Assignment #3

Week 6

Group 2 - Bubble Bobble

Remi Flinterman
4362950, remiflint@live.nl

Arthur Guijt
4377338, a.guijt@student.tudelft.nl

Laurens Kroesen
4350286, l.kroesen@student.tudelft.nl

Thomas Overklift
4080890, t.a.r.overkliftvaupelklein@student.tudelft.nl

Lars Stegman
4365801, l.s.stegman@student.tudelft.nl

TI2206 Software Engineering Methods
of the Computer Science curriculum
at the Delft University of Technology.

Supervisor: Dr. A. Bacchelli
Teaching Assistant: Aaron Ang

Table of Contents

Exercise 1 – 20-Time, Reloaded

Part A – Bubbles float up after a set amount of time

This week we decided that we wanted bubbles to start floating up after a set amount of time and not just after colliding with an enemy or a wall. The requirements for this feature are stated below.

1 – Requirements

1.1 Must Haves

- A bubble that's floating upwards shall not be able to capture an enemy.
- A bubble shall start floating upwards after 3 seconds.
- A bubble shall start floating upwards after colliding with a wall.
- A bubble shall start floating upwards after colliding with an enemy.
- A bubble shall not collide with another bubble.
- A bubble shall keep moving as before after colliding with the player.

1.2 Should Haves

- A 'warp' shall be a unit moving from the top of the bottom to the map through a gap.
- A 'warp' shall be a unit moving from the bottom of the top to the map through a gap.
- A bubble that's floating upwards shall be able to warp.

1.3 Could Haves

- A bubble shall start floating upwards slowly with a curve.

1.4 Won't Haves

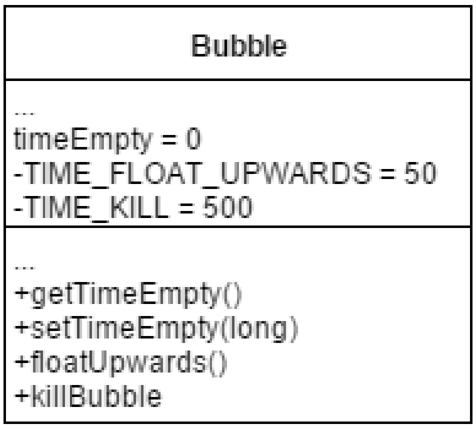
- -

2 – RDD, UML & explanation

To implement this feature we made empty bubbles change directions after a set amount of time. The bubbles start floating upwards after about 2 seconds and after approximately 15 seconds they pop.

Once a bubble starts floating upwards it cannot collide with enemies anymore, unlike when it's moving horizontally, in which case the bubble captures the enemy on collision. If a bubble hits a wall, it will immediately start floating upwards, regardless of how long it has been 'alive'.

Right now this feature uses 'ticks' to determine when a bubble starts floating upwards. Ticks are however not an absolute measure of time: the amount of ticks that can occur per second can be altered. Therefore it would be unwise to depend on this in the future. An implementation of this feature that uses seconds isn't functional yet, so it has left out of this release.



Part B – Units shouldn't be able to 'hang' onto walls

The last version of the game introduced a bug where units are able to hang onto walls. Especially enemy units seem to get stuck in a wall, and won't fall down. This week we're aiming to resolve this issue. The requirements for this fix are stated below.

1 – Requirements

1.1 Must Haves

- Units shall fall down after colliding with a wall.

