Software Engineering Methods - TI2206

Assignment 3

Group 47

Course Manager: Alberto Bacchelli
TA: Aaron Ang

Lillian Lehmann 4009037
Karin van Garderen 4144384
Christian Maulany 4097238
Rogier Vrooman 4001257
Fieke Miedema 4141369
Bas Böck 4366174

Exercise 1 – 20-Time, Reloaded (45 pts)

Requirements (improved):

Must have:

- In level 3 and higher, all enemies must be able to fire projectiles in the direction of the player, when the player is in ‘fire-range’ of the enemy
- The player must lose a life (or die when no lives are left), if the player collides with an enemy-projectile
- An enemy must be able to break free from its bubble, when being trapped in a bubble for 10 seconds
- A released enemy must be released at the same spot where the bubble was, and it must fall from there to the nearest platform underneath
- The walking speed of a released enemy must be increased by 2 points compared to the initial speed, for a duration of 10 seconds from the moment the enemy is released from the bubble
- A released enemy must be visualized angrily for a duration of 10 seconds from the moment the enemy is released from the bubble
- When a level is not cleared within 90 seconds, the magiron must appear
- When the magiron and the player collide, the player must lose a life (or die when no lives are left)

Should have:

- The magiron could float through platforms when moving towards the player
- The magiron should be an invincible enemy which will move towards the player
- The magiron should be visualized by a flying magikarp-Aaron-combination

Could have:

- /

Won’t have:

- /

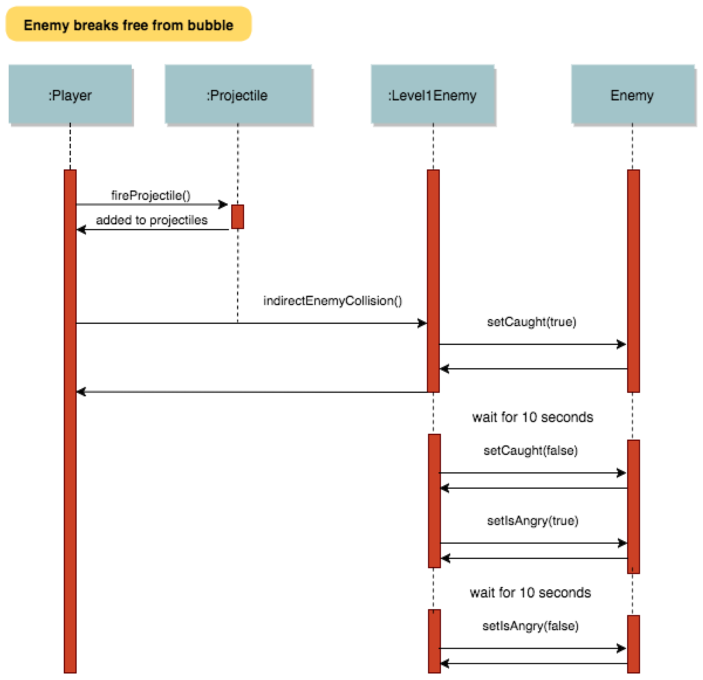
Magiron

The Magiron is an extension of Enemy. It only has a few adjustments, namely it can’t be caught by bubbles and it can float through platforms (since it it a ghost). Last it appears only after 90 seconds, if the level isn’t completed before this time.

