Software Engineering Economics and Design Patterns in Space Invaders

Sprint 3

by

A. Smesseim, S. Baraç, A. el Khalki, A.W.L. Oostmeyer, M. Maton

in partial fulfillment of the requirements for the degree of

Bachelor of Science
in Computer Science

at the Delft University of Technology,

to be presented on 9 October 2015.

Supervisor: Dr. A. Bacchelli

TU Delft Delft University of Technology

Contents

1

20-Time, Reloaded

1.1. Requirements

1.1.1. Functional Requirements

For the game Space Invaders, the requirements regarding functionality and service of the levels and stores are grouped under the Functional Requirements. Within these functional requirements, four categories can be identified using the MoSCoW[1] model for prioritizing requirements.

Must Haves

1. The game must show the current level above the playing field. At the start of the game, the current level is 1.

2. If all enemies have disappeared from the playing field (i.e. are killed), then a store is shown. This store is text-based, and allows the player to trade some of its points to purchase powerups. The powerups are:

   - an additional life for 1000 points.
   - a bomb, that lets the enemy disappear upon impact, as well as all the horizontally, vertically and diagonally adjacent enemies, for 250 points.
   - the restoration of the barricades to their original state, for 500 points.

3. The store must show a "Continue" button, that terminates the store, and shows the next level to the player. The enemies and ship are reset to their starting position, and the level counter is incremented.

Should Haves

4. The speed of the enemies should be cx where x is the current level, and c is a preset constant.

Could Haves

5. In addition to lives, bombs, and the restoration of the barricades, the store could also offer:

   - the ability for the player to move vertically, for 2000 points.
   - a shield, where the player is invulnerable at the start of the next level for 20 seconds, for 300 points.

6. Instead of having a text-based store, the store could also show sprites near the offered items. To clarify:

   - instead of showing "Buy a life", the store could show a heart sprite.
   - instead of showing "Buy a bomb", the store could show a bomb sprite.
   - instead of showing "Buy a shield", the store could show a shield sprite.

Would/won't Haves

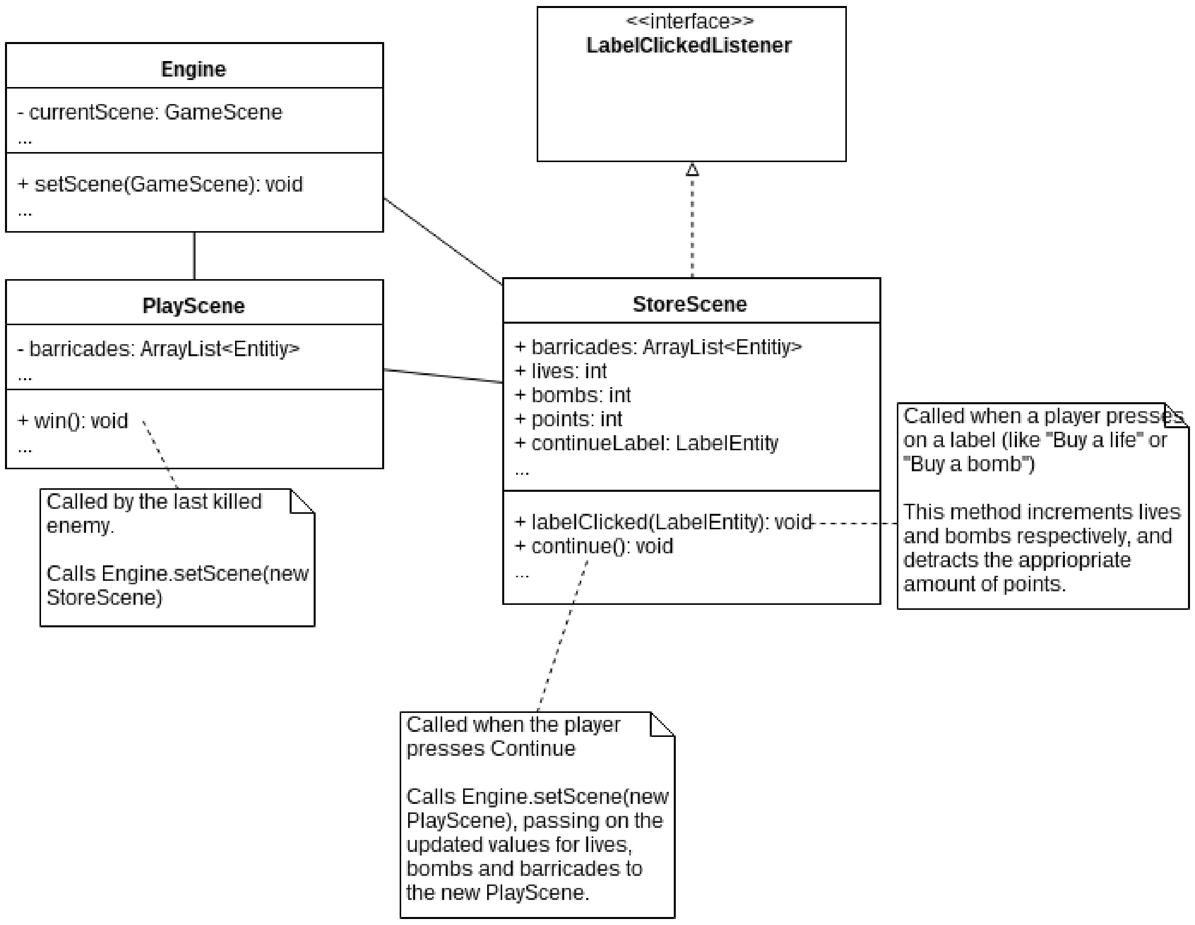
7. The game won't support saving the player's progress after each level.