1.2 Should Haves

- -

1.3 Could Haves

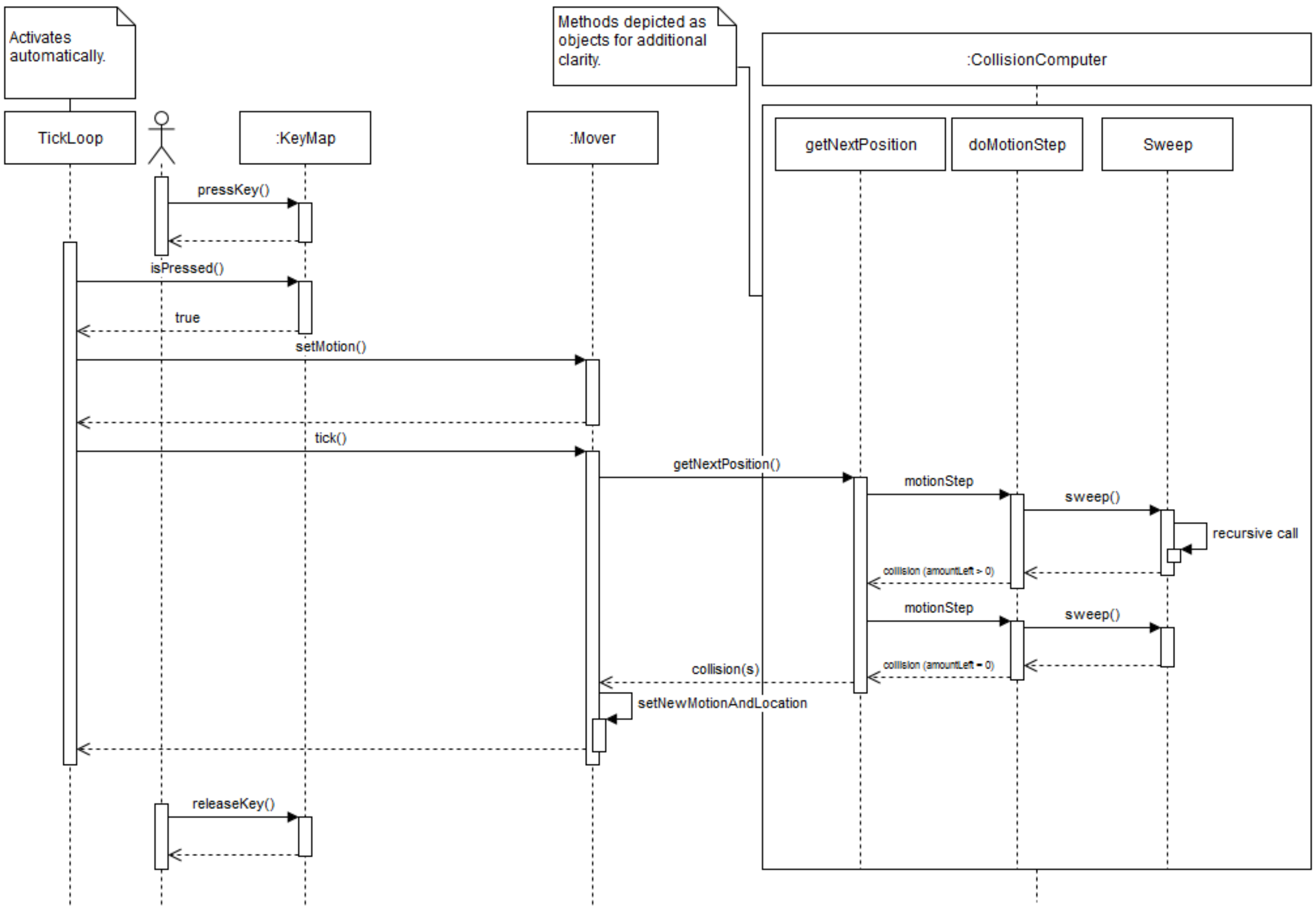
- -

1.4 Won't Haves

- -

2 – RDD, UML & explanation

Since this part of the exercise was a bug fix, there are no CRC cards or new classes. We decided to add a sequence of the collision handler in order to show what happens every tick in a more clear way than by simply describing it.



Part C – A unique sprite for captured enemies

Last week's release contained a feature where enemies could be captured by bubbles. That was really nice except that there was no unique sprite for captured enemies. That caused quite some unwanted confusion. This week we aim to implement a unique sprite for captured enemies, so they can be detected at all times! The requirements for this feature are stated below.

1 – Requirements

1.1 Must Haves

- The enemy captured by a bubble shall be represented by a unique sprite.
- An enemy's sprite shall be returned to its previous sprite when it breaks out of a bubble.

1.2 Should Haves

- The bubble sprite shall change on basis of the enemy captured.

