XO-Bubbles

A Frozen Bubble Clone in Smalltalk/Squeak

Konstantin Haase[1], Tim Felgentreff[1], Johannes Wollert[1], Michael Winkelmann[2], Robert Pfeiffer[1]

[1] Hasso-Plattner-Institut, Universität Potsdam, D-14482 Potsdam, Germany,
{konstantin.haase,tim.felgentreff,johannes.wollert,
robert.pfeiffer}@student.hpi.uni-potsdam.de
[2] Institut für Informatik, Universität Potsdam, D-14482 Potsdam, Germany,
mwinkel@uni-potsdam.de

Abstract. During the course Software Architecture we attended to in the semester 2008/09 we implemented a clone of the popular "Frozen-Bubble" open-source game using object-oriented techniques and the Squeak implementation of Smalltalk. Our clone is not simple copy though, it was designed for flexible themability and a more challenging game mechanics.

1 Introduction

1.1 Overview

The choice for our game fell on cloning the open-source game *Frozen-Bubble* [1]. Within the game, the user controls a small cannon, shooting bubbles and tries to hit other bubbles in order to create groups of 3 or more bubbles of the same color. Whenever this is accomplished these bubbles go away and the player gains points. For a larger number of balls the player is awarded exponentially more points, so the goal is to clear out large numbers of balls at once by building chains and removing balls near the top of the chain. In single player mode, the game goes on as long as the balls do not reach the bottom of the stage, much like popular arcade games as, for example, Tetris or Puyo-Puyo. In multiplayer mode players compete and try to delay filling their playfield as long as possible.

1.2 The Original

The original Frozen-Bubble was written in Perl using the *Simple Direct Media Layer* (SDL) for graphics and sound. It was developed within three months and had its first official release at the end of January, 2002. While initially it only ran on GNU/Linux it has since been ported to other operating systems supported by SDL, thanks to being licensed under the GPLv2. It also runs on the Symbian™ S60 platform, the Palm® operating system and as Java™ applet in the browser[3].

[3] Symbian is a trademark of Symbian Foundation, Palm is a registered trademark of Palm, Inc. and Java is a trademark of Sun Microsystems

The original game features a single player mode with multiple levelsets. Each level of a level set defines an initial playfield layout and a level finishes when the player manages to remove all balls from the playfield.
Additionally, the original game also has a multiplayer mode for two players locally and up to five players over the network. Here, the goal is to clear the balls faster and more efficiently than your opponents, and clearing a great number of balls is rewarded by adding "penalty balls" to the opponents' playfields.

Fig. 1. A view of the original Frozen Bubble with the 2.0 theme

1.3 The Clone

Our clone of the Frozen-Bubble game keeps the name in consensus with the original developers. We have also acquired the permissions of the original artist to release his graphics together with our code under the MIT license. The single player experience has been changed as stated above to relief us from the need to have to supply levelsets with the game.
We have not implemented a networked multiplayer mode, however, playing locally against a human opponent works to a fair extent as in the original.

Our interpretation introduces some slight changes to the gaming experience. Most notably we support "realistic" collision instead of a hex-field for the balls, which makes playing a lot more challenging. On the visual side, we support more flexible theming. In the original that has not been a goal, so "themes" are limited to replacing PNG images with a fixed geometry..

1.4 Design Goals

A goal we set for ourselves very early was themeability, e.g. the ability to modify and adjust certain visual and logical aspects of the game (for example ball images or cannon positions). This demands an efficient and maintainable theme management system, as lots of pictures and data need to be processed. As previously stated, another important aspect of our clone is "realistic" collision between objects. To achieve this, an algorithm other than simply looping through all objects present to counter the hardware limitations of the OLPC-XO. Lastly, we want our game to support multiplayer functionality. Given the time needed to implement networking, we decided to focus on a local multiplayer modes.