1.1.2. NON-FUNCTIONAL REQUIREMENTS

In addition to the functional requirements specified in chapter 1.1.1, the game should also adhere to several non-functional requirements, outlined in this chapter. These requirements will not affect the functionality of the game, but will ensure that the game is gradable by the instructors of the course TI2206. The non-functional requirements of the game are:

8. The game must be playable on Windows (7 or higher), Mac OS X (10.8 and higher) and Linux.
9. The game must be implemented in Java.
10. A first fully working version of the game must be delivered on September 11, 2015
11. Scrum methodology must be applied.
12. The source code must be hosted on GitHub as a public repository.
13. The version control used for developing this game must be Git.
14. The tests must be automated using JUnit.
15. This project must use Maven for build automation.
16. This project must use Travis CI for continuous integration.
17. This project must employ the following static analysis tools: Checkstyle, PMD and FindBugs.

1.2. UML



<<interface>>
LabelClickedListener
Engine
- currentScene: GameScene
...
+ setScene(GameScene): void
...
PlayScene
- barricades: ArrayList<Entitiy>
...
+ win(): void
...
StoreScene
+ barricades: ArrayList<Entitiy>
+ lives: int
+ bombs: int
+ points: int
+ continueLabel: LabelEntity
...
+ labelClicked(LabelEntity): void
+ continue(): void
...
Called by the last killed enemy.
Calls Engine.setScene(new StoreScene)
Called when a player presses on a label (like "Buy a life" or "Buy a bomb")
This method increments lives and bombs respectively, and detracts the appriopriate amount of points.
Called when the player presses Continue
Calls Engine.setScene(new PlayScene), passing on the updated values for lives, bombs and barricades to the new PlayScene.

