. For scenario tests, this means that not only success scenarios should be tested, but also scenarios for all kinds of faulty user input. Similarly, in addition to confirming a few cases of desired behavior, unit tests should try to break class invariants and postconditions.

Tests should have good coverage, i.e. a testing strategy that leaves large portions of a software system untested is of low value. Several tools exist to give a rough estimate of how much code is tested. One such tool is Eclemma[1]. If this tool reports that only 60% of your code is covered by tests, this indicates there may be a serious problem with (the execution of) your testing strategy. However, be careful when drawing conclusions from both reported high coverage and reported low coverage (understand why you should be careful).

The lead tester is expected to use a coverage tool and briefly report the results during the evaluation of the iteration.

2.6 What You Should Hand In

You hand in an electronic ZIP-archive via Toledo. **The archive contains the items below and follows the structure defined below**:

- `groupXX` (where XX is your group number (e.g. 01, 12, ...))
  - `design`: a folder containing **all** your design diagrams as image files (PDF, PNG, JPG, ...), including those not used in the presentation
  - `doc`: a folder containing the Javadoc documentation of your entire system
  - `src`: a folder containing your source code
  - `presentation.pdf`: a PDF version of your presentation
  - `system.jar`: an executable JAR file of your system

When including your source code into the archive, make sure to *not include files from your version control system*. If you use subversion, you can do this with the the `svn export` command, which removes unnecessary repository folders from the source tree.

Make sure you choose relevant file names for your analysis and design diagrams (e.g. `SSDsomeOperation.png`). You do **not** have to include the project file of your UML tool, only the exported diagrams.

We should be able to start your system by executing the JAR file with the following command: `java -jar system.jar`.

2.7 Evaluation

After iterations 1-5 there will be an intermediate evaluation of your solution. An intermediate evaluation consists of a presentation about the design and the testing approach, accompanied by a demo of the tests. The final part of the intermediate evaluation consists of a presentation of the testing approach for the next iteration.

The intermediate evaluation of an iteration will cover only the part of the software that was developed during that iteration. Before the final exam, the *entire* project will be evaluated. It is your own responsibility to process the feedback, and discuss the results with your advisor.

2.7.1 Presentation Of The Current Iteration

The main part of the presentation should cover the design. The motivation of your design decisions *must* be written in terms of GRASP principles. Use the appropriate design diagrams to illustrate how the most important parts of your software work. Your presentation should cover the following elements. Note that these are not necessarily all separate sections in the presentation.

[1]http://www.eclemma.org

1. A discussion of the high level design of the software.
2. A more detailed discussion of the parts that you think are the most interesting in terms of design.
3. Which GRASP and design patterns you used.
4. A discussion of the extensibility of the system. Briefly discuss how your system can deal with a number of change scenarios (e.g. extra constraints, additional domain concepts,...).
5. A discussion of the testing approach used in the current iteration.

Your presentation should not consist of slides filled with text, but of slides with clear design diagrams and keywords or a few short sentences. The goal of giving a presentation is to communicate a message, not to write a novel. All design diagrams should be *clearly readable* and use the correct UML notation. It is therefore typically a bad idea to create a single class diagram with all information. Instead, you can for example use an overview class diagram with only the most important classes, and use more detailed class diagrams to document specific parts of the system. Similarly, use appropriate interaction diagrams to illustrate the working of the most important (or complex) parts of the system.

2.7.2 Planning Tests

The evaluation of an iteration is planned between 2 and 5 days after that iteration. Immediately after an iteration ends, the assignment for the next iteration is released. The lead tester for the new iteration then starts planning and/or writing the *scenario* tests for the new iteration. The tests should compile, and they should all fail, so that it is clear that the implementation still has to be made (unless of course you manage to already implement one or more scenarios in that short period). It is of course allowed to change these tests throughout the iteration, but they should already provide a clear criterium to decide when the next iteration is finished. He will present his work at the end of the evaluation of the previous iteration.

2.8 Peer/Self-assessment

In order for you to critically reflect upon the contribution of each team member, you are asked to perform a peer/self-assessment within your team. For each team member (including yourself) and for each of the criteria below, you must give a score on the following scale: *poor/below average/adequate/above average/excellent.* The criteria to be used are:

- Design skills (use of GRASP and DESIGN patterns, ...)
- Coding skills (correctness, defensive programming, documentation,...)
- Testing skills (approach, test suite, coverage, ...)
- Collaboration (teamwork, communication, commitment)

In addition to the scores themselves, we expect you to briefly explain for each of the criteria why you have given these particular scores to each of the team members. The total length of your evaluation should not exceed 1 page.

Please be fair and to the point. Your team members will not have access to your evaluation report. If the reports reveal significant problems, the project advisor may discuss these issues with you and/or your team. Please note that your score for this course will be based on the quality of the work that has been delivered, and not on how you are rated by your other team members.

