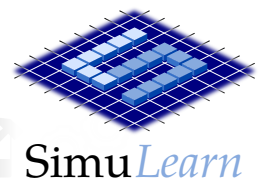


# BUILD - RUN IMPROVE - REPEAT

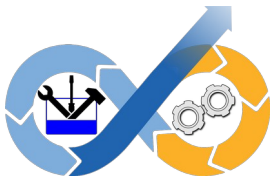
A game about implementing and improving your DevOps cycle



Facilitator's guide



this page is intentionally left blank



# Build – Run Improve – Repeat

A game about implementing and improving  
your DevOps cycle

## Table of Contents

1	Version history.....	3
2	Introduction.....	4
3	Concept of the game.....	5
3.1	Objective.....	5
3.2	Starting point.....	5
3.3	The performance levels of the DevOps-activities.....	6
3.4	Elements of the game.....	9
3.4.1	Board.....	9
3.4.2	Cards.....	9
3.4.3	Tokens.....	10
3.4.4	Dice.....	11
3.5	Having features implemented.....	11
3.5.1	Cadence.....	11
3.5.2	Create value, create revenue, invest.....	12
3.5.3	Don't go bankrupt.....	12
3.6	What can possibly go wrong?.....	12
3.6.1	Incident types.....	12
3.6.2	Incident severity.....	13
4	Let's play.....	13
4.1	Participants and responsibilities.....	13
4.2	Starting budget.....	14
4.3	Improving your way of working.....	14
4.4	Start working.....	15
4.5	Implementing features according to flow and queue size.....	15
4.5.1	Fast forward.....	17
4.5.2	Cutting corners.....	18
4.5.3	Ignoring technical debt.....	18
4.6	After each round.....	18
4.7	Dealing with reported security vulnerabilities.....	19
4.8	Dealing with incidents.....	19
4.8.1	Technical debt becomes an incident.....	20
4.8.2	Change failure.....	20
4.8.3	Count your losses.....	20
4.8.4	Fixing an incident.....	21
4.8.5	Accepting the risk.....	21
4.9	Finishing feature implementation work.....	22
4.9.1	Marking your progress.....	22
4.9.2	System aging.....	22



4.9.3	Need for refactoring.....	22
4.10	Income and investments.....	22
4.10.1	When can you invest?.....	23
4.10.2	Implementing improvements.....	23
4.11	The game ends.....	23
4.12	How can I win this game?.....	24
5	Possible scenarios.....	24
5.1	Dividing or sharing decisions, budget and costs.....	24
5.1.1	Dividing.....	24
5.1.2	Sharing.....	24
5.2	Start from performance level 0 or from your real performance level.....	25
5.3	Force the team to implement features faster (and cut corners).....	25
5.4	Work to implement improvements or not.....	25
6	Thank you.....	26
7	Contact information.....	26



# 1 Version history

Version	Date	Changes
0.1	2020-06-02	Initial draft version
0.2	2020-07-14	Processed feedback
0.3	2020-10-30	Improvements after tryouts <ul style="list-style-type: none"><li>• variability of incident cost</li><li>• extra operate aspects</li><li>• extra incident types</li></ul>
0.4	2020-12-27	Made a distinction between treating a CVE and an incident: different pawn
1.0	2021-02-20	First major version Implemented major improvements from previous tryouts: <ul style="list-style-type: none"><li>• technical debt introduced</li><li>• tokens with symbols instead of pawns</li><li>• accept risk of incidents instead of fixing them</li><li>• implement improvements</li></ul>
1.1	2021-08-18	Added follow-up of implemented features + system aging/degrading, the risk of failing changes and the need for refactoring.



## 2 Introduction

At the time of the initial creation of this game I am part of a team that introduces and supports cloud native development. My role is to take care of DevOps-related processes and responsibilities, like testing, security and release management. I don't remember what exactly happened that particular day when I got the initial idea to make a game about DevOps. It probably had to do with some misconceptions about DevOps and CI and CD in particular. That event made my mind wander... Wouldn't it be great to make a DevOps game, to deal with these misconceptions?

One of these misconceptions about DevOps is that people often think in the first place about delivering faster to production. That is only a small part of the story: Continuous Delivery (CD) is always preceded by Continuous Integration (CI). In other words: first focus on delivering quality then on delivering faster – and beyond off course, operate and monitor. But DevOps is more than CI and CD. It all starts with the mindset. This game focuses on the full DevOps cycle:

- The development part of the cycle is not new. We know it from eXtreme Programming: from Plan, over Code and Build to Test. This part focuses on delivering quality and get fast feedback from the code you checked in. This is the core of agile software development.
- The Operations part closes the cycle. This is the new part that builds upon the foundations of quality software development, with continuous automated testing as the linking pin to be able to Release, Deploy, Operate and Monitor.

Since the left part of the cycle is the oldest, focusing on delivering quality, which is the part you need to invest in first? Exactly: the left part of the cycle. Doing the wrong investments, or implementing things in the wrong order will have some pretty bad consequences. And that is exactly what this game aims to do: experience the impact of taking the wrong decisions when implementing DevOps in your organization.

This is the 2<sup>nd</sup> game I make, after the Scrumban simulation. I know that the best known DevOps related simulation is *The Phoenix Project Game*, based on the bestseller novel “The Phoenix Project”. It covers the whole spectrum of DevOps, not only the automation part. But The Phoenix Project Game takes at least half a day and preferably a full day to do, which is a lot of time, and you need to attend a train-the-trainer session first, before you can facilitate that simulation. It also aims at management and business understanding of DevOps principles. This is not my objective: I wanted to make a game that focuses specifically on technical aspects of DevOps, that can be done in a team within a time slot of an hour and a half or 2 hours.

I definitely hope you like it and you can use it in your own professional context.