1.3. CRC CARDS

StoreScene	
Subclasses:	None
Superclasses:	GameScene
Responsibilities:	Collaborators:
Showing labels and performing right action upon click Make next level Switch scene to next level	LabelEntity PlayScene Engine

PlayScene	
Subclasses:	None
Superclasses:	GameScene
Responsibilities:	Collaborators:
Creating game entities Creating the store Switch scene to the store	Entity StoreScene Engine

2

Design Patterns

2.1. Description Design Patterns

2.1.1. Observer pattern

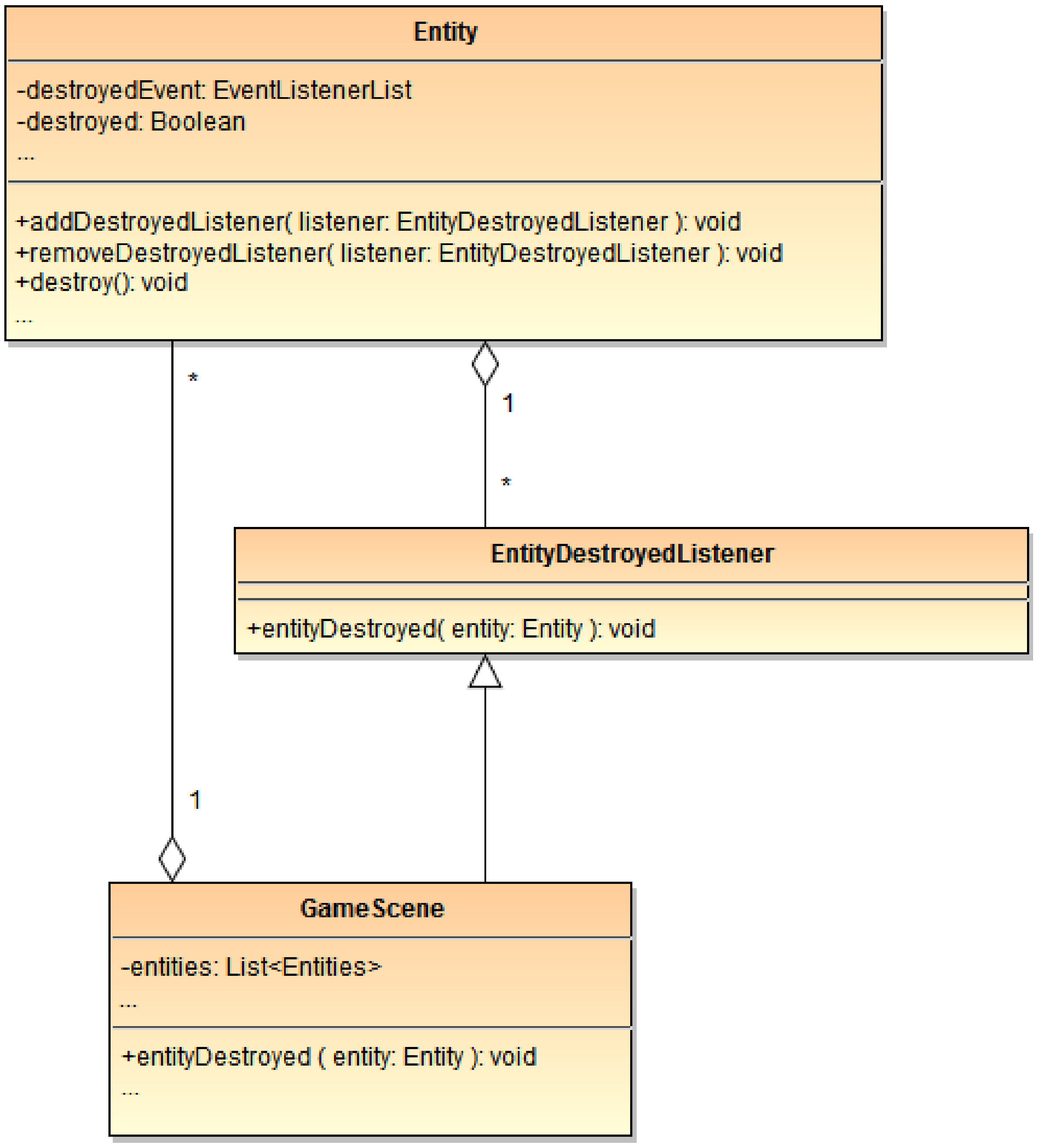
During the design of the space invaders game we tried to structure the classes as a directed acyclic graph: The engine has a reference to a scene, a scene has references to entities. We also tried to place business logic like collision handling as deep down as possible to ensure less coupling. This resulted in the problem that an Enemy needs to notify the Scene when it's shot to allow the Scene to remove the entity. Because an Entity shouldn't have a reference to it's containing Scene, we implemented a remove event that the Scene or other entities can subscribe to. We implemented this using the Java EventListenerList and an EntityDestroyedListener interface listeners must conform to.