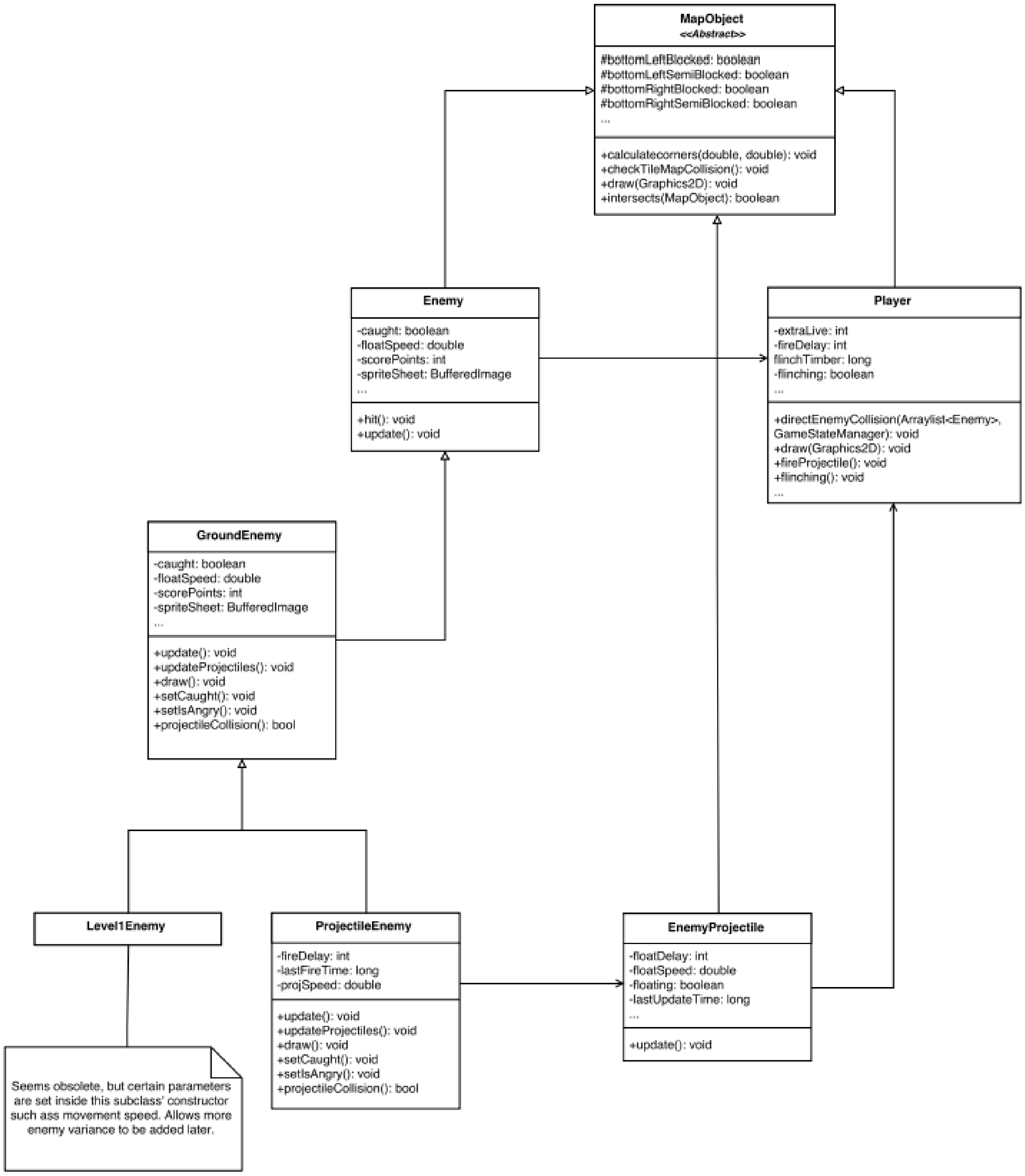
The Magiron is positioned and draw in LevelState, just like the other enemies. The 90 seconds delay is programmed in the Magiron class.

Things that can be added next sprint are the animation, Magiron is now represented by a heart, and the Magiron should move towards the player instead of moving randomly.