1.3 Could Haves

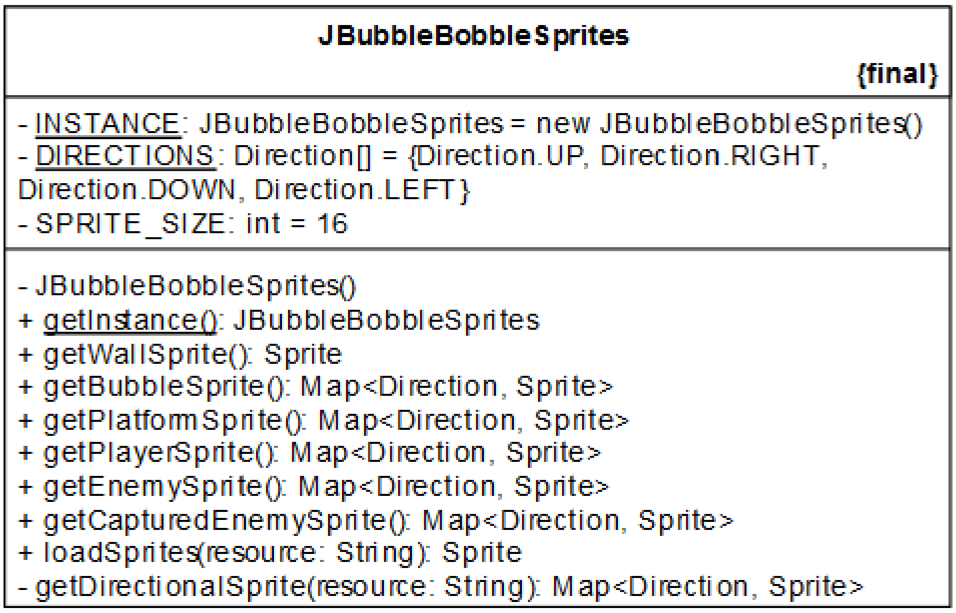
- -

1.4 Won't Haves

- -

2 – RDD, UML & explanation

We did the implementation of this feature together with the creation of a singleton class pattern. The use of this pattern is explained in some more detail in exercise 2a.
Because adding some extra sprites didn't cause the creation of any new classes any other spectacular things there is no CRC for this part. We did create an in depth UML of the JBubbleBobbleSprites class in order to show all the methods that (load and) return the different sprites.



```
JBubbleBobbleSprites
{final}

- INSTANCE: JBubbleBobbleSprites = new JBubbleBobbleSprites()
- DIRECTIONS: Direction[] = {Direction.UP, Direction.RIGHT,
Direction.DOWN, Direction.LEFT}
- SPRITE_SIZE: int = 16

- JBubbleBobbleSprites()
+ getInstance(): JBubbleBobbleSprites
+ getWallSprite(): Sprite
+ getBubbleSprite(): Map<Direction, Sprite>
+ getPlatformSprite(): Map<Direction, Sprite>
+ getPlayerSprite(): Map<Direction, Sprite>
+ getEnemySprite(): Map<Direction, Sprite>
+ getCapturedEnemySprite(): Map<Direction, Sprite>
+ loadSprites(resource: String): Sprite
- getDirectionalSprite(resource: String): Map<Direction, Sprite>
```