2.1.2. Strategy pattern

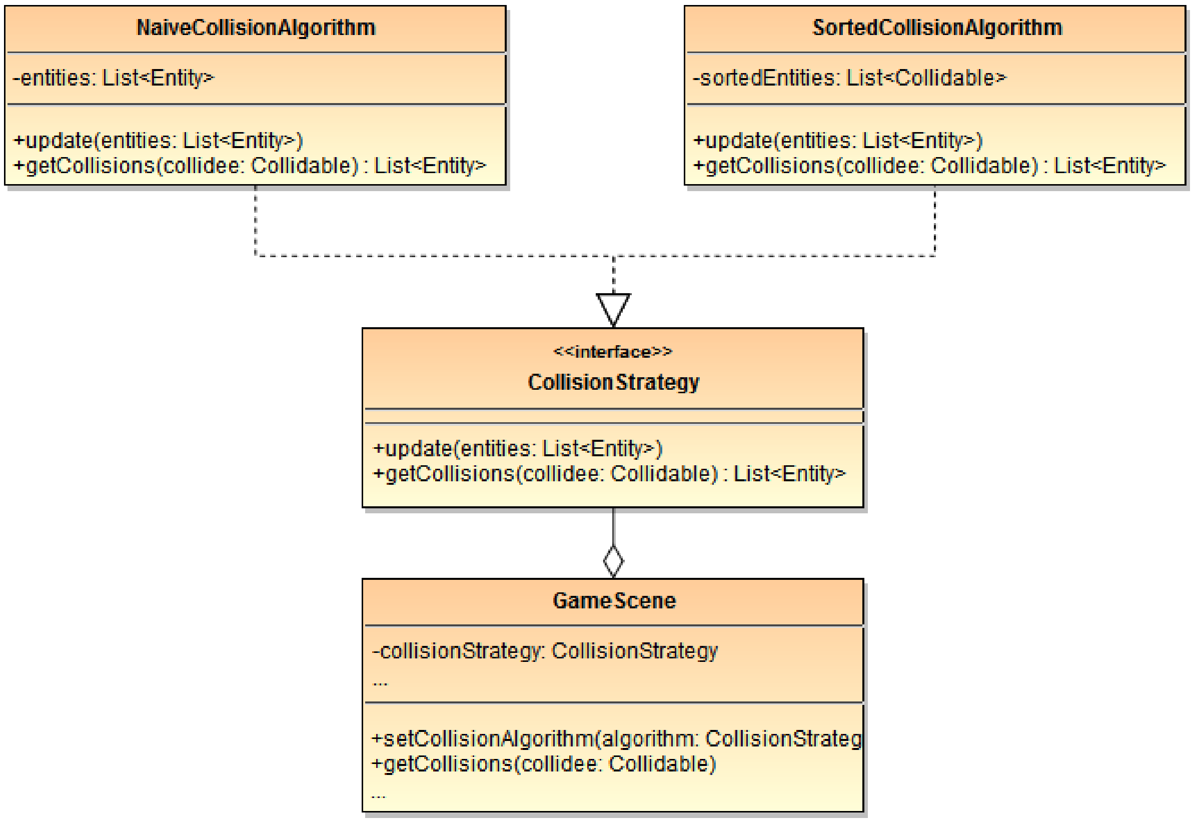
Our initial algorithm used to calculate collisions tried every possible combination resulting in a lot of comparisons that didn't use the branch predictor well. We added an algorithm that sorts the list of Entities on their x position before checking for collisions which reduces the amount of checks needed to calculate collisions and results in better branch prediction performance. Because this algorithm takes time to sort all entities during the update function we decided to allow a scene to revert back to the naive algorithm which only takes time during a collision check. To allow a scene to use a different algorithm we implemented an interface which collision algorithms must conform to and implemented a class for each algorithm.

