Group 7

Assignment 3

Authors:
Emilie de Bree - 4247558
Toine Hartman - 4305655
Jeffrey Helgers - 4318749
Jim Hommes - 4306090
Joost Pluim - 4162269
Matthijs Verzijl - 4282604

Supervisor:
Alberto Bacchelli

Teaching Assistant:
Aaron Ang

October 9, 2015

Contents

1. 20-Time, Reloaded

1.1 Question 2

1.1.1 Introduction

In this section the Responsibility Driven Designs and the UML for the design for this sprint can be found. The new features that have been added include having multiple lives, monsters dropping powerups and a score system. The Assignment 3 Requirements Document contain the requirements for these new features.

1.1.2 The PowerUps: Responsibility Driven Design

Powerup

Responsibility:
This class is responsible for triggering the effects when picked up. It also handles all movements of itself by calculating its speed towards its destination.
Collaborations:
To be able to trigger its effects it collaborates with Player, Monster and Bubble. It extends SpriteBase the get a sprite to be drawn.

Player

Responsibility:
This class is responsible for handling the effect, triggered by Powerup.
Collaborations:
It collaborates with Powerup and Bubbles. When Powerup wants to access Bubble, this goes through Player.

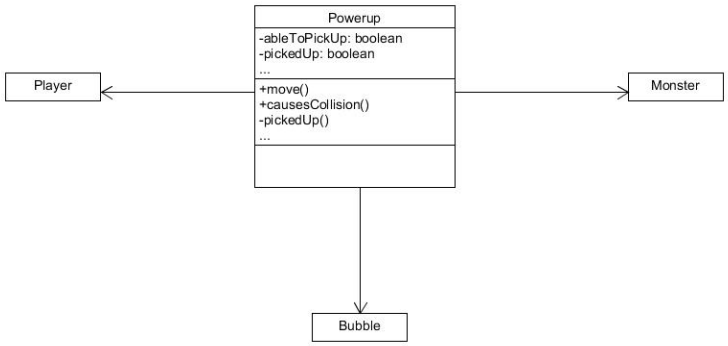
Bubble

Responsibility:
This class is responsible for handling the effect, triggered by Powerup.
Collaborations:
It collaborates with Powerup to know when an effect is triggered.

Monster

Responsibility:
This class is responsible for handling the effect, triggered by Powerup.
Collaborations:
It collaborates with Powerup to know when an effect is triggered.

1.1.3 The PowerUps: UML



The addition of a powerup is achieved by making a new class. This class is the instance of a powerup, as seen in the game. That is why it extends SpriteBase.

When a powerup is picked up, it triggers an effect. This effects for example Player, by increasing its speed. The relations in the UML above represent these effects.

1.1.4 Multiple Lives: Responsibility Driven Design

Player

Responsibility:
This class is responsible for the handling the multiple lives. It keeps track how many lives are left for that instance and handles the subtraction of a live and what happens when there are no lives left.
Collaborations:
It collaborates with Monster and Powerup, because it needs a monster collision to subtract a life and Powerup to add one.

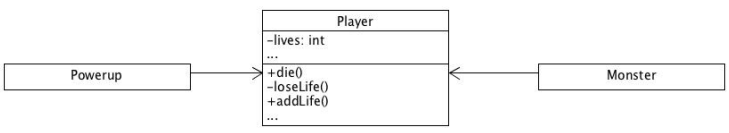
Monster

Responsibility:
This class is responsible for triggering the effect, without this there can not be a subtraction of a life.
Collaborations:
It collaborates with Player, to which it triggers the effect.

Powerup

Responsibility:
This class is responsible for triggering the effect, without this there can not be a addition of a life.
Collaborations:
It collaborates with Player, to which it triggers the effect.

1.1.5 Multiple Lives: UML



On a collision with a monster a life gets subtracted from the number of lives of a player. If there

are powerups available which will give an extra life and the players collides with this powerup (a heart) a life is added to the players number of lives.

1.1.6 A Score: Responsibility Driven Design

Powerup

Responsibility:
This class is responsible for the triggering the effects when a powerup is picked up. It increases the score when a coin is picked up.
Collaborations:
This collaborates with Player to let it know that the score has increased.

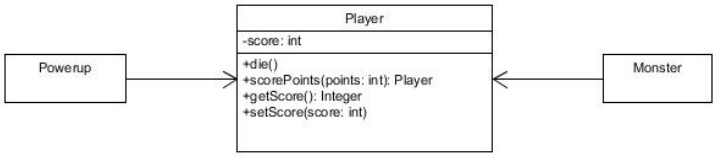
Player

Responsibility:
This class keeps a track of the player's score.
Collaborations:
This collaborates with Powerup and Monster to get information about when the score changes.

Monster

Responsibility:
This class is responsible for the triggering the effects when a player collides with a monster. In this case the score will decrease.
Collaborations:
This collaborates with Player to let it know that the score has decreased.