1.5 Outline

In this paper, we will present our game design and shed some light on the decisions it is based on. We will begin with an introduction to our system architecture

and gradually expand that to include the whole system structure.
Finally we will outline our development plan and process and elaborate on the future of our project, licensing issues and the like.

2 Architecture

The central classes of our game handle theming, player actions, game logic and the graphical representation of the game. These central classes of our game are depicted in Fig.2. The first instantiated class is FBGame. FBGame controls some of the global game functionality, like the number of players, reward strategies like highscores and the conditions of victory and defeat. It is also responsible for instantiating the FBWorld class which encapsulates graphical representation of the game.

FBWorld is passed a symbol at startup, which is the name of a FBTheme. FBWorld holds a reference to the current theme and creates its main submorphs, like a menu, highscore list the changing backgrounds and the in-game playfields. These Morphs, however, are not held in an instance variable of any kind, but are returned to the FBGame instance, for co-ordination.
Out of technical necessity, FBWorld also catches all keystrokes, handling them, however, is up to the FBGame. Global events, like pause and quit, are handled directly, all other keystrokes are passed on to all playfields in the game (if any) and handled there.

FBPlayfields are created by the world upon the requirements of the theme and then returned to the FBGame. The playfields receive their own reference to the theme and are themselves responsible for their content and reaction on player input, thus allowing us to implement multiplayer gaming simply by creating more playfields in our world.

3 Design Process

3.1 Design Goals

In order to achieve a pleasant and cleanly written FrozenBubble clone, we specified certain design goals. A goal we set for ourselves very early was themeability, e.g. the ability to modify and adjust certain visual and logical aspects of the game (for example ball images or cannon positions). This demands an efficient and maintainable theme management system, as lots of pictures and data need to be processed. As previously stated, another important aspect of our clone is "realistic" collision between objects, which came with certain performance considerations. Lastly, we wanted our game to support multiplayer functionality. Given the time needed to implement networking, we decided to focus on a local multiplayer modes.

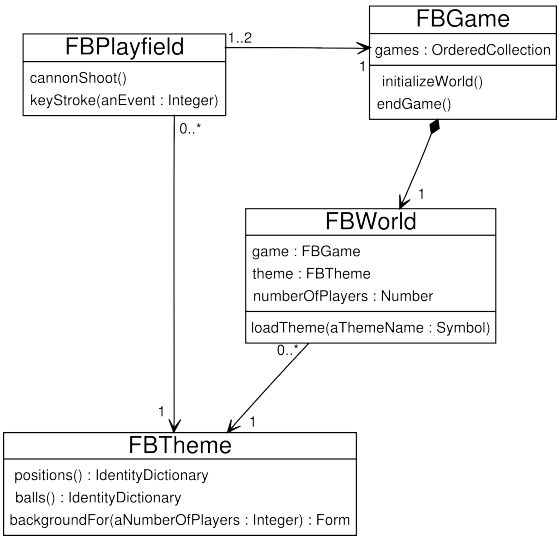

Fig. 2. The Central System Architecture

3.2 Design Problems and Solutions

Slow Disk I/O. Different from the original, we wanted to decouple the game from its representation to the point that we could allow different themes to change the positions of the game objects for different images and backgrounds. We decided to use PNG images to allow great visual flexibility and XML files to define the theme's positions. As disk I/O is expensive, we needed a way to avoid it as much as possible, without sacrificing flexibility. Thus, lazy initialization for our images holding class was not an option. We created a class solely responsible for loading images and use the flyweight pattern to avoid reloading a theme we already loaded.

Collision Between Balls. When a ball is shot into the playfield, it has to interact with the playfield and other objects in its path. Simply looping over all those objects and testing for collision is in O(n) and results in slower movement of the ball and a general loss in responsiveness. A quick solution would have been to keep two lists of all objects sorted by their x and by their y axis, and only testing those items closest to the current ball in both lists. Now O(n) would only be the worst case. It occured to us that using rasterization or using precalculated vectors similar to ray-tracing techniques could get us a constant complexity. We decided to implement rasterization, because it was a faster solution and allowed straightforward optimization.