The deadline for the peer/self-assessment for each iteration is the Sunday after the deadline for that iteration at 18:00. Submit your peer/self-assessment by e-mail to both Prof. Bart Jacobs and your project advisor, using the following subject:

[SWOP] peer-/self-assessment of group $groupnumber$ by $firstname$ $lastname$

3 Objectron

The domain of this year's project is a game that is played on a grid. Each of two players try to reach their target before the other player does.

3.1 Actions

The game is turn-based. During each turn, a player is allowed to perform 3 individual actions. The kind of each action can be chosen by the player. The following kinds of actions are supported:

1. The first kind of action is to move.
   (a) During a single action, a player can move one square in any direction (vertical, horizontal, and diagonal).
   (b) No two players can occupy the same square at the same time.
   (c) A player cannot occupy a square that is occupied by a wall.
   (d) A player cannot leave the grid.
   (e) When a player moves, he leaves a light trail behind on his previous square. The light trail remains active during 2 additional actions. After that, the light trail disappears. Therefore, the maximum length of the light trail is 3 squares.
   (f) A player cannot cross any light trail: he cannot enter a square that contains a light trail, and he cannot pass through a diagonal light trail.
   (g) A player is not allowed to end the turn on the square that he started the turn on.
   (h) If a player cannot move during his turn, he is trapped and loses the game.
   (i) If a player reaches the starting position of his opponent, he wins the game.
2. The second kind of action is to pick up an item.
   (a) A player can only pick up an item that is placed on the same square as the player.
   (b) A player can carry at most six items.
3. The third kind of action is to use an item.
   (a) A player can only use an item that is in his inventory.
4. The fourth kind of action is to end the turn. In this case, an empty action is performed for each remaining action in the turn.

3.2 The Grid

The grid on which the game is played has a rectangular shape for which the size can be configured when the game is started. The minimum grid size is 10x10. Player 1 starts on the bottom left corner of the grid, player 2 starts on the top right corner. The finish for each player is the starting location of the other player.

3.2.1 Walls

At the start of the game, walls are placed on the grid. A wall forms a barrier that a player cannot cross. To ensure that the game is still playable, placement of the walls is constrained.

1. Walls are placed either horizontal or vertical, and have a width of 1 square.
2. The direction of each wall is chosen randomly.
3. The minimal length of a wall is two squares.

4. The maximum length (in terms of the numbers of squares that it covers) of a vertical (horizontal) wall is 50% of the vertical (horizontal) size of the grid, rounded up to an integer value.
5. The number of walls is chosen randomly.
6. There is at least 1 wall.
7. At most 20% of the squares in the grid, rounded up to an integer value, is covered by a wall.
8. Walls can neither touch each other nor intersect. *If the largest vertical wall in Figure 1 is shifted 1 square up, it touches the horizontal wall. If we extend the largest vertical wall upward by 3 squares, and the horizontal wall to the right by 2 squares, the walls intersect.*
9. A wall cannot cover the starting position of a player.

3.3 Items

3.3.1 Light Grenades

Light grenades are items that can be picked up and used by a player.

1. When a player uses a light grenade, the light grenade is removed from his inventory and placed on the current square of the player. Initially, the light grenade is inactive. The light grenade becomes active when the player leaves the square.
2. An active light grenade is invisible and cannot be picked up.
3. A player cannot use multiple light grenades on the same square.
4. When any player enters a square with an active light grenade, it explodes. The player that drove over the light grenade is blinded and loses his next 3 actions
5. An exploded light grenade is not active.
6. Active and exploded light grenades cannot be picked up.

At the start of the game a number of inactive light grenades is placed on the grid.

1. The placement of light grenades is random.
2. At the start of the game, 5% of the squares contains a light grenade, rounded up to an integer value.
3. A light grenade cannot be placed on a wall.
4. There can be at most a single light grenade on each square.
5. The starting position of a player cannot contain a light grenade.
6. For each player, at least one light grenade is placed in the area of 3x3 squares that covers the starting position of that player.

4 Domain Model

Figure 2 shows the domain model of Objectron.