Sequence diagram for enemy breaking free from bubble



Class diagram for ProjectileEnemy



Exercise 2 – Design patterns (30 pts)

Iterator/composite pattern for drawing

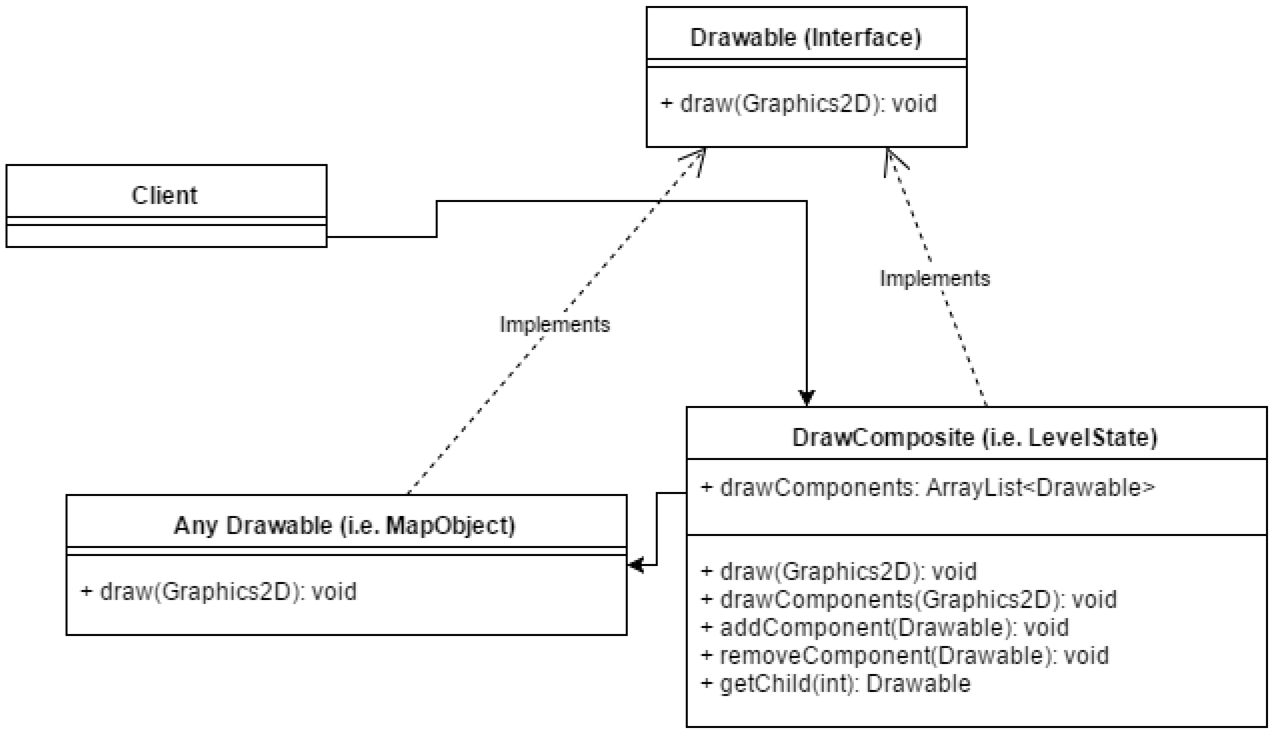
Natural language description

Currently, the logic of drawing elements to the screen involves a tree-like calling of subsequent draw() methods throughout many different classes. The most complicated part of this is the draw method in the LevelState class, which iterates over all MapObjects and the HUD. If we want to add or remove elements from the level, we would manually have to add or remove them here as well. The aim of implementing the Composite pattern is to make this tree-like process of drawing explicit, to enforce the ability to draw objects and to enable manipulation of the elements to be drawn from outside the draw() method, even in runtime. The only problem here is that the draw sequence does not follow a structure of composite objects. It goes from LevelState to MapObjects. Therefore, it is not completely applicable and we need to adjust the pattern slightly.

First of all, considering that so many classes can be drawn, we enforce this ability by having these classes implement an interface: 'Drawable'. It only contains the assumption that these classes have a draw() method.