Falling Balls. When balls can collide and stick to each other, having them fall down if three or more of the same kind touch is trivial. However, balls hanging off balls that fall may have to fall as well. We considered two ways of checking for such conditions. We first planned to regard balls in the playfield as a network of nodes and use a graph search algorithm to determine whether or not a given ball has a connection to the top of the playfield via others. This proved difficult to implement and hurting encapsulation. Thus we applied a metaphor of a garbage collector (GC) to the problem, where balls ball that are not connected to the top are collected. Performance was no pressing concern here, as the number of balls in one playfield is relatively limited, and the algorithm runs at a moment where the user has just shot, so taking a second for the calculation would not hurt the game flow.

Simple Multiplayer Mode. Several problems occured when we implemented a simple local multiplayer mode. In this chapter, we will discuss the most important design decisions.

Players need to be informed of critical actions happening in other players' playfields
Direct communication between playfields would be the most simple solution like and could be implemented easily. Additionally only the most rudimentary communication is needed, so there would be very few overhead. But this is not an object-oriented solution and playfields would need to be enumerated directly in the code, making the code hardly maintainable and badly extensible.
In contrast, delegating communication through a mediator makes very maintainable and expandable code. Other than the previous proposal, this is indeed an object-oriented take on the problem. However, there is more than twice as much communication happening compared to the previous, simplified solution. Due to the obvious disadvantages of hardcoding different playfields, we chose the latter proposal.

Different playfields have to be controlled by different players. And, of course, different playfields have to exist in the first place
Our first idea we came up with was enabling the generic playfield to manage all needed behaviour on itself. That way, no new code needs to be introduced, rendering this scheme very simple. On the other hand, the playfield might have to be extended for all purposes to work properly. This might lead to a very huge class indeed, affecting the readability.
Introducing an Abstract Factory for playfields was another idea, thus implementing several behavioural patterns. The obvious advantage of this proposal is, that Abstract Factories are extensible and object-oriented. The downside of it would be, that this is a very powerful pattern and would probably be overindulgence because we just need two different playfields (single- and multiplayer behaviour). Because of that, we chose our initial idea of keeping things simple.

Key events need to be redirected to the corresponding playfield
The most easy-to-code version would be the introduction of a major, keyhandling class that knows what to invoke when a certain event takes place. Obviously, this has several downsides, as there would be strong coherence between objects and no encapsulation at all.
As such a way of handling events is not maintainable and hardly extensible, we chose to catch key events through a class but unlike the previous solution, more object-oriented and maintainable patterns would be applied.
An option is the Observer pattern where every object registers itself for certain keystrokes at a dispatch-object. That way, only objects that actually handle a specific event are informed. However, a lot of communication is happening that way.
Another pattern that can be considered is the Chain of Responsibility, passing the event in question to other objects, when it cannot be handled by oneself. It is a very intuitive and object-oriented way of solving our problem, but has the same disadvantages of the Observer pattern. In fact, the overhead problem is even more severe.
As the complexity of a Chain of Responsibility is a lot less than that of an Observer pattern, we chose the latter scheme of handling keystrokes. Our problem just is not complex enough to justify a pattern as powerful as the Observer.

4 System Overview

4.1 Extensions to the core system classes

Rewarding the Players
FBRewardStrategies are supporting objects which add rewarding behaviour to the game logic (Fig.4). Its specializations implement different strategies to reward the players for good gaming. Good gaming is measured by the number of balls falling, if more are falling at the same time, more reward points are gained. The reward strategies have to implement a method rewardPlayer:for: (Listing 1.1).
In single player mode, a FBHighscoreReward adds a simple highscore to the screen (responsible for adding the score visual is also the strategy itself) and for any number of reward points reported by the FBPlayfield, score points are accumulated. In multiplayer mode, the strategy only rewards, if the player eliminates more than three balls from the field with a single shot. If that happens, a number of balls half as great is distributed among the other players, and shot randomly in any direction. This happens via a call to the playfields which have to shoot random balls, since FBRewardStretegies hold a collection referring to all playfields.