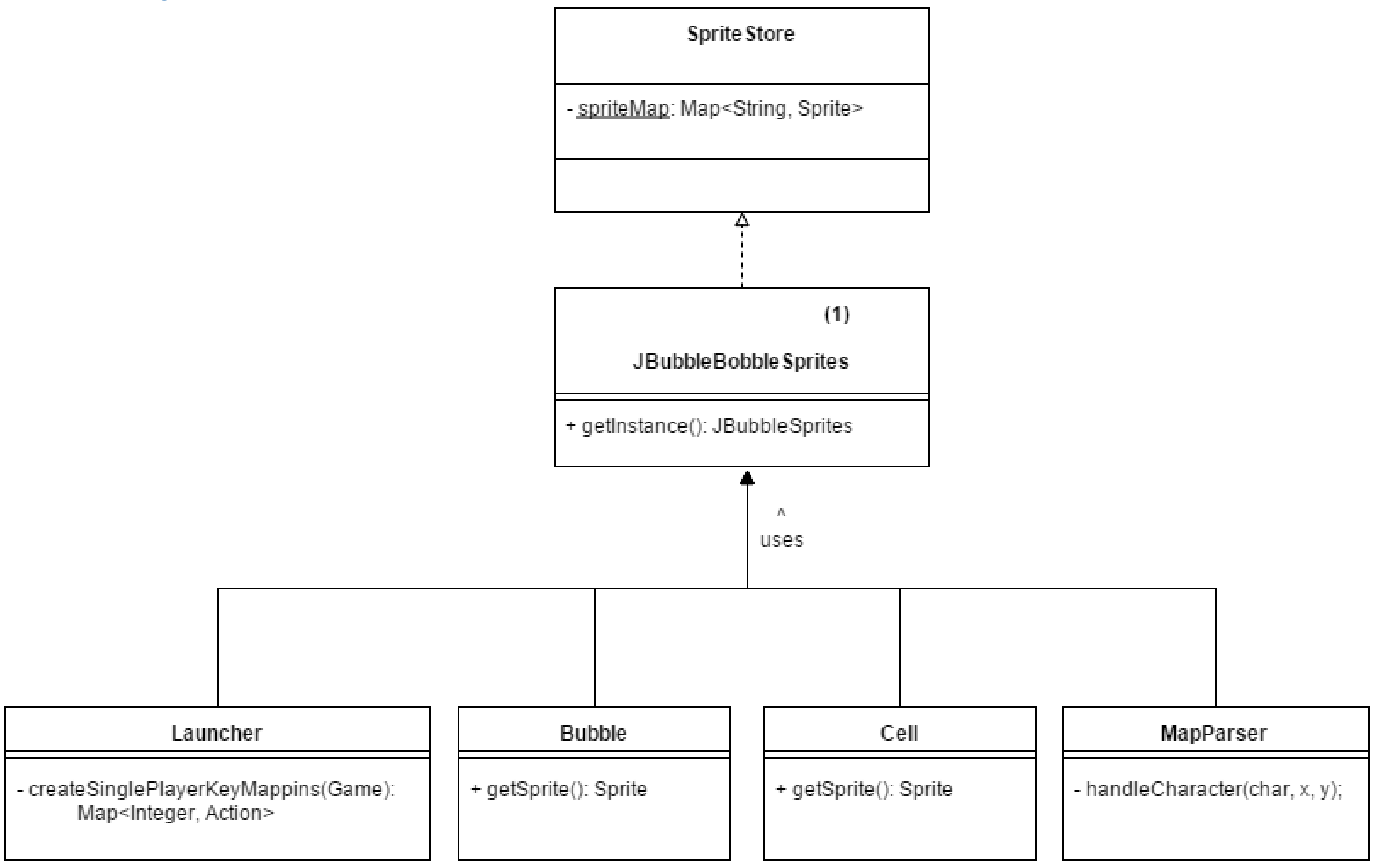
Exercise 2 – Design Patterns

Part A – The singleton pattern

1 – Explanation in human language

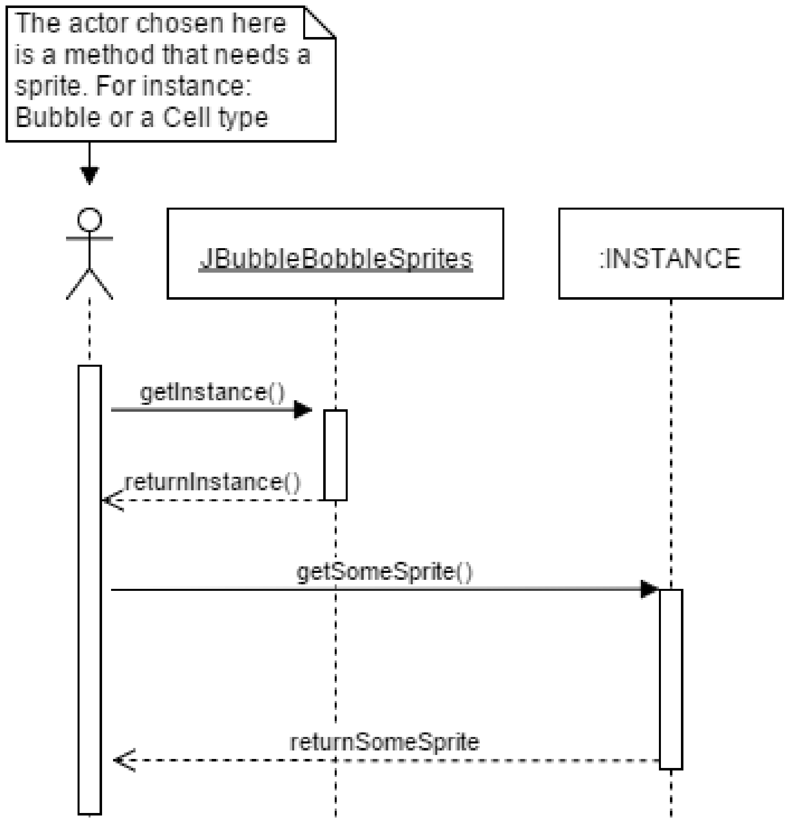
For the sprites that are used in the game we created a class using the singleton pattern.
There is a so called spriteStore in which all of the sprites are stored after they have been created the first time; this is basically a map where all Sprites are stored with their 'names' as key values. The class JBubbleBobbleSprites is an extension of this class that uses the singleton pattern. This means we enforce that there is only one instance JBubbleBobbleSprites at all times. This makes sure no resources are wasted loading (duplicate) sprites over and over again in a new map, they can just be returned using the existing map.
Apart from the class discussed above there are two other classes that also follow the singleton pattern: MusicPlayer & AudioDetector.