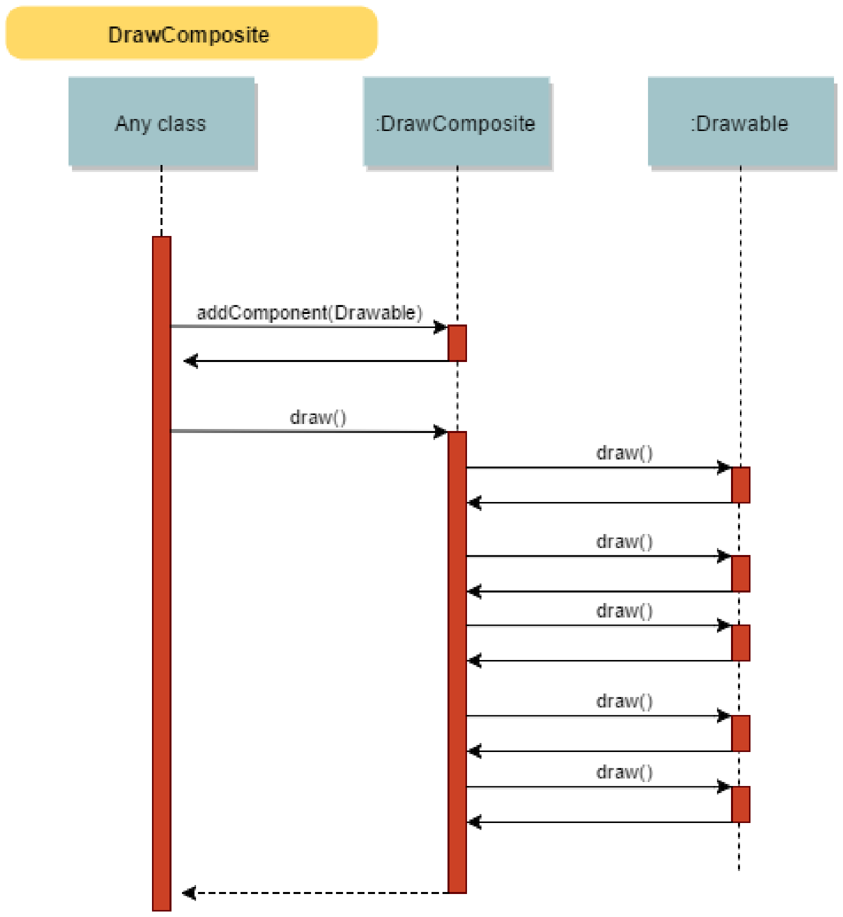
For the implementation of the Composite pattern we use the abstract class 'DrawComposite', which implements Drawable. The DrawComponent contains a list of Drawable objects, which it iterates over when drawingitself. Also, methods for adding and removing objects from the list are required. The abstract class is extended by GameState, so that its logic may be used by LevelState and any other state we might want in the future. Any GameState which does not need the logic of DrawComposite may simply override the draw method without calling the drawcomponents.

Class diagram


Drawable (Interface)
+ draw(Graphics2D): void
Client
Implements
Implements
DrawComposite (i.e. LevelState)
+ drawComponents: ArrayList<Drawable>
+ draw(Graphics2D): void
+ drawComponents(Graphics2D): void
+ addComponent(Drawable): void
+ removeComponent(Drawable): void
+ getChild(int): Drawable
Any Drawable (i.e. MapObject)
+ draw(Graphics2D): void

This implementation of the Composite pattern is different from the traditional form which was taught in the lectures. We have chosen not to implement the leaves of the tree as a separate class, mainly because these leaves can be many different sorts of classes. An interface can contain all the logic we need, because the only demand is that it contains a specific method. Not implementing the leaves means that there is no need for the Composite hierarchy to exist between Component and Composite. Only the Composite needs to contain logic, making one abstract class enough.