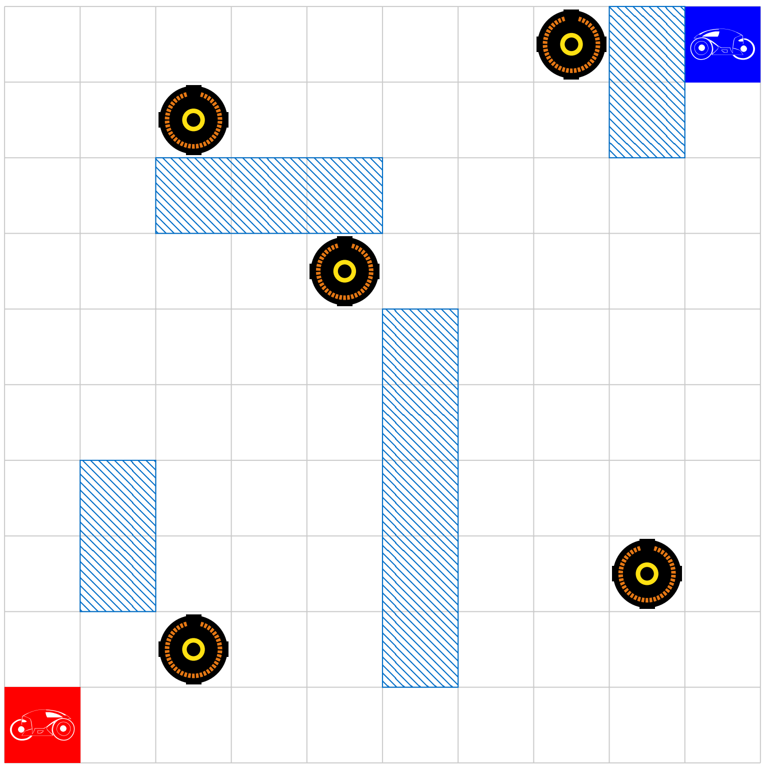

Figure 1: Example of an initial game setup.

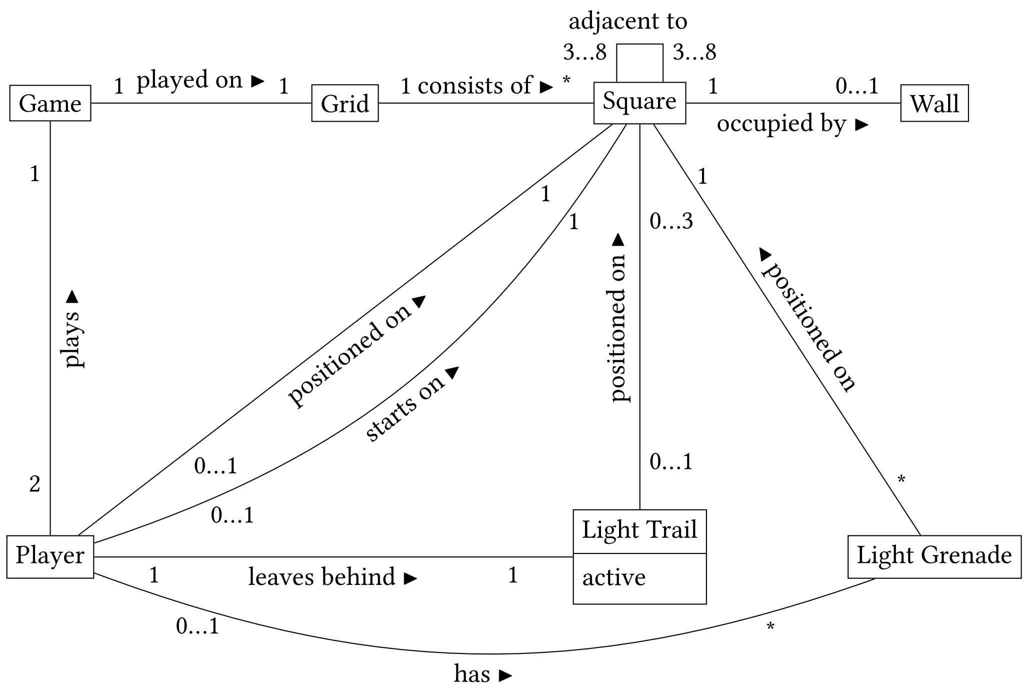

Figure 2: Domain model.

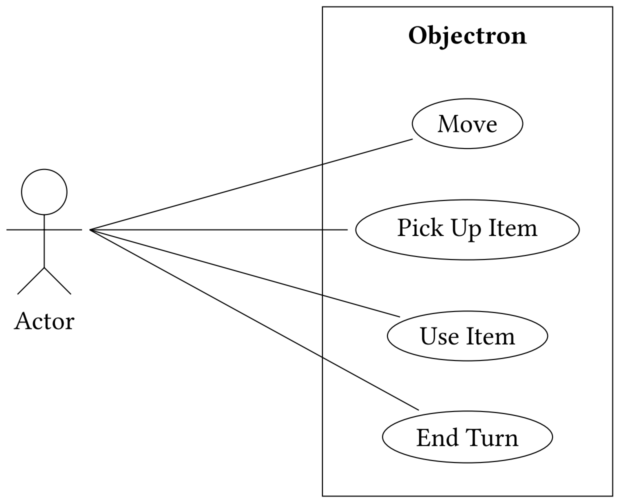

Figure 3: Use case diagram for Objectron.

5 Use Cases

Figure 3 shows the use case diagram for Objectron. The following sections describe the various use cases in detail.

5.1 Use Case: Move

Primary Actor: The player whose turn it is.

Precondition: It is the player's turn, and the player has performed less than 3 actions in the current turn.

Success End Condition: The player has moved.

Main Success Scenario:

1. The player selects a direction in which he wants to move.
2. The system moves the character of the player 1 square in the selected direction.
3. The system adds 1 to the number of actions that the player has performed during this turn.

Alternate Flow:

1. (a) The user selects a direction that would position his character on a square that is covered by a wall.
2. The use case ends.

Alternate Flow:

1. (b) The user selects a direction that would move his character off the grid.
2. The use case ends.