FBTheme and FBImageLibrary
To avoid expensive reading from hard disk as mentioned in our design solutions (3.2) we implemented the FBTheme class as a flyweight, holding references to previously created instances of a particular theme. If a requested theme

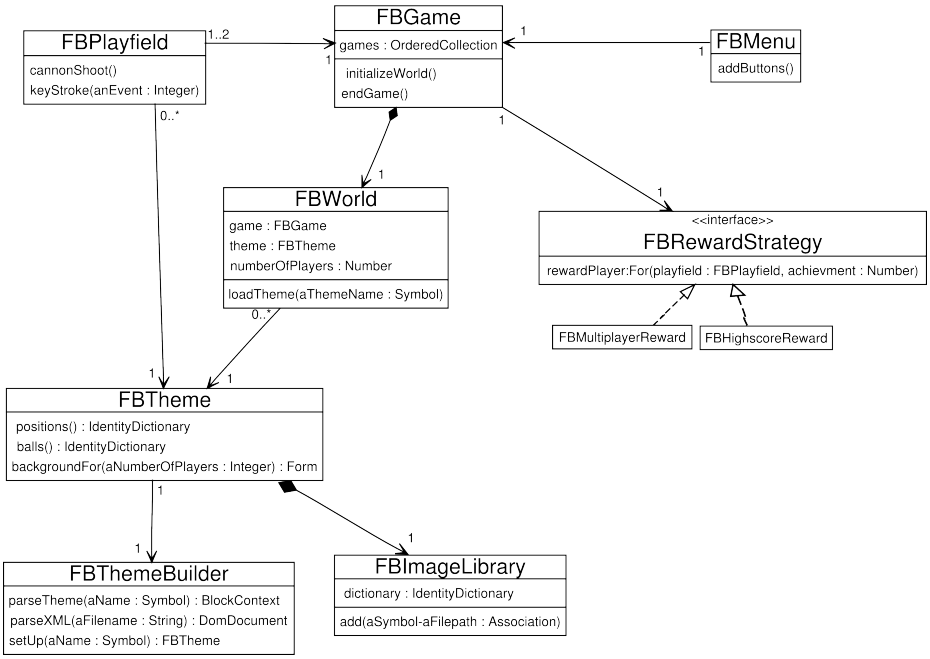

Fig. 3. The extended core system

type is not yet held in the list of instances 1.2, the class calls a builder class, FBThemeBuilder 3, to parse the requested theme's XML file, load the images and then return a fresh instance of the theme. Additionally, each theme contains an FBImageLibrary. The image library is just a simple wrapper around an IdentityDictionary to ease filling it with Forms. FBTheme and FBImageLibrary also support scaling to adjust to any screen size (without accounting for necessary interpolation). The FBImageLibrary is filled prior to starting the actual game, so that all external resources are cached. A lazily initialized image library would have resulted in slow disk I/O during run-time.

4.2 FBPlayfield and associated classes

Playfield Morphs
As aforementioned, FBPlayfield is responsible for creating its own contents (4). Such content displays a cannon, a player avatar, messages for the user and numerous coloured balls. Each of these serves only a single, limited purpose, with FBPlayer and FBMessage merely wrapping different types of graphical interaction. The cannon is the entity that reacts to the user's input, panning left and right and sending balls off into the field. However, it is controlled entirely by the playfield and does not act itself. The FBBall is an exception to this simplicity. Because of that, we decided to separate all logic concerning falling FBBalls from

```
FBHighscoreReward»rewardPlayer: aPlayer for:
    anAchievmentScalar
    "I reward a single player with an exponentially
rising rate of points"

    self score:
        self score + (anAchievmentScalar raisedTo: 2)
```

Listing 1.1. The Highscore Calculation Method

```
FBTheme class»return: aSymbol
    "Return the instance of the requested type if we have it.
        If not replace any old instance with the type
        requested."
    instances ifNil: [instances := IdentityDictionary new].
    ↑ instances
        at: aSymbol
        ifAbsent: [
            | instance |
            instance := self builder setUp: aSymbol.
            (instances add: aSymbol -> instance) value].
```

Listing 1.2. FBTheme "return:" method

the rest of the playfield and provide an additional abstraction layer, FBPlayfieldController.