Koen Vastmans, SimuLearn  
October 2019 – August 2021



## 3 Concept of the game

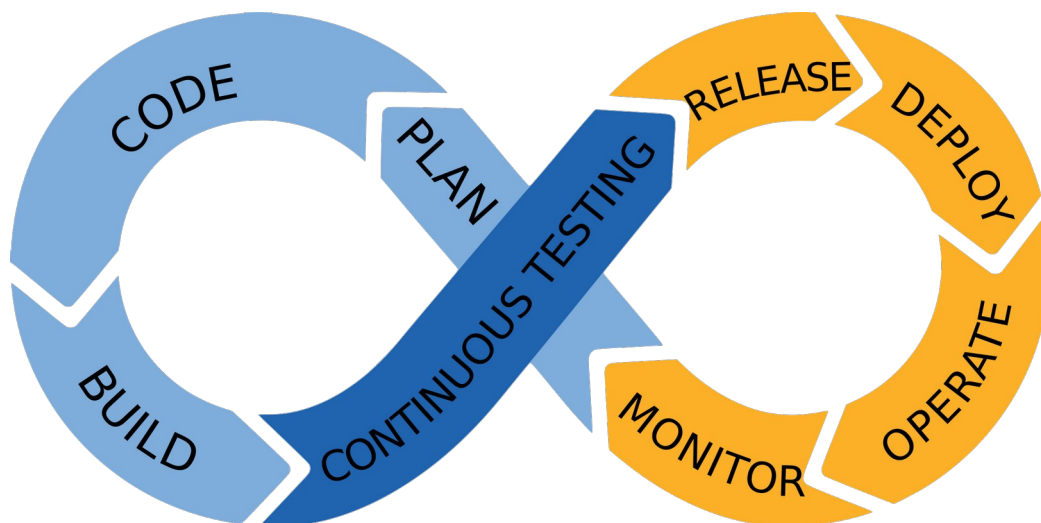
### 3.1 Objective

The objective of the game is to experience how improving your development and operations activities and evolving to a DevOps approach will speed up the delivery of value and also improves the quality of what you deliver, which dramatically reduces the risk of service unavailability caused by bugs or security breaches. Initially these improvements will go slow, but once you have reached the tipping point, delivery and improvement intervals will become shorter and shorter. That's exactly what thorough research on the findings of the annual DORA report has discovered.

Participants of the game will also experience that doing the wrong investments – in the wrong order, that is – will have the opposite effect on your quality.

### 3.2 Starting point

The concept of this game is based on the infinite loop of DevOps:

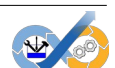


The colors of the loop make a distinction between typical development activities (in blue) and typical operational activities (in orange). The darker blue arrow in the middle, Continuous testing, is the linking pin to smoothly go from development to operations.

Each stage of the loop has a number of activities and each activity has 4 performance levels, from 0 to 3, similar to low, medium, high and elite performers as mentioned in the DORA report and the book “Accelerate – The science of DevOps” (subtitled “Building and Scaling High Performing Technology Organizations”) from Nicole Forsgren, Jez Humble and Gene Kim.

The starting point of the game is a board with all activities on performance level 0. This means:

- you work in a typical project oriented waterfall approach
- with strict separation of duties between development and operations activities
- with lots of manual activities
- and limited quality controls



### 3.3 The performance levels of the DevOps-activities

You can step up the performance ladder in each activity by investing (part of) your profit in improvements.

These are all the activities/aspects of the development stages and their maturity level:

Stage	Activity	Performance level 0 Low performers	Performance level 1 Medium performers	Performance level 2 High performers	Performance level 3 Elite performers
<b>Plan</b>	Approach	Project approach with big specification upfront (“waterfall”)	Iterative project approach (agile/”Scrum” principles & techniques)	Iterative product approach, driven by product backlog	Flow of changes (Kanban-style)
	Team	Separate build and maintenance team	technology knowledge divided over different teams (change request driven)	all required knowledge in the team	generalizing specialists
	Work visualization	Nothing	only planned work	only own work	all work is visualized
<b>Code</b>	Quality	Nothing	Coding guidelines	Manual reviews	Automatic code scans
	Versioning	Nothing	Just versioning	Branching per release	Trunk based development
	Security	Nothing	secure coding guidelines	External library scans	code scans (SAST tool)
<b>Build</b>	Approach	Manual build	Scheduled nightly	scheduled every hour	upon commit
	Breakers	compilation errors	failing unit tests	code quality scans	Security scans
<b>Test</b>	Approach	Manual testing in UAT	Automated functional tests (unit, integration)	Automated non-functional tests (stress, load)	Chaos injection in production



Stage	Activity	Performance level 0 Low performers	Performance level 1 Medium performers	Performance level 2 High performers	Performance level 3 Elite performers
	Responsibility	Business is responsible	Business and IT all do their part of the tests (overlap)	Business and IT do their part of the tests (no overlap)	Shared responsibility: business decides what to test, IT decides how to test it
	Security	Nothing	Penetration testing before going to production	Recurring penetration testing	DAST tool continuously scans behavior or running application and reports vulnerabilities

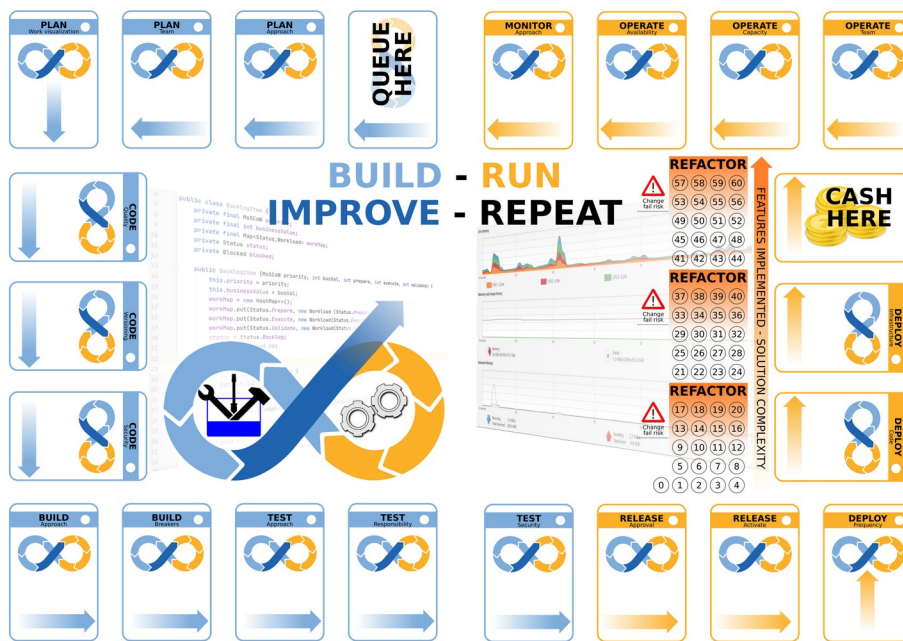
These are all the activities/aspects of the operations stages and their maturity level:

Stage	Activity	Performance level 0 Low performers	Performance level 1 Medium performers	Performance level 2 High performers	Performance level 3 Elite performers
<b>Release</b>	Approval	Separate release management team guarding over the planned changes and their impact	Business go-no go meeting	Product owner	Team – 4 eyes principle
	Activate	upon deploy	fixed date, via feature toggles	on demand Via feature toggles	On commit Only with trunk based development
<b>Deploy</b>	Frequency	Quarterly	Every month	Every week	Constant flow Only with Kanban-style plan approach and trunk-based development