Sequence diagram



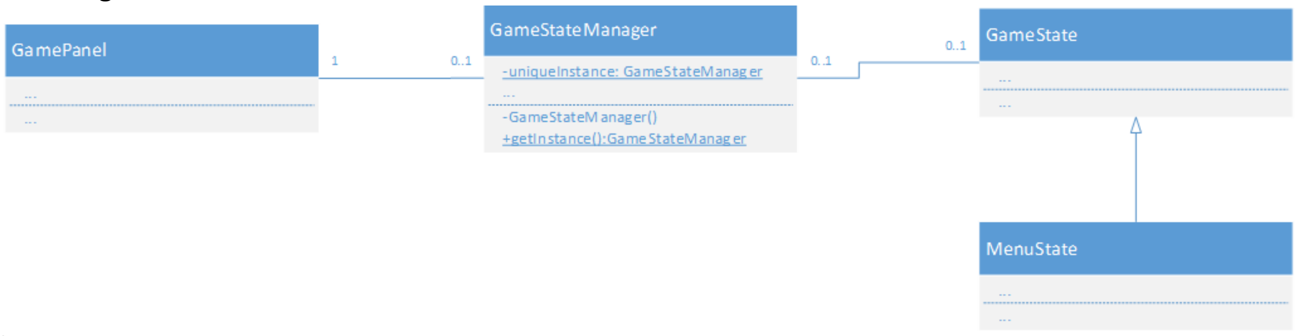
Although in our case there is only one instance of the DrawComposite class that makes use of the tree structure, this pattern adds to the flexibility of our system in two ways. Firstly, adding visible objects in the game is easier and they can be added and removed from any method. Secondly, this structure may be implemented in any other part of the draw sequence when the complexity of our system increases.

Singleton pattern for GameStateManager

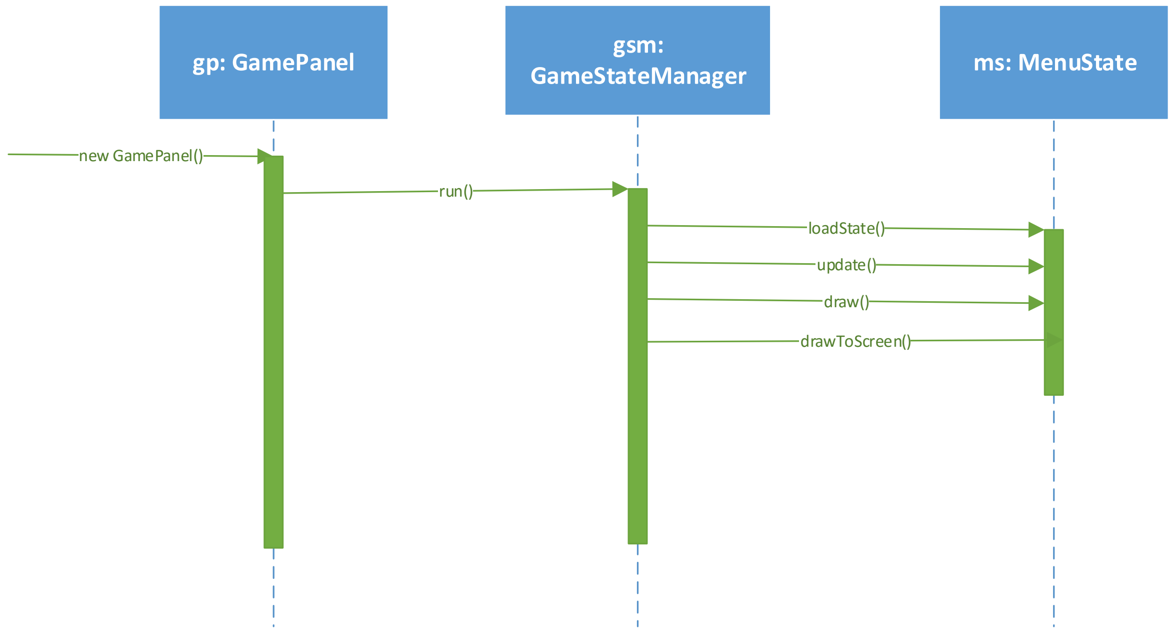
Natural language description

We have chosen to apply the Singleton design pattern to the GameStateManager class. GameStateManager should always have only one instance and be able to provide a global point of access to it. This because it manages the different gamestates and we don't want two instances of the GameStateManager do different things with the states at the same time. We implemented it by building a public static synchronized getInstance() method which returns the instance of a GameStateManager if it is not null, else it returns the already existing instance.