First it has to look differently, showing different colors in our themes.

Secondly a Ball has different stages in its lifetime, being loaded into the cannon, then flying, hanging from the ceiling (or, indeed, from other balls) and finally falling and vanishing.

Finally a Ball needs to collide with the playfield boundaries and other balls in the field.

Thus its implementation is a bit larger, as it implements the FBCollider interface, utilizes different "states" to act on in its step method and and also draws itself from a collection of images.

4.3 FBMap

Collision Detection As mentioned in section 3.2, collision detection is handled by one collision map per playfield. This collision map is an instance of FBCollision. A collision is detected a posteriori, meaning that the ball first moves to the next position and than checks whether it collides with some other object. However, this is not strictly necessary and not reflected in the implementation of the collision map. Any object may ask the collision map whether a collision

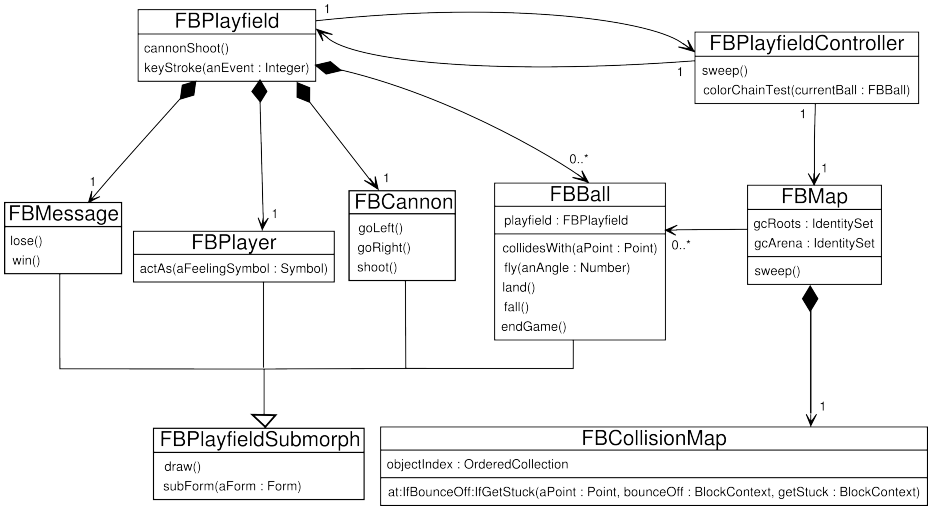

Fig. 4. The classes connected to each playfield

is happening at a given point on the map and may react accordingly. There are two kinds of collision: One where a ball usually should get stuck and one where it should bounce of. As can be seen in Listing 1.3, the API call wraps around doYouGetStuckAt: and doYouBounceOff: - checking in that order, since if a "stuck collision" should occur, it has a higher priority than a "bounce of collision".

```
FBMap» at:aPoint ifBounceOff:bounceOffBlock ifGetStuck:
    getStuckBlock

        (self doYouGetStuckAt: aPoint)
                ifTrue: [getStuckBlock value]
                ifFalse: [ (self doYouBounceOff: aPoint)
                        ifTrue: bounceOffBlock ]
```