2.2. CLASS DIAGRAMS

2.2.1. OBSERVER PATTERN

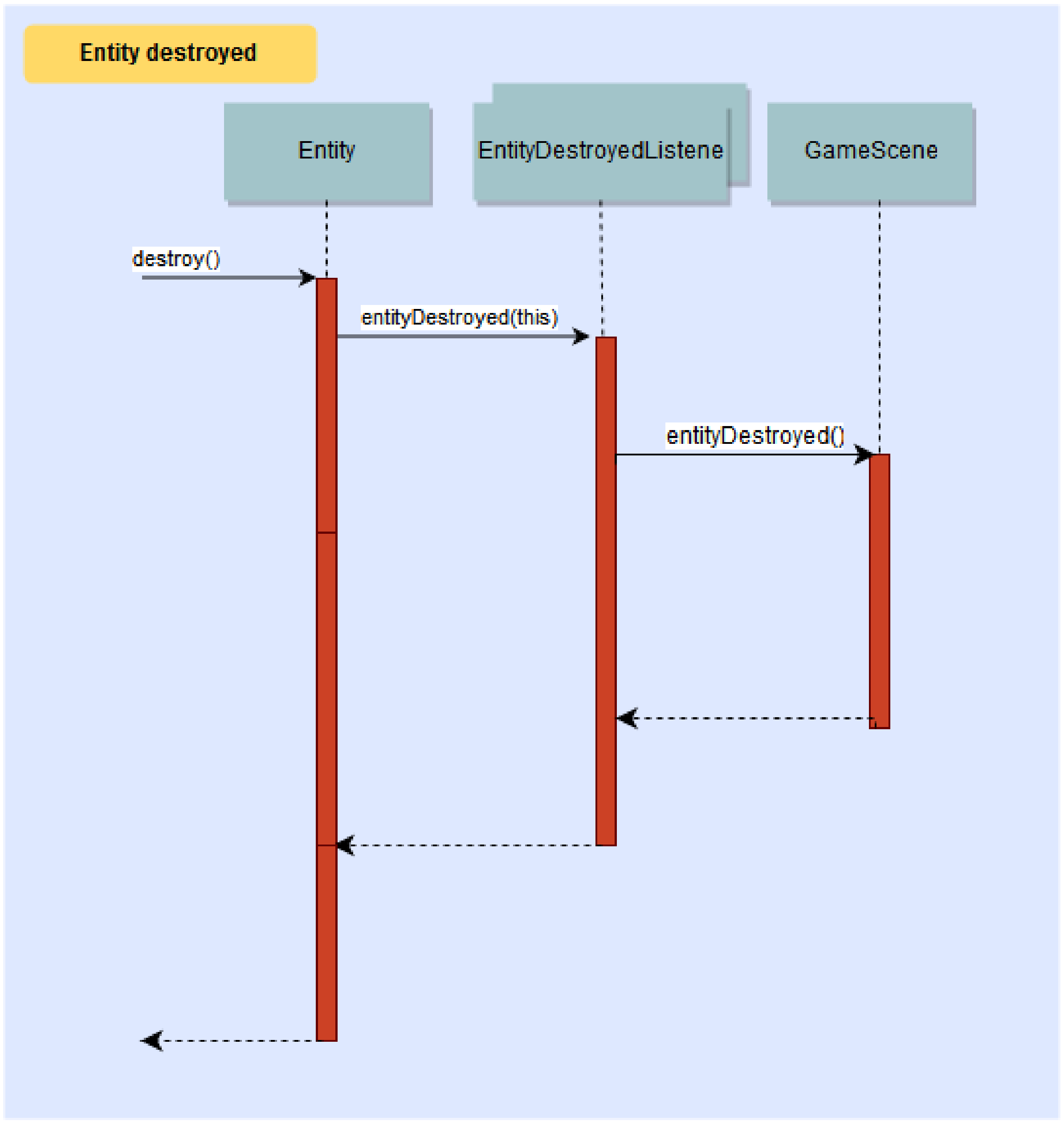

Entity
-destroyedEvent: EventListenerList
-destroyed: Boolean
...
+addDestroyedListener(listener: EntityDestroyedListener): void
+removeDestroyedListener(listener: EntityDestroyedListener): void
+destroy(): void
...
*
1
*
EntityDestroyedListener
+entityDestroyed(entity: Entity): void
1
GameScene
-entities: List<Entities>
...
+entityDestroyed (entity: Entity): void
...