1.1.7 A Score: UML



On a collision with a monster the player's score will decrease. On contact with a coin power up, the player's score will increase.

2. Design Patterns

2.1 Question 1

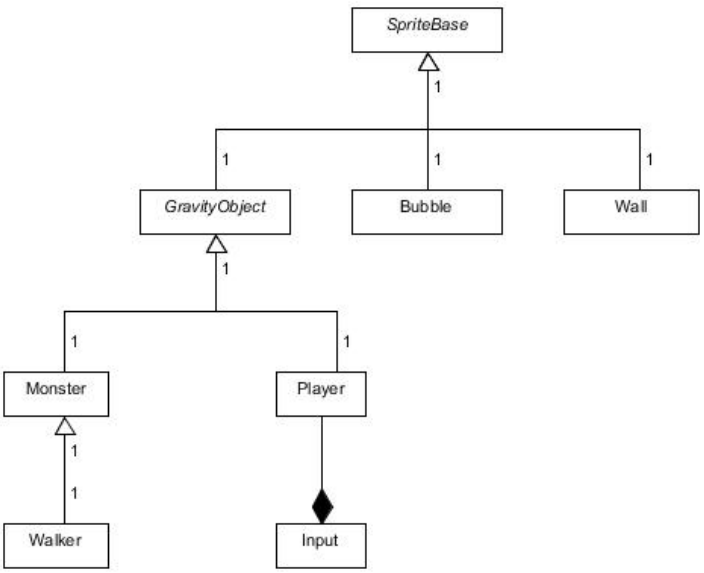
The two design patterns we used in our code are the Observer and Strategy.

The strategy pattern was easy to implement, because we already had a start to this pattern in our code. This pattern is used in the model package, where the classes are children from the 'SpriteBase' class. Not all of these classes are movable object, so we made a difference between the objects that can move and those that can't be moved. After identifying that behaviour, we could specify another behaviour in the movable object. The second behaviour that we specified is gravity, the object 'Bubble' doesn't apply to gravity because the bubbles float to the top of the screen, the other objects are part of the gravity object.

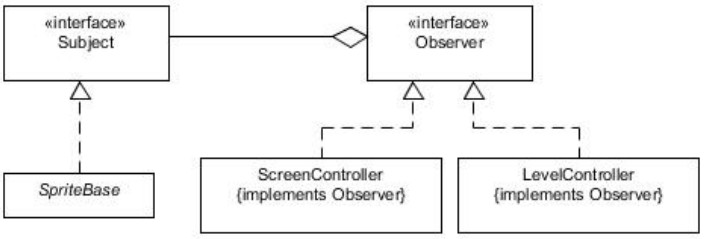
For the second pattern, observer, we had to refactor the code. In the observer pattern you have a Subject with a couple of observers, these observers update their state when something changes in the subject. In our code the subjects are instances of 'Subject' and the 'LevelController' and 'ScreenController' are the observers. When, for instance, a monster is caught by a bubble and the player touches that bubble, the monster and bubble have to disappear from the screen, this is notified to the 'ScreenController' and then the screen can be updated without the specific bubble and monster.