Listing 1.3. API method for detecting collision

Up to this point the API is independent from how an collision is actually detected. The playfield is divided grid, each cell being the size of a ball. Those cells are represented in the objectIndex collection. Each ball that lands somewhere registers on all cells where another ball could possibly collide with it, which are all the cells that are direct neighbours of the cell in which the ball landed. Note that the cell a ball "is in" is the cell the ball's center is in. Now doYouGetStuckAt: aPoint returns true if one of the balls registered on the cell aPoint is in, would collide with a ball with a center at aPoint (or if the ball would be close enough to the upper border). So instead of comparing with all balls on the playfield, which would be the most trivial implementations, we only compare the current position to the position of the balls near by. Since a cell is the size of a ball, there

can only be up to 8 balls surrounding a cell, independent of playfield size or total number of balls. Therefor, collision detection is in $O(1)$ complexity, compared to $O(n)$ of the trivial implementation.

As mentioned in 3.2, an alternative implementation could have been pre-computing the route with vectors. However, this has at least two downsides. First, it does not cope with changes happening while the ball is flying. But this is not likely to happen. The other downside is, that it does need more computation than our solution. Not only would we need some concept of vectors, we would also have to check with every ball on the playfield or use some grid algorithm like the one we are using for our implementation. Thus we would have the computation we already have plus the vector algorithms. The advantage would be to avoid collision detection computation while the ball is flying. However, profiling did show that this would probably not lead to an performance boost. Moreover, our current implementation does not check for every pixel lying on the way, but only for those where it will "jump" to on every step, so we have less comparisons than compared to the vector approach. Not checking for every pixel forces us to readjust the position as soon as the ball got stuck.

Ball Falling Conditions As discussed in section 3.2, we modeled the process to decide whether balls can fall down on garbage collection. Our first design was inspired by reference-counting and planned for every ball to keep track of all objects it holds on to and all objects that hold on to it. When an object falls down, it informs the objects that hold on to it, via the Squeak built-in event mechanism [2], that they lost a holder. When an object has lost its last holder, it should itself fall down and inform all objects that hold on to it that it is gone. This approach had several drawbacks. First, the logic for this falling down was distributed over the FBCollisionMap, FBBall and FBCollider[4] classes. Second, every Ball observed the Balls next to it. When a Ball was removed from the structure, the observing Balls removed it from their lists of holders. In the event that a Ball had no holders, it fell down, thereby invoking events on its observing neighbours. This resulted in a control flow that was hard to understand, as the information of the removal of a Ball was propagated to the other Balls implicitly through events and not explicitly through messages.

Third, reference-counting garbage collectors have a well-known problem with cyclic references.

In order to separate the concern of letting balls fall down the from the rest of the game logic, the FBMap keeps track of the balls and the relationships between them.

Like a mark-and-sweep garbage collector, the sweepBalls method traverses the balls in the Playfield and marks every ball it can reach, beginning with the balls that stick to the top of the playfield. The Balls are traversed depth-first. All unmarked balls cannot be reached from balls that hang. They are removed from the map.

[4] This class had only been needed with the reference counting approach and was discarded for the final version.

```
FBMap»sweepBalls
        "Grep chains for all roots and remove the rest"

        | markedColliders removableColliders |
        markedColliders := gcRoots copy.
        self gcRoots do: [:each |
                markedColliders addAll: (self
                        chainFor: each
                        except: markedColliders)].
        removableColliders := (self
                        ejectAll: markedColliders
                        from: self gcArena).
        removableColliders do: [:each | self remove: each].
        ↑ removableColliders
```