Stage	Activity	Performance level 0 Low performers	Performance level 1 Medium performers	Performance level 2 High performers	Performance level 3 Elite performers
	Code	Manual deploy Separate team	Automated build	Automated deploy to test, manual approval for UAT, production	Automated deploy to production Only with Kanban-style plan approach and trunk-based development
	Infrastructure	Know your colleague: call the infra guy to do the changes	Get infrastructure changes via service request	Changes to infrastructure are done via self service	Infrastructure as code, part of your source code repository
<b>Operate</b>	Team	Segregation of duties	closer collaboration between dev and ops team	shared responsibility between dev and ops team	you build it, you run it E2E team responsibility
	Availability	Only 1 production instance	Cold standby	Hot standby	Load balancing and failover
	Capacity	No capacity management in place	Fixed capacity, based on historic usage and capacity metrics	Elastic (manually sized) capacity, based on historic usage and capacity metrics	Automated capacity management based on usage metrics and feedback
<b>Monitor</b>	Approach	Nothing	Information radiators followed up by a separate team	Automatic escalation to team members	Self-healing & self-learning system

## 3.4 Elements of the game

### 3.4.1 Board

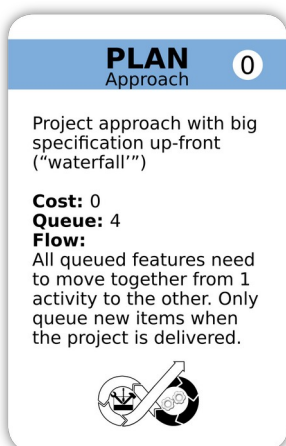


This is a board game, so the heart of this game is the board. The board represents all the different DevOps stages, visualized in the infinite loop. Each stage contains several activities. The board contains placeholders for all these activities. This is where you put the corresponding cards. At the start of the game you put all the cards of the zero performance level on the board.

### 3.4.2 Cards

The cards are the driving elements of the game. They represent your performance level. You start with the zero performance level of all activities of the DevOps stages (depends on the scenario you choose – see [Possible scenarios](#)). The investments you can do to improve your maturity are represented by a card you can buy and place at the activity.

Cards have 2 sides. You can only see the front side of the cards for now. The back sides can only be viewed in case of an incident. Each card front contains the following information:

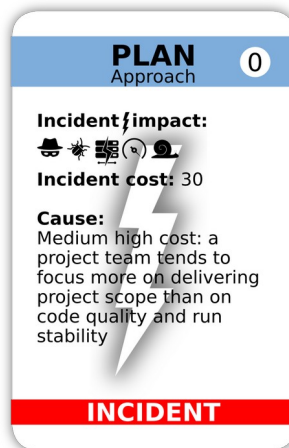


- The stage
- The activity
- The performance level + description
- The implementation cost
- The queue size – how many items need to be at that activity before you can on to the next activity
- The flow – restrictions about moving features from one activity to another



The back side of the cards contains:

- The stage
- The activity
- The performance level
- What can cause this impact:
  - a bug
  - a security breach
  - excessive load
  - a performance issue
  - a system outage
- The cost of fixing the incident
- An explanation of the cost



The reason why you should keep the cards face up, is because you cannot know what the cost of an incident will be up-front. This would influence your investment decisions and one of the aims of the game is also to experience what the impact of wrong decisions is.

You cannot immediately buy the performance level 3 card of an activity if you are still at level 0. You really need to go step by step: first invest in level 1 improvements, then level 2 and finally level 3. Why? Because each higher level builds upon the previous levels. But in which order you invest and where you invest more initially, is entirely your decision.

### 3.4.3 Tokens

Tokens represent work. There are different kinds of tokens, in different colors, with different symbols, representing different kinds of work:



Blue token  
A feature to be implemented



Green token  
An improvement to be implemented



Orange token – only this symbol  
A reported security vulnerability to be fixed





Orange token – different symbols  
Technical debt to be solved



Red token – different symbols  
Incident to be fixed

Tokens need to be moved across the board, according to the value of the die and the indicated flow and queue size. Features and technical debt follow the same cadence, as indicated in Plan – approach. Improvements can follow their own cadence. Incidents and reported security vulnerabilities need to be fixed as soon as possible.

### 3.4.4 Dice

There are 2 dice in the game:

- 1 normal die, for the number of moves each participant can make
- 1 special one, for the incidents



The normal die is simple: you throw a 1, you move 1 token, you throw a 6, you move 6 tokens.

The incident die has special symbols:



Reported security vulnerability



Bug



Security breach



System outage



Unexpected load



Performance issue

The usage of the die is explained in the paragraph [“What can possibly go wrong?”](#).

## 3.5 Having features implemented

### 3.5.1 Cadence

Each activity indicates the cadence: this means the number of tokens that need to be in the activity before you can move on to the next activity or stage. Typically in a project oriented waterfall approach, you move all the tokens one by one from Plan to Code. If all tokens are moved to the first activity of the Code stage, you move all the tokens one by one to the next activity of Code, and so on. Same with release and deploy: as long as you stick to longer deployment cycles, you cannot move individual tokens beyond the deployment stage, no matter how your Plan approach is.

When your performance improves, you will see that you get a better flow of features across the board, which will speed up the value creation and hence faster revenue, which allows more improvements of your DevOps activities.



### 3.5.2 Create value, create revenue, invest

Once a token is moved to the Cash Here spot, you can remove it from the board. This is where the value is created. Each feature that goes live (each blue token that arrives at the “Cash here” spot) will give you 100 extra credits, but only once all features to be implemented are delivered together (depending on queue size and flow). These credits can be invested in improvements to your activities. And this is where the tricky bit starts: invest wisely, because doing investments in the wrong order can seriously impact your stability and increase the cost of fixing bugs and security vulnerabilities.

### 3.5.3 Don't go bankrupt

Make sure you don't invest every credit you earn, because things can go wrong and that will cost you credits. And the last thing you want, is that you cannot afford the cost of fixing problems, because you will go bankrupt and the game ends because of your wrong decisions...