2.2 Question 2

2.2.1 Class Diagram for Strategy

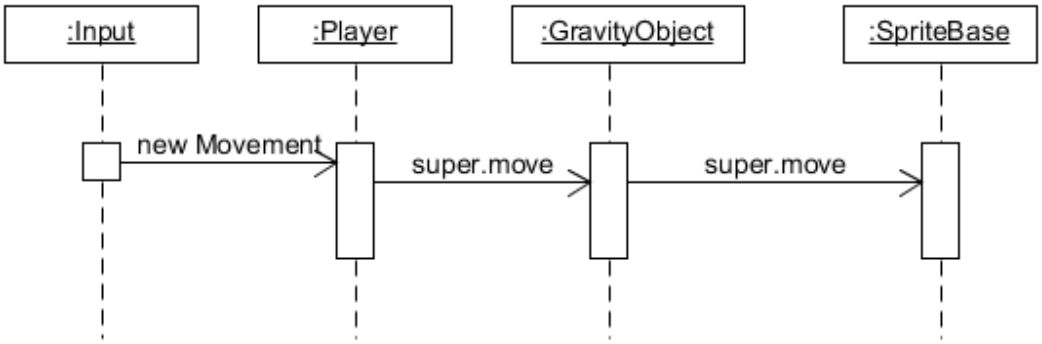


2.2.2 Class Diagram for Observer



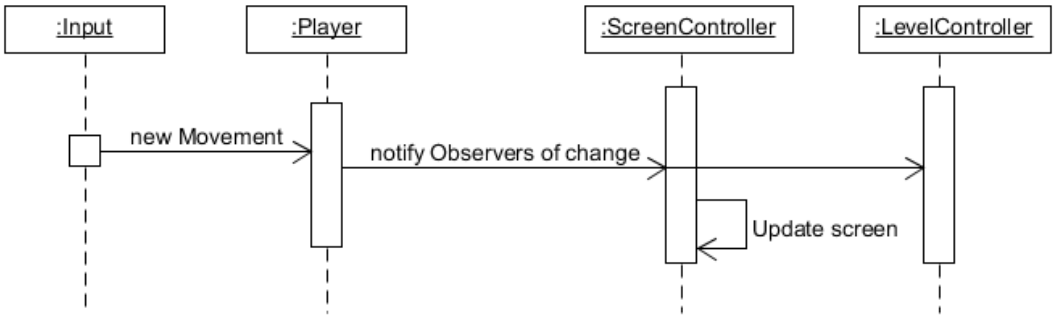
2.3 Question 3

2.3.1 Sequence Diagram for Strategy



2.3.2 Sequence Diagram for Observer

The observer design pattern, the player has a new movement and he notifies his observers that some change is made.



3. Software Engineering Economics

3.1 Question 1

For the research 352 software projects were collected in a repository. All of those 352 projects were assigned a label in each of 8 different categories (ranging from 'Business Domain' to 'Primary Programming Language'). Furthermore they were assigned different values of their measurements (such as Size and Duration).

To determine whether a project is labelled as a good or bad practice, first the average of three factors had to be determined:

1. Project Size (in Function Points)
2. Project Cost (in Euro's)
3. Project Duration (in months)

Once these averages have been calculated, two indexes can be determined:

1. Duration Deviation from Mean Duration (expressed in percentage deviation from the mean duration) corrected for the applicable project size: negative deviation is regarded 'good', positive deviation is regarded 'bad'.
2. Cost Deviation from Mean Cost (expressed in percentage deviation from the mean cost) corrected for the applicable project size: negative deviation is regarded 'good', positive deviation is regarded 'bad'.

Every project can now be evaluated regarding both averages (mean duration and mean cost). We can distinguish 4 cases:

- If both evaluations are 'good', the project is regarded as a **Good Practice**.
- If both evaluations are 'bad', the project is regarded as a **Bad Practice**.
- If duration is regarded 'good' and cost is regarded 'bad', the project is regarded **Time over Cost**.
- If cost is regarded 'good' and duration is regarded 'bad', the project is regarded **Cost over Time**.

3.2 Question 2

As described on page 70 of the paper, only 6 projects with the Project Factor "PPL Visual Basic" were analysed. Of those 6 projects, 5 scored as a Good Practice and one scored as Cost over Time. Therefore the Project Factor "PPL Visual Basic" was selected as one of the factors which relate to Good Practice. As you can see however, the number of samples in this group is too small to conclude that Visual Basics always is a success factor. A good improvement would be to set a minimum sample size n in order to be selected as a success or failure factor.

3.3 Question 3

1. **Test Driven Development**: This would probably be a good practice. Although Test Driven Development might take more time and could therefore maybe end up in the 'Cost over Time'-quarter, it would probably end up as a Good Practice. This is because the developed software will be more maintainable and will be easier to verify for bugs. Because of this, the quality of the software will improve which will mean relatively fewer costs than, for example, when bugs have to be removed at the end. Test Driven Development also means investing a little more time in testing while developing software. This could result in a relatively longer duration, but we are convinced that this little investment of time will result in lower overall duration of the project, since bugs will be found easier and therefore will help prevent more bugs.

2. **Including Design Patterns**: This would probably be good practice mainly due to the same reasons as Test Driven Development. Initially working with Design Patterns might take a bit more time and will require a team with more skills (so more expensive employees). However in the end it will results in easily extendable and more maintainable software which will result in a positive practice.

3. **Visual Development Process**: With this we mean that during the development process, intensive use is made of visualisation (such as UML's) to keep track of for example software structure or model relations. We believe that this will result in a global understanding of the desired structure of the application which will result in lower cost and duration. It is easier to discuss problems, improve software or add new features, once it is understood how the application will be / is constructed. Visualisation is a very good way of doing all of this.

3.4 Question 4

1. **Rules & Regulations driven**: Having to comply with a lot of Rules & Regulations, means having to spend a lot of time on checking whether the software won't violate any of these Rules & Regulations. This can easily result in a project duration that is longer than projects with comparable size. Having to comply to these Rules & Regulations can also easily result in more reviews being necessary and therefore more costs.

2. **Dependencies with other systems**: As one can imagine, once a certain dependency changes, bugs can easily occur. Sometimes developers can control the dependencies themselves, to prevent many changes. However often dependencies can be outside of the range of the developers of the project. Once a dependency changes in a late stage of development, this can lead to large adjustments in code, which in its turn can also easily result in bugs. Both of these factors will result in a longer duration and higher costs.

3. **Once-only project**: For a Once Only project, it might be necessary to do a lot of initial work on the set-up of a project. When continuing a project, this work has already been done which results in a shorter duration of the project. When setting up a project, the exact set-up might still be unclear, what can result in changes to the set-up once the team already started developing. This wastes time and therefore a Once-only project often has a large chance of being a bad practice.