2 – Class Diagram



SpriteStore
- spriteMap: Map<String, Sprite>
(1)
JBubbleBobbleSprites
+ getInstance(): JBubbleSprites
^ uses
Launcher
- createSinglePlayerKeyMappins(Game): Map<Integer, Action>
Bubble
+ getSprite(): Sprite
Cell
+ getSprite(): Sprite
MapParser
- handleCharacter(char, x, y);

3 – Sequence Diagram


The actor chosen here is a method that needs a sprite. For instance: Bubble or a Cell type
JBubbleBobbleSprites
:INSTANCE
getInstance()
returnInstance()
getSomeSprite()
returnSomeSprite

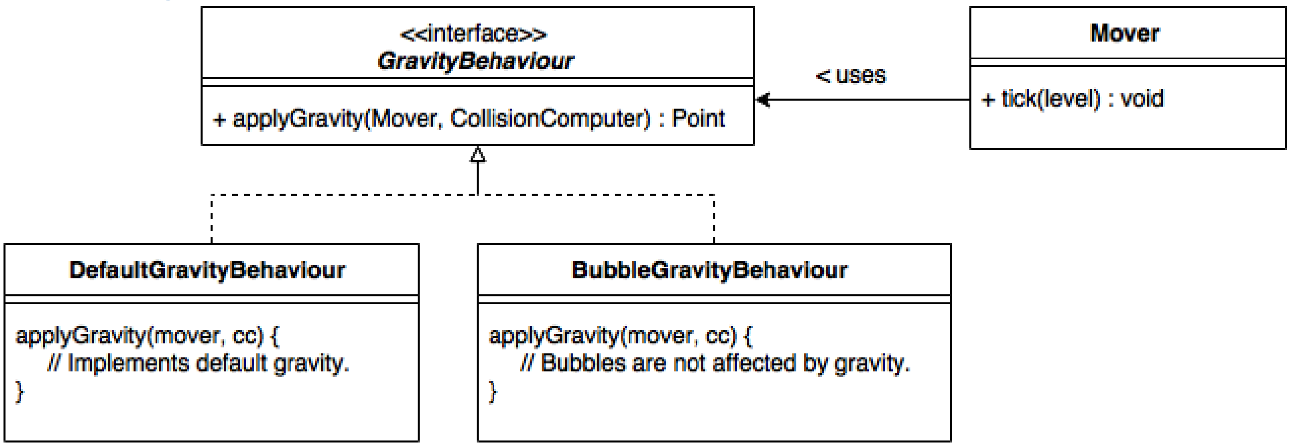
There are several methods in different classes that use the singleton class so below there's a somewhat generalized sequence diagram. For some sprites directions also have to be defined this is not taken into account here.

Part B – The behavior pattern

1 – Explanation in human language

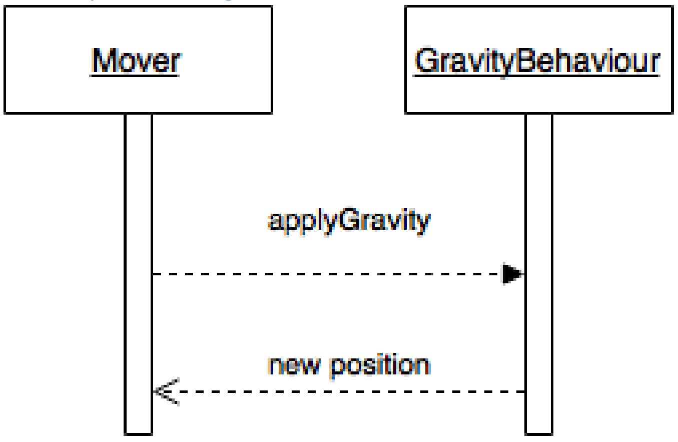
We created the behavior pattern for our tick methods so that different Units, like Bubble, Player or Enemy, can be affected differently by gravity. This might come in handy when we decide to create a level with inverted gravity or a low gravity environment. Bubbles are not affected by gravity, thus they need different behavior than Players and Enemies.