## 3.6 What can possibly go wrong?

### 3.6.1 Incident types

Well, everything can go wrong. And you better make sure that you have enough credits to pay the cost of solving these incidents. After playing a round (each participant moved tokens on the board), you roll the incident die, together with the normal die. In case of an even value, the incident occurs. There are 6 types of incidents:



A reported security vulnerability that needs urgent mitigation  
this will only cost you effort: you need to solve the issue first, instead of delivering new features. **Make sure you solve it before it becomes a security breach!**



A bug that needs to be fixed  
this will cost you both money and effort

- the effort is similar to the effort of mitigating a reported security vulnerability
- the cost is the sum of all impacts on the back of your cards on the board, according to your performance level and the severity of the incident



A security breach that needs to be fixed asap  
this will also cost you both effort and money

- the effort is again the effort to fix the incident
- the cost is again the sum of all impacts of your cards on the board



Your system is down that needs to be up and running again as soon as possible  
this will cost you both effort and money

- the effort is again the effort to fix the incident



- the cost is again the sum of all impacts of your cards on the board



Your system is unable to process the extra load  
this will cost you both effort and money

- the effort is again the effort to fix the incident
- the cost is again the sum of all impacts of your cards on the board



Your system experiences a performance issue  
this will cost you both effort and money

- the effort is again the effort to fix the incident
- the cost is again the sum of all impacts of your cards on the board

### 3.6.2 Incident severity

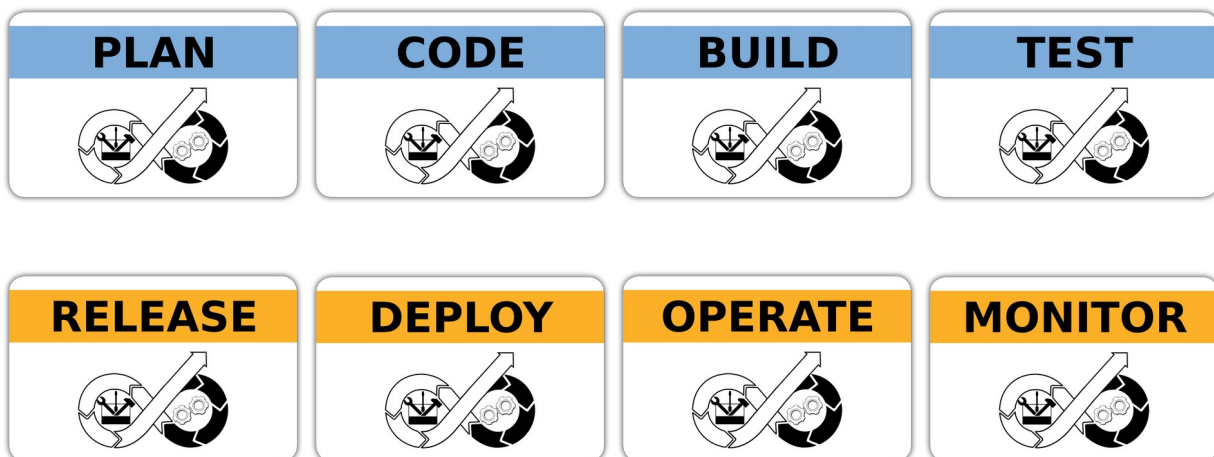
As far as financial loss of an incident is concerned: the normal die will also have a meaning here. Not only will the incident only occur in case of an even value. The value itself will determine the severity of the incident, and also the multiplier of the financial loss (see the paragraph [Count your losses](#) for details).

It is not because you are fixing an incident that takes several rounds, that nothing new can go wrong. You still need to roll the die at the end of a day of trying to fix an incident.

## 4 Let's play...

### 4.1 Participants and responsibilities

The game can be done with at least 2 and at most 8 participants. Each of the activities on the DevOps loop require an owner, someone who takes the responsibility for improving the performance level of the different aspects of that activity. Divide these responsibilities among the participants of the game. You need at least 2 participants, so that you can divide the responsibility between development and operations activities (easily recognizable by the color of the activity). You can assign ownership of an activity via the dedicated cards:

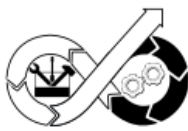


How you deal with these responsibilities, is up to you to decide (see [Scenarios](#)).

## 4.2 Starting budget

Each game gets a starting budget of 1000 credits. This is a shared budget for all stages, all activities. These credits need to be spent on both improving your performance level and on fixing incidents. If and how you divide the budget among the different activities, is entirely the responsibility of the participants.

If you have play money – like in Monopoly – you could use play money to make the spendings and earnings more tangible, especially if you choose to divide the budget over the different owners (see [Possible scenarios](#)). Someone will play the role of “the bank” then (like in Monopoly). But you can also use the available balance sheet:



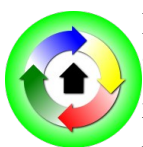
# Account balance

What	Debit	Credit	Balance
0. Initial balance			
1.			
2.			
3.			
4.			

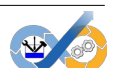
The sheet contains an initial balance line and lines for each spending and earning (separate columns) and the resulting balance. In case you choose to divide responsibilities and thus also budgets, each participant will have its own balance sheet, or 1 per responsibility domain.

## 4.3 Improving your way of working

At the start of the game the participants can decide what to improve first, before they start implementing features. Or they can simply start with the basic setup (mainly manual work) and try to create value first (and hope nothing goes wrong...). Just make sure that there are enough credits in case of an incident.



Depending on the time you can spend on the game, you can choose to either simply “buy” improvements and have them immediately available or buy and implement the improvements. At the start of the game you can simply buy your improvements, but once you start implementing features, all improvements need to be implemented ([if you choose to do so](#)). You take a green token and put it on the Queue here spot. They don’t have to be





part of the same cadence of the features you implement, meaning that you don't have to stick to the queue size and flow of your Plan approach. Most important thing is that you dedicate someone (or a few participants) to implementing these improvements.

## 4.4 Start working

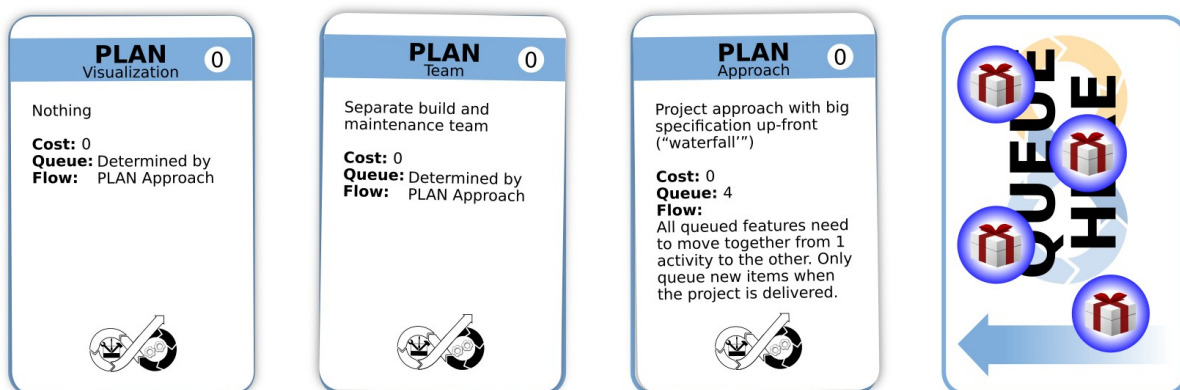
Working on a feature means moving the blue token representing that feature across the board, from the "Queue here" spot to the "Cash here" spot. Solving technical debt (orange tokens) or implementing improvements (green tokens) follow the same path. In case of an incident you move the red token from the first Code activity to the "Cash here" spot (obviously: you skip the Plan-related activities because you don't "plan" an incident). The speed at which you can move the tokens is determined by rolling the die. This indicates how many moves you can do on the board. In case of features, how many moves does not mean how many positions 1 token can be moved at once: the queue size and flow indicated on the cards determines how many tokens need to be in the same activity before moving on to the next one. This represents a batch size, the same activity applied to a number of features. Again, queue size and flow are not applicable to incidents: you want to solve these as quickly as possible.