Alternate Flow:

1. (c) The user selects a direction that would position his character on a square that is covered by the other player.
2. The use case ends.

Alternate Flow:

1. (d) Moving in the selected direction would let the character of the player cross a light trail.
2. The use case ends.

Alternate Flow:

3. (a) The player has reached the starting position of the other player.
4. The player wins the game.

5.2 Use Case: Pick Up An Item

Primary Actor: The player whose turn it is.

Precondition: It is the player's turn, and the player has performed less than 3 actions in the current turn.

Precondition: If the player has not yet moved during this turn, he must have more than one action left.

Success End Condition: The user receives one of the items on the current square.

Main Success Scenario:

1. The user indicates he wants to pick up an item.
2. The system presents a list of items on the current location.
3. The user selects an item from the list.
4. The system removes the item from the gameboard at the position of the player and puts it in the inventory of the player.
5. The system adds 1 to the number of actions that the player has performed during this turn.

Alternate Flow:

1. (a) There are no items on the current square of the player.
2. The use case ends.

Alternate Flow:

1. (b) The user already has six items in his inventory.
2. The use case ends.

5.3 Use Case: Use An Item From The Inventory

Primary Actor: The player whose turn it is.

Precondition: It is the player's turn, and the player has performed less than 3 actions in the current turn.

Precondition: If the player has not yet moved during this turn, he must have more than one action left.

Success End Condition: The user has used an item.

Main Success Scenario:

1. The player selects an item from the inventory.
2. The user indicates he wants to use an item.

3. The system lets the character of the player use the item.
4. The system adds 1 to the number of actions that the player has performed during this turn.

Alternate Flow:

1. (a) There are no items in the inventory of the player.
2. The use case ends.

Alternate Flow:

3. (a) The user decides he no longer wants use an item and cancels the use case.
4. The use case ends.

5.4 Use Case: End Turn

Precondition: It is the player's turn, and the player has performed less than 3 actions in the current turn.

Primary Actor: The player whose turn it is.

Success End Condition: The turn of the player ends.

Main Success Scenario:

1. The user indicates that he wants to ends his turn.
2. The system asks for confirmation and warns the user that we will lose the game if he did not yet move during this turn.
3. The user confirms
4. The system lets the character of the player perform an empty action for each remaining action in this turn.

Alternate Flow:

3. (a) The player does not confirm
4. The use case ends.

Good luck!
The SWOP Team members

A Tools

In the following two appendices, we describe how you can use the *notnullcheckweaver* and *Contracts For Java* tools to improve the robustness of your software and save you work.

The tools that we describe in this document perform pre- and postcondition tests at runtime. The first tool, *notnullcheckweaver*, uses annotations to easily specify which values are allowed to be null, and which are not. The second tool, *Contracts For Java* (cofoja), allows you to specify pre- and postconditions in annotations.

Both tools use Java agents that insert code which throws an exception as soon as a problem is detected. This allows you to immediately pinpoint precisely which code causes the problem.

Using these tools saves you work when writing runtime tests and writing test cases for you program. We will explain this further after we have explained the annotions used by these tools. **However, your program must still work correctly if the Java agent is not active!**. You can assume that the agents are active in the testing phase, but you cannot assume that they are active when the user runs the program.

If a condition that must hold *does not depend* on data that is obtained from an external source (input from the user of the program, a network connection, a file, ...), then these annotations can be used. Annotations can be used here because any violations of these conditions are the result of programming errors. Such errors should be detected during the testing phase, when the agents are active.

If a condition that must hold *does depend* on data from an external source, then that data must be verified explicitly by the program. You cannot assume that for example the user of a program enters a valid ID number. You also cannot rely on annotations to check whether the data is valid because the agent may not be active.

Listing 1: "The name of a Person cannot be null."

```
import notnullcheckweaver.NotNull;

public class Person {
  @NotNull private String _name;
  private Person _bestFriend;

  public Person(String name, Person bestFriend) {
    _name = name;
    _bestFriend = bestFriend;
  }
}

public static void main(String[] args) {
  Person bob = new Person("Bob",null)
  Person alice = new Person("Alice",bob);
  // The following constructor call will fail.
  Person eve = new Person(null,alice);
}
```

B NotNullCheckWeaver: Checking for Null References

In most cases, method parameters, fields of objects, and return values of methods should not be null. In a traditional defensive programming style, this means that you must write lots of checks in your code to enforce this. The NotNullCheckWeaver tool allows you to easily use annotations to specify which values are allowed to be null, and which are not. The tool includes a runtime Java agent that enforces the behavior specified by the annotations at runtime.

B.1 Null Check Annotations

Annotations are used to specify whether or not the value of a field or parameter, or the return value of a method can be null. The `@NotNull` annotation specifies that null is not a valid value. The `@Nullable` annotation specifies that null is a valid value.

By default, null is a valid value. This default can be overridden for individual fields, parameters, or return values, for all such elements within a class, and for all such elements within a package or one of its subpackages. The rule for a particular element is determined by the *most specific annotation* for that element.

In the remainder of this section, we show how these annotations can be used.