Listing 1.4. Mark-and-Sweep Ball Collector

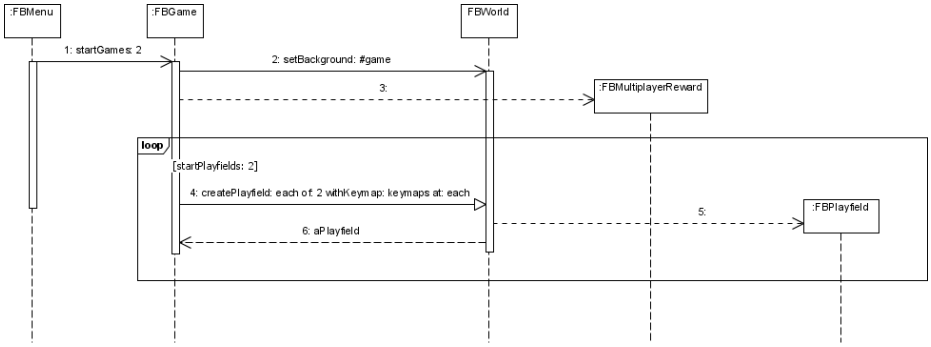

Fig. 5. Starting a 2 Player Game

5 Examplary Two Player Game Sequence

5.1 Starting the Game

As to be seen in Fig.5, the main menu is created and added to the world by requet of the FBGame instance, returning the reference to the FBGame. This menu offers a few options to start a game session. Clicking on the "Start 2 Player Game" button results in the message FBGame»startGames: to be send with the parameter two. A loop is initialized, creating 2 discreet FBPlayfields by calling the createPlayfield: with: method of FBWorld and linking the corresponding keymaps to them. A FBMultiplayerReward object is instantiated for the game. The FBWorld instance catches any keyboard interrupt and hands it over to the game. If this keystroke is not registered in the games keymap (e.g. it's not bound to pause or exit) it is handed over to all the playfields, which may ignore it or act accordingly - if it's in their keymap. As we assume in this example, that the

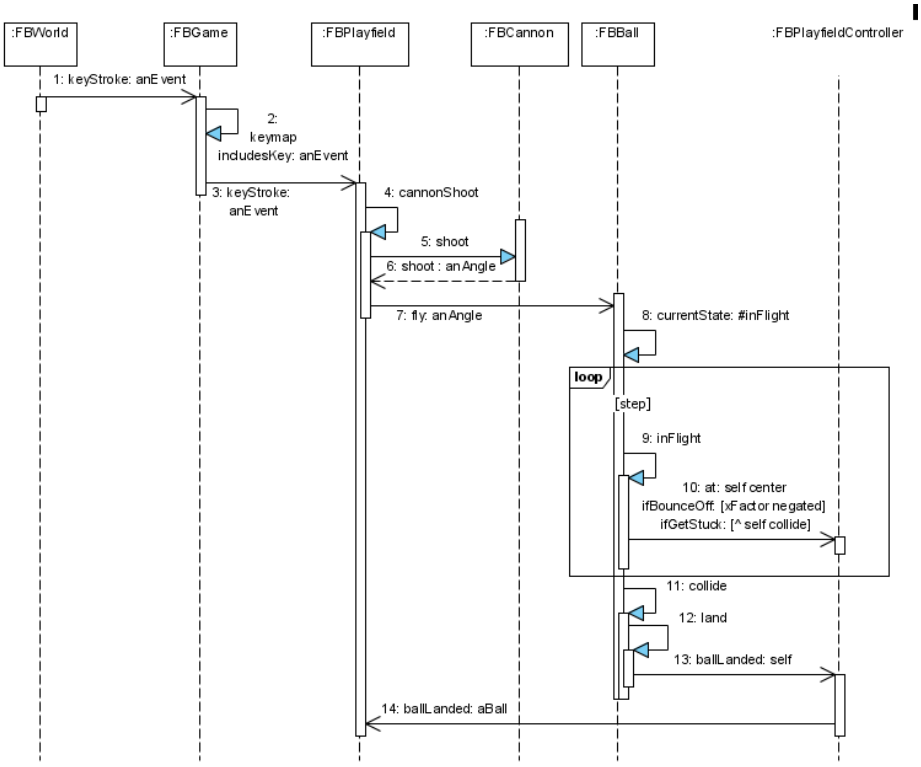

Fig. 6. Shooting a Ball

key that was pressed is bound to the shoot command of the playfield, it picks the shoot event from the keymap. Therefor the playfield will ask it's cannon to shoot. The cannon sends the message shoot: to the playfield it belongs to, with it's current angle as parameter. The playfield sends the message fly: to it's current ball so it takes of with the given angle. The ball keeps flying until ti collides, as described in section 4.3. As soon as the ball got stuck, it notifies the playfield controller, which in turn notifies the playfield.