In some cases a token can skip an activity: that means that there is no specific action needed to do this activity. This is either done automatically or there is no action at all. See further on [Fast forward](#).

When working on a feature or an incident, it does not matter for what activities (stages) you as a participant are responsible: each participant can move whatever token in whatever activity. The responsibilities only matter when improving your performance is concerned. Just go clockwise from 1 participant to the other, as in any board game.

## 4.5 Implementing features according to flow and queue size

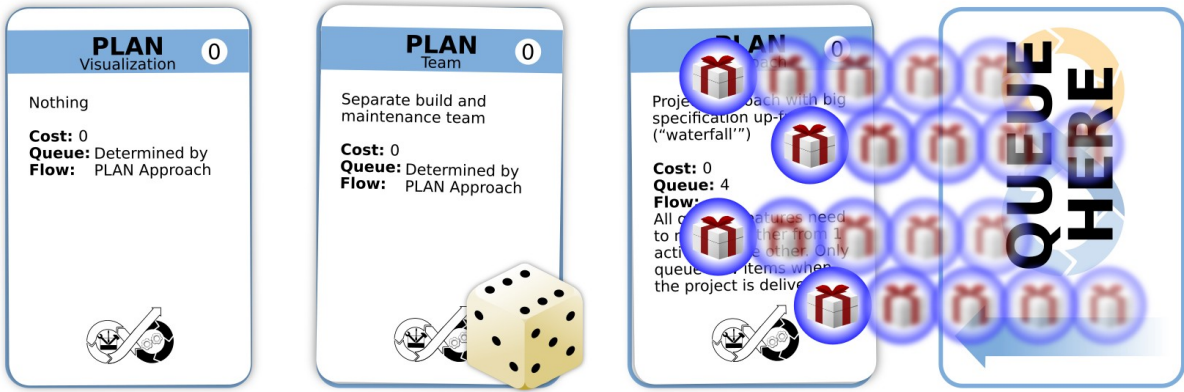
Let's take the initial situation, where all activities have performance level 0. According to the Plan approach, you queue 4 features and you need to move them all 4 from one activity to another.



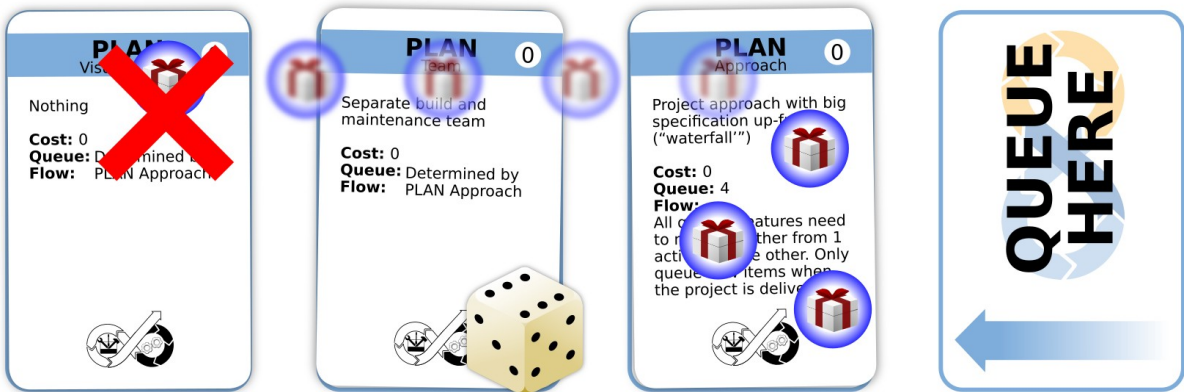
A picture says more than a thousand words, so let's explain visually with an example how to move features across the board, according to the queue and flow principles.

Suppose the first person throws a 6. Then you first move all 4 tokens to the first activity:

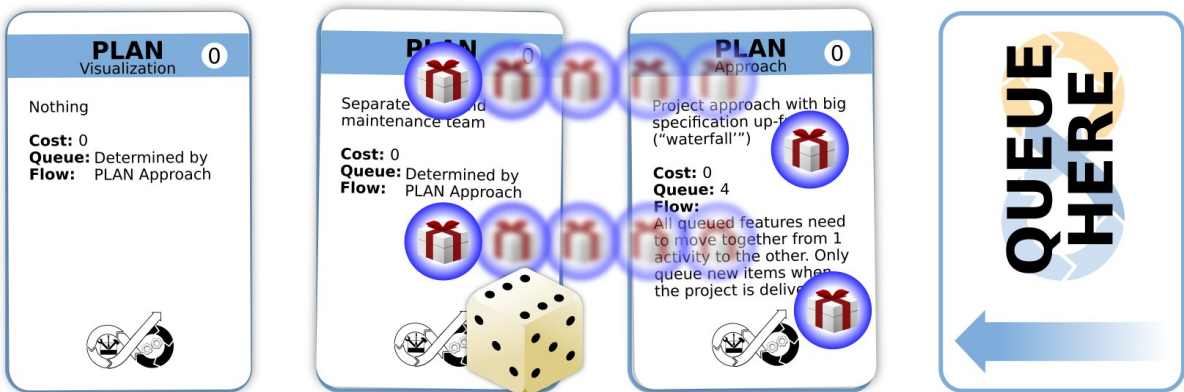




You have 2 more moves remaining. What you cannot do, according to the flow is moving 1 feature 2 steps further:

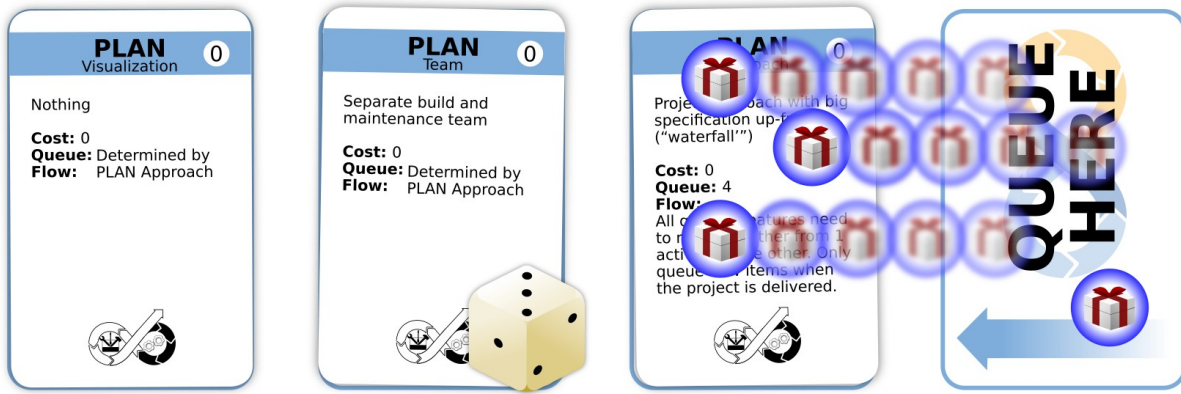


You can only do this:



In case you throw a 3 with the die, you can only move 3 tokens:



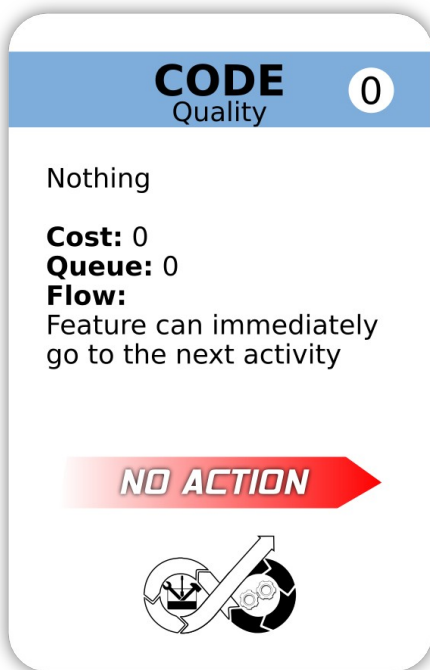


Whatever value the next person throws with the die, the first move he/she needs to do, is move the 1 token that is still on the “Queue here” position, before moving other tokens.

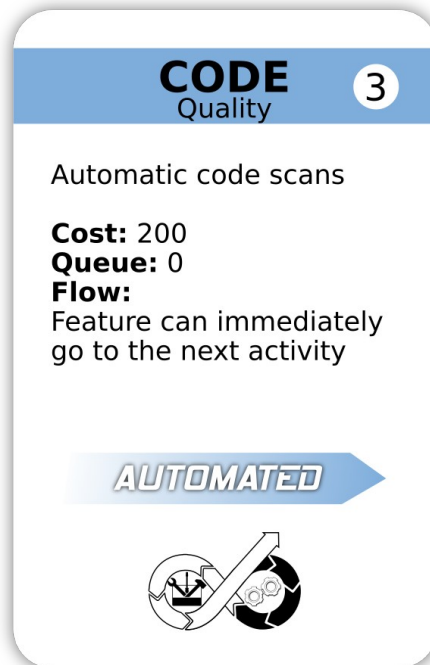
**i** The same cadence is applicable to fixing technical debt, but not for implementing improvements or fixing incidents or security vulnerabilities.

### 4.5.1 Fast forward

Some activities can immediately be skipped, you can fast forward the implementation of your feature or whatever type of work. This can either be:

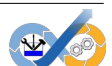


Because an activity is not yet implemented (red arrow)



Because the activity is automated (arrow in the color of the activity – blue or orange)

This will definitely speed up the implementation time, but one is not without risks...



### 4.5.2 Cutting corners

Sometimes the pressure to implement and release a new feature is so big that you need to compromise on quality (see also [Force the team to implement features faster](#)). This is possible on some activities, but with the cost of technical debt introduced in the game:



If there is only 1 symbol on the card, you will need to pick an orange token with that symbol and put it on the “Queue here” spot. If there is more than 1 symbol on the card, roll the special die until you have one of the symbols matching the ones on the card and then put that token on the “Queue here” spot.

You can plan solving technical debt along with the implementation of your features, or you can ignore it for now. Solving technical debt means that the orange token needs to be moved until it reaches the “cash here” spot. You don’t get any revenue though!

**CODE**  
Quality
1

Coding guidelines

**Cost:** 100  
**Queue:** Determined by  
**Flow:** PLAN-Approach

Skip:

### 4.5.3 Ignoring technical debt

If you choose to ignore technical debt and continue focusing on implementing new features, know that this can backfire sooner or later. Unsolved technical debt might cause an incident one day...

### 4.6 After each round



Once every participant has moved tokens on the board, someone has to roll the special die, the one with the symbols, together with the normal die. This is meant to see if a new security vulnerability is reported that needs patching or an incident has occurred that needs fixing.

If the normal die has an odd value – 1, 3 or 5 – you can ignore the symbol of the die. If you roll an even value, the symbol on the special die tells you what occurred and what you need to do:

	<a href="#">Dealing with reported security vulnerabilities</a>
Any other value	<a href="#">Dealing with incidents</a>



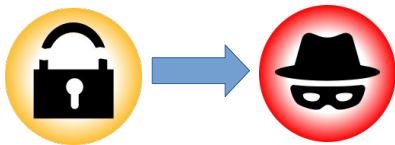
## 4.7 Dealing with reported security vulnerabilities



When a new security vulnerability is reported (also called CVE – common vulnerabilities and exposures), you take a security vulnerability token and put it on the board. You want to fix these as soon as possible, to avoid bigger problems. So you skip the Plan activities and put the token immediately on the first Code activity.

Solving a CVE means that you need to do exactly the same thing as when you implement a feature: use the regular die to move the fix through all activities of all the stages. You can ignore the queue size here. You want to solve this CVE as fast as possible, which means that you don't work on new features as long as you haven't solved the CVE. Automatic activities (like e.g. automated code scan) can be skipped immediately.

Once the fix has reached the “Cash here” space on the board, it is considered to be delivered. But you don't earn any money: this is not a feature that got delivered. Once you reach this spot, you can remove the token from the board.



**Beware! Make sure you solve this before you throw another security vulnerability with the die.** Otherwise the security vulnerability becomes a security breach. You will need to replace the orange token with a red one and treat it as an incident! This also includes financial impact.