2 – Class Diagram



<<interface>>
GravityBehaviour
+ applyGravity(Mover, CollisionComputer) : Point
< uses
Mover
+ tick(level) : void
DefaultGravityBehaviour
applyGravity(mover, cc) {
// Implements default gravity.
}
BubbleGravityBehaviour
applyGravity(mover, cc) {
// Bubbles are not affected by gravity.
}

3 – Sequence Diagram



Exercise 3 – Software Engineering Economics

1 – Good and bad practice

In the paper it is explained that projects are categorized in a good / bad practice matrix with two axis: cost and duration. Projects are considered 'good / successful' if they were *cheaper* (in euro's) and realized in *shorter time* (in months) than the average project. These numbers are scaled by the *size* of the project (in function points). Otherwise larger projects would always end up in the bad-bad quadrant.

After the projects have all been placed in a quadrant of the good / bad practice matrix they are all analyzed too see which of 56 predefined 'project factors'[1] are applicable to each project.
These factors are then individually analyzed to see if there is significant correlation between a certain project factor and a quadrant in the good / bad practice matrix. If this is the case a project factor can be defined as good or bad practice.

2 – Visual Basic is overrated

The finding of the researchers that using Visual Basic as the main programming language during a project is a good practice is not so much uninteresting as it is unreliable. Of the 352 projects that have been analyzed, only 6 projects used Visual Basic. The fact that all of these projects were in the good-good practice quadrant of the matrix might have been heavily influenced by other project factors such as using agile structure and a release based model[1] or even sheer luck. The sample size is just too small to draw any reliable conclusion regarding this factor.

3 – Additional factors

It would be particularly interesting too look at the good practices in some more detail, in order to find out what *exactly* is making a project factor into a success factor.
For instance: the paper states that release based models and an agile structure (both imply a steady heartbeat) are success factors. These short project cycles clearly work better than using the waterfall model and / or a single release model. It might be worthwhile to look at the interval of these releases in more detail: are shorter sprints better than longer sprints? An additional project factor to measure this by might be *sprint length* or *time between releases / heartbeats*.

A difficulty in the method of measurement used by these researchers is that all projects are scaled using function points. Function points are of course somewhat standardized but only come into play after the requirements for a project have been defined. This means that if the requirements for a project are incomplete and / or vague the reliability of the function point estimate suffers. The quality of the project requirements could be measured by some extra project factors in this field:
Was there a *feasibility study* done before this projects? Has a *throwaway prototype* been made to improve the quality of the requirements? Doing either of these would be good practice

Another thing that has not been taken into account in the original 56 project factors is the design method(s) used in the projects. Has a project team made use of UML and / or other types of diagrams, and if so, what kind of diagrams, and in what stages of the project. These are of course very specific characteristics but a suitable project factor might simply be: *UML / diagram usage*, generally speaking using UML or other models is good practice because it improves communication within the project team and it can aid communication with the customer.

[1] H. Huijgens et al., "How to Build a Good Practice Software Project Portfolio?", p. 64 table 1.

4 – Bad practices

Of the 9 project factors that have been indicated as ‘failure factors’ the first three mentioned are:

1. Rules & Regulations (& Safety)
    - This is a bad practice because rules & regulations limit the amount of options for a project team. For instance: a project team might be forced to use a certain kind of format that is limiting the possibilities to store/manipulate/use certain data. A team might additionally be hindered by safety measures which they have to take into account, this complicates things even further. Two examples are: Digi-d (uses BSN) and banks (use IBAN). Both of these formats limit possibilities and require strict safety measures.
2. Dependencies with other systems
    - Dependencies are bad because they complicate things. This is something that can be said for practically all of the failure factors actually. Things are more costly in both time and money because they are complicated.
      Dependencies complicate things in particular because once again extra things have to be taken into account. Using data from another system, or perhaps writing to a database that another application uses in turn. You often have to adapt (to) existing software in these kind of cases.
3. New technology & framework solutions
    - Projects that try to create unprecedented content are in the failure zone; this is not because creating new things is bad, but because it is difficult. This is due to several factors. It is hard to estimate the cost for something new: there are no experts that can be consulted, and a team (while it might have ‘general’ experience) is unexperienced in this particular area when starting the project; the function point analysis might be unreliable and unexpected things happen more often than with a project that has several instances like it realized before. All these thing contribute to the fact that creating new technology takes more effort than reusing old things.