Class diagram



Sequence diagram



Exercise 3 – Software Engineering Economics (15 pts)

3.1 Explain how good and bad practice are recognized

In the research of Huijgens, van Solingen and van Deursen (2014) a good project is reckoned as good practice when the performance of the project is better than average on both cost and duration. Better than average means in this context better than the average cost and duration of the projects of the same project size in the total repository (of 352 finalized projects). When projects perform worse than average on both cost and duration, they are regarded as bad practice.

3.2 Explain why Visual Basic being in the good practice group is a not so interesting finding of the study

There are three reasons for this. First, naming a programming language as a success factor is a bit tricky, because choosing another programming language asks for a more long time approach and isn't easy to change for just one project. Second, the strong significance for Visual Basic is only found in 6 projects. Other factors strongly related to good practice occur in 18 projects or more. Third, it can be argued that Visual Basic projects environments are on average less complex than others, which make them more likely to end up in the Good Practice quadrant.

3.3 Enumerate other 3 factors that could have been studied in the paper and why you think they would belong to good/bad practice

Test-driven development

Test-driven development would be a good practice: it is seen as an integral part of software development and testing proactively is linked with a shorter time to market. When testing is regarded as a side-issue or the first thing to skip when the schedule is tight, you can end up with code full of errors that you wrote weeks ago. Since its effect on the project lead time, it would be interesting to study whether projects use test-driven development or not.

Software review

As we noticed in our own project, it is very useful to look through the code that other people have written. Not only does it makes you up to date with what other people are working on, somebody else sees different things than the maker of the code her-/himself. Karl Wiegers from IBM even calls software reviews '[...] one of the most powerful software quality practices available, all software groups should become skilled in their application'. This is why implementing software reviews can be seen as a good practice.

Defects management

In 172 of the projects, the number of defects was registered. It was however not analyzed if any of the projects used defects management. Defects management focusses on defect tracking and the chain of command when a defect is found. When proper processes, checks and testing is used as defects management, a project can be rolled out and completed with fewer unforeseen circumstances. Proper defects management can thus be regarded as a good practice.

3.4 Describe in detail 3 bad practice factors and why they belong to the bad practice group

Dependencies with other systems

When doing a project which relies on other projects or systems for its functionality, you will encounter a couple of problems. First of all, you need to understand the other project or system very well in order to implement your new project without disrupting the old project/system. Secondly, dependencies reduce the possibility of re-using a single module of your newly built system. Thirdly, dependencies can cause a domino-effect, in which a small error in one part of the system causes failure throughout the whole system because of all the dependencies. These are the reasons why dependencies with other systems is regarded as bad practice.

Once-only project

Once-only projects are in this research regarded as the opposite of release-based projects. In once-only projects, the team needs to do a lot of things for the first time and for one project only. They for example work with a different environment or a new programming language. When you do such things for the first time, you are bound to encounter problems you did not plan on. Because the projects are once -only, there is no learning curve in which you can use the knowledge and experience you gained in an earlie r project. This is why it is advised to never run once-only projects if you can do your project release-based.

Many team changes, inexperienced team

In small teams, but certainly in bigger teams, teamwork and co-ordination are required to successfully fulfill projects. This teamwork and co-ordination is halted by many team changes. Experienced people leaving and new people entering the team causes information to be lost and increases the time spent on learning how to work together, instead of working on the project. The same counts for working with an inexperienced team: whether they are inexperienced with the technology or inexperienced with working together in this setting, time is bound to be lost with learning (the technology or the teamwork) instead of doing. This is all why many team changes and/or an experienced is seen as bad practice.