2.2.2. STRATEGY PATTERN

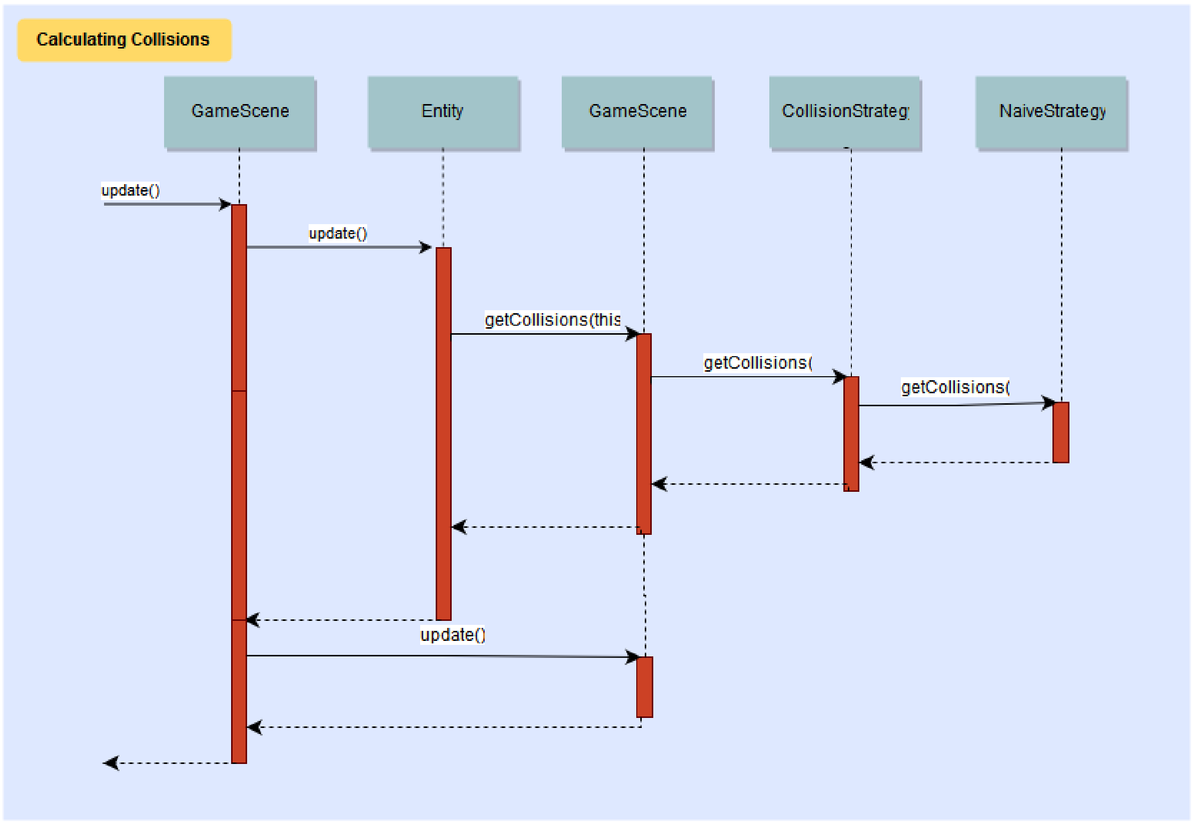

NaiveCollisionAlgorithm
-entities: List<Entity>
+update(entities: List<Entity>)
+getCollisions(collidee: Collidable) : List<Entity>
SortedCollisionAlgorithm
-sortedEntities: List<Collidable>
+update(entities: List<Entity>)
+getCollisions(collidee: Collidable) : List<Entity>
<<interface>>
CollisionStrategy
+update(entities: List<Entity>)
+getCollisions(collidee: Collidable) : List<Entity>
GameScene
-collisionStrategy: CollisionStrategy
...
+setCollisionAlgorithm(algorithm: CollisionStrateg
+getCollisions(collidee: Collidable)
...

2.3. SEQUENCE DIAGRAMS

2.3.1. OBSERVER PATTERN



2.3.2. STRATEGY PATTERN



3 Software Engineering Economics

3.1. Recognition of good and bad practices

The recognition of good and bad practices consists of 4 parts. The recognition depends on a dataset of finished projects and on project factors.

1. In the first part the overall average of all projects in our dataset is analyzed using project size, costs and duration measured in function points, euros and months, respectively.

2. Following part 1, we determine which project is a good practice and which project is a bad practice. A project is good practice when the performance on both cost and duration is better than the average cost and duration. Ofcourse this is corrected for the applicable project size. A project is bad practice when the performance on both cost and duration is worse than the average cost and duration. Again this is corrected for the applicable project size. To do this determination we classify the projects in a Cost/Duration Matrix.

3. After the classification we analyze how the project factors are related to the four quadrants of the Cost/-Duration Matrix. The outcome of this analyses is a percentage based on number projects per quadrant for every project factor and a percentage based on cost per quadrant for every factor.

4. The fourth and final part of this practices recognition analyzes specific subsets per research aspect. A research aspect is defined as strongly related when the percentage Good Practice or Bad Practice is 50% or more. In this part we ask the question: "What factors should be embraced and what factors should be avoided when composing a project portfolio?".

3.2. Visual Basic as good practice