5.2 Rewarding a Player

Rewarding a player consists of very few steps, as depicted in Fig.7. First, the FBPlayfield of the player to be rewarded collects its reward through the collectReward: method, thus calling the FBGame to distribute our reward. FBGame itself does not handle specific rewards itself, instead it informs the necessary FBRewardStrategy subclass, in this case FBMultiplayerReward, that a certain FBPlayfield needs to be rewarded. The bonus for a multiplayer playfield is ac-

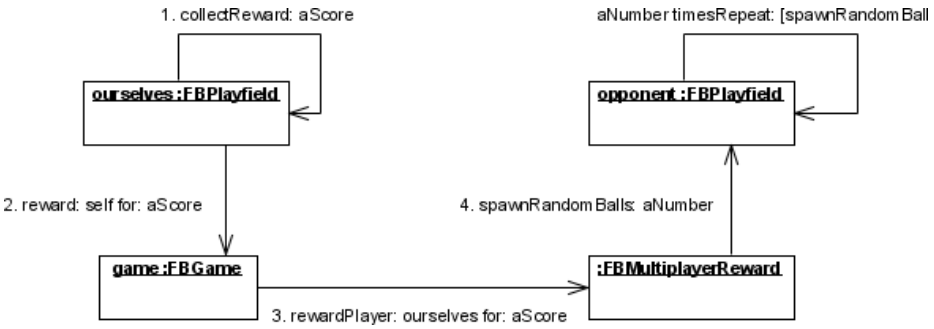

Fig. 7. Rewarding a Player for his Actions

tually a penalty for others, some FBBalls are randomly shot into the opponents playfield.

6 Development Process

Our development roadmap scheduled three successive releases. The first release, called "Waitaha", was developed using the pair programming technique: While one developer was typing code at least one other developer was constantly watching and reviewing it. This version was the first basic clone of the original Frozen Bubble game.

"Little Blue" was the second release and the first one to be presented to the public, featuring a restructured architecture and a stable API. With this version there was a shift in development with each of the developers encountered a shift in programming techniques: Less pair programming but heavy reliance on version control (Monticello and Git) and frequent meetings for code review.

The final version - "Rockhopper" - is considered feature complete for general game purposes.

7 Conclusions

We managed to build a functional and flexible FrozenBubble clone for Morphic, but given our time constraints, not all features we would have liked to implement made it into the final version. Especially a working network multiplayer mode would have been quite delightful. It is possible however to play a local multiplayer mode, limited in use only by the self-blocking keypress messages provided by morphic.

Several schemes and patterns helped us to achieve our goals, especially the git version control system, modelling the system components before actually coding and the simple and intuitive smalltalk language along with the Squeak environment, especially the class browser.

Acknowledgements

We are indebted to the developers of Frozen Bubble for allowing us to release our clone with the name and graphics of the original game under the MIT license. We also thank Professor Hirschfeld, Tobias Pape and Michael Perscheid for their advice.

References

1. Cottenceau, G., Younes, A., Bidan, M.L., Kim, Joham, D., Amblard-Ladurantie, A.: Frozen Bubble - the official home. http://www.frozen-bubble.org (February 2009)
2. Collaborative Squeak Wiki: Squeak and events (observer pattern). http://wiki.squeak.org/squeak/1214 (February 2009)
3. Beck, K.: Smalltalk: best practice patterns. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1997)
4. Black, A., Ducasse, S., Nierstrasz, O., Pollet, D., Cassou, D., Denker, M.: Squeak by Example. Square Bracket Associates (2007) http://SqueakByExample.org/.
5. Klimas, E.J., Skublics, S., Thomas, D.A.: Smalltalk with style. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1996)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional (1995)