B.1.1 Field or Parameter Annotations

In the code in Figure 1 the `@NotNull` annotation ensures that the name of a person cannot be null. The tool will insert code to verify that the result of every read of `_name` returns a non-null result. The best friend of a person, however, can still be null. When the constructor of `Person` is invoked with a null name, the assignment to the `_name` field will cause an exception. This is the case when `eve` is constructed in the `main` method.

The tools will also verify whether or not an object is properly initialized. When the `_name` field is still null after the execution of the constructor. In such a case, the object is not properly initialized and the constructor must be fixed. This is illustrated in Figure 2.

Listing 2: "A bug in the constructor."

```java
import notnullcheckweaver.NotNull;

public class Person {
  @NotNull private String _name;
  private Person _bestFriend;

  public Person(String name, Person bestFriend) {
    _bestFriend = bestFriend;
    // At this point, an exception will be thrown
    // because the _name field is still null.
  }
}

public static void main(String[] args) {
  // The following constructor call will fail
  // because of a bug in the constructor.
  Person bob = new Person("Bob",null)
}
```

Listing 3: "Using method annotations."

```
public class PhoneBook {
  @NotNull private Map<Person,Integer> _map =
     new Map<Person,Integer>();
  @NotNull private Map<Integer,Person> _reverseMap =
     new Map<Integer,Person>();

  public void add(@NotNull Person person, @NotNull Integer phoneNumber)
    {_map.put(person,phoneNumber);
    _reverseMap.put(phoneNumber,person);}

  public @NotNull Integer phoneNumber(@NotNull Person person)
    {// This may be a bug. This method should explicitly throw an exception.
    return _map.get(person);}

  public @NotNull Person phoneOwner(@NotNull Integer phoneNumber)
    {// This is a bug. This method should explicitly throw an exception.
     return _reverseMap.get(person);}

  public static void main(String[] args)
    PhoneBook book = new PhoneBook();
    Person bob = new Person("Bob",null)
    Person alice = new Person("Alice",null)
    book.add(bob,123456789);
    // the following method call will cause an exception
    // because alice is not in the map of the phone book.
    Integer numberOfAlice = book.phoneNumber(alice);
  }
```

B.1.2 Method Annotations

To specify whether or not the return value of a method can be null, an annotation can be added to a method.

Figure 3 illustrates this for a class of phone books. The methods to look up a phone number or a person should never return a null value. Instead, an exception should be thrown. Remember that you cannot assume that the not-null checker agent is active when the program is used. You can only assume that it is active during the testing phase to find programming errors. Therefore, if the user of the program can enter a phone number in the user interface, there must be code that explicitly verifies that the phone number is registered. According to the GRASP principles *cohesion* and *information expert*, class `PhoneBook` is the best candidate to perform this test.

Listing 4: "Using a class annotation."

```java
import notnullcheckweaver.NotNull;
import notnullcheckweaver.Nullable;

@NotNull
public class Person {
  private String _name;
  @Nullable private Person _bestFriend;
  private Color _hairColor;
  private Color _eyeColor;

  public Person(String name, @Nullable Person bestFriend,
                Color hairColor, Color eyeColor) {
    {_name = name;
    _bestFriend = bestFriend;
    _hairColor = hairColor;
    _eyeColor = eyeColor;}

  public void changeName(String name) {_name = name;}

  public void dyeHair(Color newHairColor)
    {_hairColor = newHairColor;}

  // In this case, a null parameter makes sense, so
  // we use an @Nullable annotation.
  public boolean isBestFriend(@Nullable Person person)
    {return _bestFriend == person;}

  // It makes little sense to talk to nobody, so
  // the person cannot be null. No annotation is needed
  // because of the class modifier.
  public boolean talkTo(Person person) {...}
}
```

B.1.3 Class Annotations

In Figure 4, a class annotation is used to specify that within the context of class `Person` no field or parameter can have a null value. This means that the `_name` field no longer needs an annotation. The `_bestFriend` field, however, now needs the `@Nullable` modifier to allow it to have a null value. The difference with the previous version is that no field or parameter within `Person` can be null unless it is annotated with `@Nullable`.

Because the `name` parameter of the constructor cannot be null, the constructor of `Person` will thrown an exception even before its body is executed when it is now invoked with a null name. This also means that the `bestFriend` parameter must be annotated with `@Nullable`. Otherwise, no person can be constructed without a best friend even if the `_bestFriend` field can be null. Alternatively, another constructor can be provided that only assigns the name of a person. In the latter case, the `_bestFriend` must of course still be `@Nullable`.

The example also contains a number of new fields and parameters that cannot have a null value. Because of the class annotation, however, we do not have to write modifiers for each of them. This saves us a lot of work. It is clear that by using the appropriate class annotation, the amount of annotations can be reduced significantly. In the next part, we will see how package annotations can be used to reduce the number of modifiers even further.