## 4.8 Dealing with incidents

An incident occurs when one of the following values is rolled:

Normal die	
Special die	

As already mentioned, incidents can occur at the end of each round. But there are also other moments when incidents can occur:

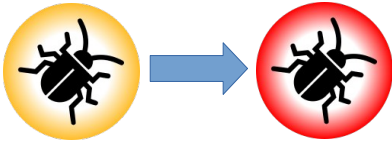
- technical debt can become an incident
- the introduction of a new feature can cause failures

In case of an incident, the normal die does not only determine whether the incident occurs or not, but also what the severity of the incident is (see [Count your losses](#)).



### 4.8.1 Technical debt becomes an incident

If you have chosen to focus on implementing new features instead of fixing technical debt, this is not without a risk. When you roll an incident with the same value as a technical debt item that is still open, your incident counts double: you need to [count the losses](#) of both the new incident and the technical debt item that has become an incident. Additionally, the technical debt item becomes an incident, meaning that you will need to swap the orange technical debt token for a red incident token with the same symbol.



This incident needs to be [fixed](#) (or the [risk accepted](#)).

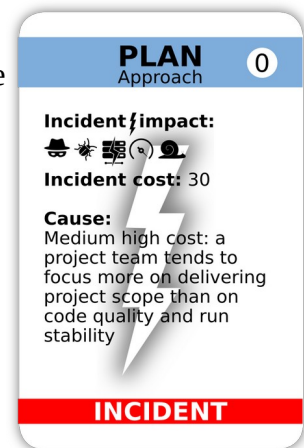
### 4.8.2 Change failure

Implementing a new feature is not without risks. Some errors can sneak in, resulting in an incident upon activation. Whenever a new (set of) feature(s) is released (i.e. whenever all blue tokens that need to be delivered together, have reached the “Cash here” spot), you need to roll the 2 dice again to see if the change is successful or not. In this case only the incident symbols count: if you roll the symbol of the security vulnerability, your change is still successful (ignore that value).

### 4.8.3 Count your losses


In case of any incident other than a reported security vulnerability you have financial impact. You have no idea what the financial impact of these incidents is, until they happen... In that case, you flip all the cards to know what your loss is.

Each activity card has an indication which incidents cause impact: any combination of security breach, bug, system outage, unexpected load or performance issue. The incident cost is the financial impact of the incident on that particular activity, **multiplied by a severity related factor**. The value of the normal die determines the severity of the incident. This is an overview of die values, severity levels and cost multiplier:



Die value	Incident severity	Cost multiplier
	Low priority	10%
	Medium priority	50%



Die value	Incident severity	Cost multiplier
	High priority	100%

Sum the incident costs of all impacted activities to know your global incident cost and multiply it by the cost multiplier corresponding the incident severity. Flip the cards back with the front side face up.

Alternatively, if you chose to divide decisions, budgets and costs per domain (stage of the DevOps cycle), you sum only the costs of your responsibility domain(s).

Either way, losing money means handing over bank notes to the bank or writing the losses on the balance sheet.

#### 4.8.4 Fixing an incident



To fix an incident you need to take the red token corresponding the incident type and put it on the board. You don't "Plan" an incident, so you can skip the Plan stage when fixing it. In other words: move the red token immediately to the first Code-activity.

To fix the incident, you need to do exactly the same thing as when dealing with a CVE: move it as fast as possible to the "Cash here" spot. You can also ignore the queue size here, skip automatic activities and you don't work on new features as long as the incident is not fixed.

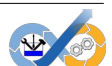
If fixing the problem is not done in a single round, you will still need to roll the die: in the meantime things can still go wrong!

#### 4.8.5 Accepting the risk

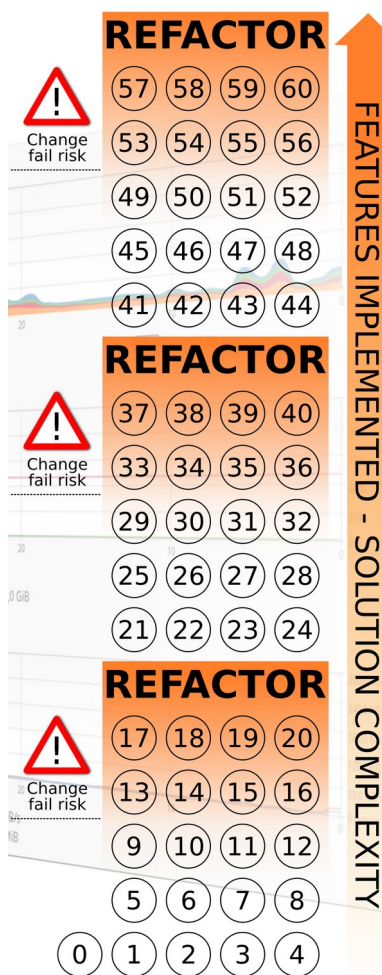
If the severity of the incident is low, you can choose to only count the losses of the incident but give priority to implementing new features over fixing the incident. In that case you will still need to put an incident token on the board, but you leave it there.

Accepting the risk means that you also accept that things can get worse. When a similar incident occurs and you haven't fixed the previous one, you will have to:

- pay twice, for the accepted incident and for the new one
- add another incident token to the board and accept or fix the incident

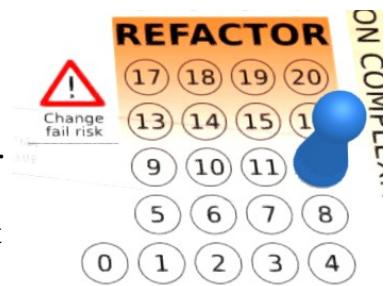


## 4.9 Finishing feature implementation work



### 4.9.1 Marking your progress

The righthand side of the board contains cells with numbers. These numbers are meant for marking the number of features you implemented. You use a pawn for this. At the start of the game the pawn is positioned at the cell with the 0. Whenever you implement few features, you move the pawn to mark the total number of implemented features.



### 4.9.2 System aging

Through the course of time – and with more features implemented – your system is aging, or even worse, degrading. This means that the risk for incidents, especially during release, increases. At a certain point – as of 13, 33 or 53 – your system has reached a level of complexity where the risk of failing changes is real. In that case, when you have implemented new features, you will need to roll the dice, to see if – and of what severity – an incident occurred during release.

### 4.9.3 Need for refactoring

At a certain point – once you reach 20, 40 or 60 – you don't want to take the risk of failing changes due to system complexity anymore. In that case you will need to refactor your system. Which refactoring is required, will be determined by rolling the incident die (if you roll the symbol of reported vulnerability, you will need to roll again). Put the corresponding orange token on the board and plan the implementation of the refactoring.



You can ignore the refactoring, but that has consequences. As long as you haven't implemented the refactoring, you will need to roll the dice for each new feature release because changes can still fail due to the complexity of your system.