'Visual Basic' is not so interesting, because the conlusion of this Good Practice is based on a diverse, small set of projects within two different companies.

3.3. Factors and their practice quadrant

- Managing Requirements

  Requirements are dynamic. They do not stay the same. Different stakeholders can change the requirements at any time. The stakeholders agreement is on what the system should do and not how. The how-part is something for the developers to figure out. That is why Managing Requirements is a Good Practice Factor.

- Visually Model Software

  Visual modelling improves our ability to manage software complexity. This also captures the structure and behavior of the different components. furthermore Visualisation of software promote unambiguous communication. That is why Visually Modelling software is a Good Practice Factor.

- Insufficient testing

  Testing is the base of each Software Project. If there is not enough testing involved, it is not guaranteed that the code is from decent quality. Also the number of defects present in a project where there is no testing done, is higher than the defects in a project where testing is a main concern. That is why Insufficient testing is a Bad Practice Factor.

3.4. THREE BAD PRACTICES IN DETAIL

- Dependencies

  Dependencies often relates to complexity. When a project depends on X, the whole project fails if X is not working anymore. So it is bad have large dependencies.

- Once-Only

  Once-Only is the opposite of release-based. Instead of delivering a working version once in a while, we deliver a working version only once. This Once-Only release is more vulnerable than a Release-based release, because each release is better than the previous release. This is called software-evolution.

- Many team changes

  When a team starts a project everyone is up to date on what they are working on. But when suddenly a team member drops out, a part of the tasks must be distributed to the remaining team members. This raises the work load on the members. This is hard to solve, because adding a new member is not a solution. When a new member comes in to the group, this member must be informed of everything that is going on.

Bibliography

[1] S. Ash, *Moscow prioritisation briefing paper,* DSDM Consortium (2007).