B.1.4 Package Annotations

It is also possible to specify a default annotation for all fields, formal parameters, and return value in a package and all of its subpackages. To do this, you must create a file named *package-info.java* in the directory of the package that you want to annotate. Figure 5 shows how the default annotation for package `person` is set. The @annotation is also used for every subpackage of `person` *unless* that subpackage specifies its own default annotation.

Listing 5: Using @NotNull as the default annotation in the `person` package.

```
@NotNull
package person;
import notnullcheckweaver.NotNull;
```

B.2 Using The NotNullCheckWeaver Tool

You can download the notnullcheckweaver tool at the following URL:
http://code.google.com/p/notnullcheckweaver/

To use the notnullcheckweaver tool in your project, add the notnullcheckweaver jar file to your project. To use the tool at runtime, specify it as a VM argument: java -javaagent:notnullcheckweaver.jar mypackage.MyProgram. In Eclipse, open the settings for the *Run Configuration* and add the following to the ``VM arguments'' in the ``arguments'' tab: -javaagent:path-to-notnullcheckweaver.jar. The path of the cofoja jar can also be an absolute path (start the path with a /).

C Contracts for Java

Contracts For Java (cofoja) is a project from Google to provide support for the runtime verification of pre- and postconditions. Instructions for *using* cofoja in your code are described on the following website (we show an example in Section C.3:
http://code.google.com/p/cofoja/wiki/QuickReference

C.1 Running Contracts For Java

To add support for cofoja in Eclipse, you should follow these instructions. The instructions that you find when you search on the internet (and which are linked to by the cofoja website) are incomplete and outdated (the asm jar no longer seems to be required).

1. Make sure that your Eclipse program is running in a JDK, and not in a JRE. Cofoja needs the Java compiler, which is not available in a JRE. See http://wiki.eclipse.org/Eclipse.ini for instructions on how to do this.

2. Download the latest cofoja jar file from http://code.google.com/p/cofoja/

3. Add the cofoja jar to the build path of your project.

4. Open the properies dialog of your project

5. In ``Java Compiler'' $\rightarrow$ ``Annotation Processing'' enable the project specific setting and add the following options (click ``new''). Note that %PROJECT.DIR% is a shortcut for the root directory of your Eclipse project.

    - **key:** com.google.java.contract.classpath
      value: %PROJECT.DIR%/path-to-cofoja-jar-file.jar

    - **key:** com.google.java.contract.sourcepath
      value: %PROJECT.DIR%/src (or wherever you put the source files)

- **key:** com.google.java.contract.classoutput
  value: %PROJECT.DIR%/bin (or wherever the class files are placed)

6. In ``Java Compiler" → ``Annotation Processing" → ``Factory Path" enable the project specific setting, add the cofoja jar file, and make sure that it is the first in the list (disable any other factories if you are having problems).

7. Finally, in the run configurations for your project, add the following to the ``VM arguments" in the ``arguments" tab: -javaagent:path-to-cofoja-jar-file.jar. The path of the cofoja jar can also be an absolute path (start the path with a /).

8. To disable runtime verification of contracts, create a separate run configuration that does not have the VM argument described above (or remove the argument from the existing configuration).

C.2 Combining Contracts For Java with NotNullCheckWeaver

You can use cofoja together with the notnullcheckweaver. The notnullcheckweaver is much more convenient for dealing with null values. In this case, you must specify cofoja as the first VM argument, and notnullcheckweaver as the second VM argument. The agents are used in the order in which they are passed as arguments. By specifying notnullcheckweaver as the second agent, it will also protect field and parameter reads in the pre- and postconditions of cofoja.