## 4.10 Income and investments

As already mentioned, each time a feature is delivered, it creates revenue: you receive 100 credits for each feature that is delivered. Keep in mind though that you will only get revenue when the full scope of an iteration or project is delivered. In other words: if you e.g. still work according to a waterfall approach, your queue size is 4. You will only create revenue when all 4 items have reached the “Cash here” spot.





Delivering features and creating revenue is a good moment to invest in improving your way of working because you get extra financial means. But it is not the only moment.

It is up to the participants (or your decision strategy – see [Scenarios](#)) to decide how the improvements process is done.

#### 4.10.1 When can you invest?

There are several moments when you will typically invest in improvements:

- At the start of the game is a good moment to improve some things, to avoid big disasters and financial losses.
- In general you invest when you have new revenue.
- But sometimes a serious incident can also be the trigger to invest in improvements, if you still have the means, that is: the incident can already cause so much financial damage that you might not have the financial means to improve the biggest flaw too...

But it does not have to be limited to these moments. Whenever you feel like improving and you have the financial means, you can do so.

#### 4.10.2 Implementing improvements



[If you choose to implement improvements too](#) – not just spend the money and take the improvement for granted – you will need to bring a green token into the game. For each improvement you want to implement, you place green token on the “Queue here” spot.

If you have more than 1 activity you wish to improve, you need to remember these activities. Therefore you take the cards of the next level of the activities and place them in the middle of the board.

You can assign participants to work on these improvements instead of working on new features. These improvements don't have to follow the same cadence (flow and queue size) as indicated by the Plan approach.

Once these green tokens reach the “Cash here” spot, you can place the card(s) of the activities on their corresponding spot on the board.

### 4.11 The game ends...

... when time's up. That is the easiest bit. If you only have 1 hour to play, then the game ends after that hour.

The game can end earlier if, due to bad luck and/or wrong decisions, you go bankrupt. That is too bad. If you still have time, you can start all over again and hopefully learn from your mistakes.

Eventually, the game ends once you have reached the highest performance level at all activities. You could continue, just to earn more money but what is the use if there are no more improvements to be made?



## 4.12 How can I win this game?

Typical question you get when you use gamification, even if it is in a learning context: how can I win? Well, you know how you can lose: when you go bankrupt. So not going bankrupt brings you 1 step closer to winning.

If you have more than 1 team playing, you can compare teams:

- performance levels of all the activities; you could calculate an average per stage, overall average
- cash (even though this does not mean a lot, because one team could have invested more than another, or had more bad luck and lost more on incidents)
- value created: how many features did you deliver in total? You can keep track of the number of features you delivered, if you want to.

## 5 Possible scenarios

### 5.1 Dividing or sharing decisions, budget and costs

#### 5.1.1 Dividing

You could opt for dividing the starting budget among the participants. In a lot of companies this is probably the case. Each participant decides how much of his/her starting budget will be spent to improve their domain(s). This also means that you split the responsibilities: if anything goes wrong, the participants sum up the cost to fix the issue in their own responsibility domain. This can be a driver to improve within their responsibility domain, but on the other hand, the budget to be spent is small... And what if one domain has bad luck and has no budget left to pay the cost of an incident? Then you will need to agree with the other participants how to solve this. Maybe they will want to divide the cost, but will also want to be part of the decision making for his/her domain(s)...

When you finish a (set of) feature(s), you will need to divide the revenue among the responsibility domains and each participant can decide how much to invest and what domain to improve.

#### 5.1.2 Sharing

On the other hand, you can choose for the shared budget, shared responsibilities and shared decisions. This is more like consent decision making of Sociocracy 3.0: you continue with an action/decision, unless there is a good reason (objection) not to do so. You have a global budget and you decide together, as a management team, which domains to improve first. In case of an incident, the cost to fix that incident is paid with the global budget. And all revenue goes to the global budget and again you decide together how much of the budget is invested and which domains will be improved.



## 5.2 Start from performance level 0 or from your real performance level

By default the game starts from performance level 0. This way you experience fully what the importance is of certain investments. This is most likely the approach for sessions with a mixed audience (e.g. on meetups, conferences, etc.). This means that every single increase in performance requires investment and extra exposure to risks if you don't have the financial means yet to do that investment.

But for sessions with participants of the same organization, you can start from the actual performance level: your current plan approach, your current code quality measurements, your current test approach, your current deployment frequency and so on. This makes the game a bit more realistic and the learning experience closer to your day to day situation.

## 5.3 Force the team to implement features faster (and cut corners)

It is not uncommon that a development team is under pressure to deliver features as fast as possible. A possible reason is that the business wants to enter the market first with a certain feature, the beat the competitors. This can create extra revenue, but to do this, the team may need to compromise on quality.

As a facilitator you can say at a certain point that the new batch of features the team starts working on, will need to be implemented as fast as possible. You don't care how they do it, but it has to be done faster. And you can promise them that they will get 50% more revenue per implemented feature. This approach is especially interesting when you do a lot of manual quality checks, so that the team will need to cut corners to be able to do this. But obviously this will result in technical debt...

## 5.4 Work to implement improvements or not

The green tokens are used for implementing improvements. Using these tokens makes the game more realistic: you don't just buy e.g. a new tool, you need to implement it and adapt your processes. However, if your time is limited, you can omit the green tokens, just buy the improvement and apply it immediately.

You can also choose to start the game without the need for implementing improvements, so that the participants can get used to the game first. And after a while you can decide to introduce the need for implementation work. It is up to you as a facilitator.



## 6 Thank you

A special thank you goes to the following people for providing inspiration for this game:

- Bart Blommaerts
- Olivier Costa

I would also like to thank the people who reviewed this document and gave feedback to improve the game:

- Frederik Leijman
- Pieter Van Hees
- Robbert Valckeneers
- Sangeetha Sridhar

And off course to all the people who tried out this game and gave their feedback to improve the game, or shared their experiences:

Herman Vandezande, Bart Vanhaeren, Roel Vankriekinghe, Nele Van Beveren, Charles-Louis de Maere, Jord Rolland de Rengervé, Shibu Chacko, Jan Pannecoeck, Jean-Noël Collin, Darek Krzywania, Sven Dill, Rudy Mariën, Sébastien Barbieri, Dick Beverage, Jonas Dandois, Bieke Meeussen and all others I might have forgotten here.

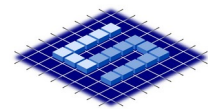
## 7 Contact information

Build-Run-Improve-Repeat is a product of SimuLearn.

More information can be found on my web site: <https://www.simu-learn.net>



Koen Vastmans: <https://www.linkedin.com/in/koenvastmans/>



SimuLearn