C.3 Example

An example of using cofoja is shown in the Figure below. The transaction methods in `Account` must explicitly perform the runtime checks because the amount of the transaction is typically entered by the user of the program.

```java
package bank;

import java.util.ArrayList;

import java.util.Collection;
import java.util.List;

import notnullcheckweaver.NotNull;

import com.google.java.contract.Ensures;
import com.google.java.contract.Invariant;
import com.google.java.contract.Requires;

/**
 * NotNull is the default annotation specified for the package.
 */
@Invariant("accountsPointBack()")
public class Person {

  /**
   * Check whether all accounts of this person have this
   * person as their owner.
   */
  public boolean accountsPointBack() {
    for(Account account: getAccounts()) {
      if(account.getOwner() != this) {
        return false;
      }
    }
    return true;
  }

  /**
   * Initialize a new person with the given name. The
   * person will have no accounts.
   *
   * @param name
   */
  @Requires("canHaveAsName(name)")
  @Ensures("getName().equals(name)")
  public Person(String name) {
    _name = name;
  }

  /**
   * Increase the age of this person.
   */
  @Ensures("getAge()␣==␣old(getAge())␣+␣1")
  public void age() {
    _age++;
  }

  /**
   * Return the age of this person.
   * @return
   */
  @Ensures("result␣>=␣0")
  public int getAge() {
    return _age;
  }

  private int _age;

  /**
   * Return the name of this person.
   * @return
   */
  @Ensures("canHaveAsName(result)")
  public String getName() {
    return _name;
  }

  /**
   * Verify if the given name is valid for this person.
   */
  @Ensures("name␣!=␣null␣||␣result␣==␣false")
  public static boolean canHaveAsName(String name) {
    return name != null;
  }

  @NotNull private String _name;

  /**
   *
   * @return
   */
  public Collection<Account> getAccounts() {
    return new ArrayList<Account>(_accounts);
  }

  /**
   * Add the given account to this person.
   * @param account
   */
  @Requires("canHaveAsAccount(account)")
  @Ensures("getAccounts().contains(account)")
  public void addAccount(Account account) {
    if(! _accounts.contains(account)) {
      _accounts.add(account);
      account.setOwner(this);
    }
  }
```

```
  /**
   * Remove the given account from this person
   * @param account
   */
  @Ensures("!␣getAccounts().contains(account)")
  void removeAccount(Account account) {
    _accounts.remove(account);
  }

  /**
   * Verify if the given account is valid for this person.
   */
  @Ensures("account␣!=␣null␣||␣result␣==␣false")
  public boolean canHaveAsAccount(Account account) {
    return (account != null);
  }

  private List<Account> _accounts = new ArrayList<Account>();
}

package bank;

import notnullcheckweaver.Nullable;

import com.google.java.contract.Ensures;
import com.google.java.contract.Invariant;
import com.google.java.contract.Requires;
import com.google.java.contract.ThrowEnsures;

/**
 * @NotNull is the default annotation specified in the bank package.
 */
@Invariant({"getOwner().getAccounts().contains(this)",
            "getBank().getAccounts().contains(this)"})
public class Account {

  /**
   * Open a new account.
   *
   * @param owner The owner of the new account.
   * @param bank The bank where the new account is opened.
   */
  @Requires({"canHaveAsOwner(owner)",
             "canHaveAsBank(bank)"})
  @Ensures({"getOwner()␣==␣owner",
            "getBank()␣==␣bank"})
  Account(Person owner, Bank bank) {
    setOwner(owner);
    _bank = bank;
    bank.addAccount(this);
  }

  /**
   * Destructor. Do not use object after invoking this method.
   */
  public void destroy() {
    _bank.removeAccount(this);
    _owner.removeAccount(this);
    // We cannot set the fields to null because that would cause an
    // exception. This is not a problem because class invariants do
    // not have to hold after a destructor call.
  }

  @Ensures("canHaveAsOwner(result)")
  public Person getOwner() {
    return _owner;
  }
```

```
/**
 * Set the owner of this account.
 */
@Requires("canHaveAsOwner(person)")
@Ensures("getOwner()␣==␣person")
public void setOwner(Person person) throws IllegalArgumentException {
  if (person != _owner) {
    if (_owner != null) {
      _owner.removeAccount(this);
    }
    _owner = person;
    _owner.addAccount(this);
  }
}

/**
 * Verify if the given person is a valid owner for this account.
 */
@Ensures("person␣!=␣null␣||␣result␣==␣false")
public static boolean canHaveAsOwner(Person person) {
  return person != null && person.getAge() >= 16;
}

/**
 * Verify if the given bank is a valid bank for this account.
 */
@Ensures("bank␣!=␣null␣||␣result␣==␣false")
public static boolean canHaveAsBank(Bank bank) {
  return bank != null;
}

@Nullable
private Person _owner;

/**
 * Return the balance of this account.
 * @return
 */
public long getBalance() {
  return _balance;
}

/**
 * Withdraw the given amount of money from the account.
 * @param amount
 */
@Requires("amount␣>=␣0")
@Ensures("getBalance()␣==␣old(getBalance())␣-␣amount")
@ThrowEnsures({"NotEnoughMoneyException",
               "getBalance()␣<␣Long.MIN_VALUE␣+␣amount"})
public void withDraw(long amount) throws NotEnoughMoneyException {
  if(amount < 0) {
    throw new IllegalArgumentException("The␣amount␣is␣negative");
  }
  if(_balance < Long.MIN_VALUE+amount) {
    throw new NotEnoughMoneyException("Cannot␣withdraw␣"+amount+".");
  }
  _balance -= amount;
}

/**
 * Deposit the given amount of money on this account.
 * @param amount
 */
@Requires("amount␣>=␣0")
@Ensures("getBalance()␣==␣old(getBalance())␣+␣amount")
@ThrowEnsures({"TooMuchMoneyException",
                   "getBalance()␣>␣Long.MAX_VALUE␣-␣amount"})
  public void deposit(long amount) throws TooMuchMoneyException {
    if(amount < 0) {
      throw new IllegalArgumentException("The␣amount␣is␣negative");
    }
    if(_balance > Long.MAX_VALUE-amount) {
      throw new TooMuchMoneyException("Cannot␣deposit␣"+amount+".");
    }
    _balance += amount;
  }

  /**
   * Transfer the given amount to the given account.
   * @param amount
   * @param account
   */
  @Requires({"amount␣>=␣0","account␣!=␣null"})
  @Ensures({"getBalance()␣==␣old(getBalance())␣-␣amount",
            "account.getBalance()␣==
␣␣␣␣␣␣␣␣␣␣␣␣␣old(account.getBalance())␣+␣amount"})
  @ThrowEnsures(
   {"NotEnoughMoneyException",
      "getBalance()␣<␣Long.MIN_VALUE␣+␣amount",
    "NotEnoughMoneyException",
      "account.getBalance()␣>␣Long.MAX_VALUE␣-␣amount"})
  public void transfer(long amount, Account account)
    throws IllegalArgumentException, NotEnoughMoneyException,
           TooMuchMoneyException {
    try {
      withDraw(amount);
      account.deposit(amount);
    } catch(NotEnoughMoneyException exc) {
      throw exc;
    } catch(TooMuchMoneyException exc) {
      deposit(amount); // restore original state
      throw exc;
    }
  }

  private long _balance;

  /**
   * Return the bank that manages this account.
   * @return
   */
  @Ensures("canHaveAsBank(result)")
  public Bank getBank() {
    return _bank;
  }

  private final Bank _bank;
}

package bank;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

import com.google.java.contract.Ensures;
import com.google.java.contract.Invariant;
import com.google.java.contract.Requires;

/**
 * @NotNull is the default annotation specified in the bank package.
 */
@Invariant("accountsPointBack()")
public class Bank {
```

```java
/**
 * Initialize a new bank with the given name.
 * The bank will have no accounts.
 */
@Requires("canHaveAsName(name)")
@Ensures("getName().equals(name)")
public Bank(String name) {
  // We must verify this explicitly because the name
  // is typically entered by the user of the program.
  // We cannot rely on precondition checking because
  // it may not be activated, and we cannot expact
  // the user user to know of preconditions,
  // let alone satisfy them.
  if(! canHaveAsName(name)) {
    throw new IllegalArgumentException("Invalid␣name:␣"+name);
  }
  _name = name;
}

/**
 * This method is needed because cofoja does not support
 * quantifiers such as forall and exists.
 * @return
 */
public boolean accountsPointBack() {
  for(Account account: getAccounts()) {
    if(account.getBank() != this) {
      return false;
    }
  }
  return true;
}

/**
 * Return the name of this person.
 * @return
 */
@Ensures("canHaveAsName(result)")
public String getName() {
  return _name;
}

/**
 * Verify if the given name is valid for this person.
 */
@Ensures("name␣!=␣null␣||␣result␣==␣false")
public static boolean canHaveAsName(String name) {
  return name != null;
}

private final String _name;

/**
 * Create a new account for this given person.
 */
@Ensures({"result.getOwner()␣==␣person",
          "result.getBank()␣==␣this"})
public Account createAccount(Person person) {
  return new Account(person,this);
}

/**
 * Return the accounts managed by this bank.
 */
public Collection<Account> getAccounts() {
  return new ArrayList<Account>(_accounts);
}
```

```
  /**
   * Remove the given account from this bank. Only invoked
   * by the destructor of an account.
   */
  @Ensures("!␣getAccounts().contains(account)")
  void removeAccount(Account account) {
    _accounts.remove(account);
  }

  /**
   * Add the given account to this bank. This method is package
   * accessible because it must only be invoked
   * by the constructor of an account.
   */
  @Ensures("getAccounts().contains(account)")
  void addAccount(Account account) {
    _accounts.add(account);
  }

  private List<Account> _accounts = new ArrayList<Account>();
}
```

C.4 Simplify Testing With Annotations

In the testing phase, you can assume that the Java agents are active. This means that many test cases in a unit test become trivial. Every condition from a method that is specified using the annotations of notnullcheckweaver and cofoja will be verified automatically when that method is executed. All you have to do is invoke the method with different test parameters. Every condition that is not expressed via the annotations, however, must still be verified in the unit test.

Figure 6 shows two test methods for the banking example. In the test of the correct transactions, no actual tests are written because the transactions are completely specified using pre- and postconditions in the form of notnullcheckweaver and cofoja annotations.

Listing 6: Testing classes that are annotated with contracts.

```
@Test
public void testCorrectTransactions() {
  Bank bank = new Bank("BonusBank");
  Person poorCustomer = new Person("Very␣Poor");
  poorCustomer.age(30);
  Person evenPoorerCustomer = new Person("Even␣Poorer");
  evenPoorerCustomer.age(33);
  Account poorAccount = bank.createAccount(poorCustomer);
  poorAccount.deposit(1000);
  poorAccount.withDraw(300);
  Account poorerAccount = bank.createAccount(evenPoorerCustomer);
  poorerAccount.deposit(100);
  poorAccount.transfer(200, poorerAccount);
}

@Test(expected=PreconditionError.class)
public void testFaultyWithdraw() {
  Bank bank = new Bank("BonusBank");
  Person poorCustomer = new Person("Very␣Poor");
  Account poorAccount = bank.createAccount(poorCustomer);
  poorAccount.deposit(1000);
  poorAccount.withDraw(1001);
}

@Test(expected=ArgumentNotNullCheckException.class)
public void testInvalidPersonConstruction()
  {Person person= new Person(null);}
```