The process to:

# Building successful applications

## Table of content

## Before we get started

## Before we get started

So, you have decided that you want to build an application; that is great! This guide is here to help you build apps that provide real value to your work and that other people will definitely want to use as well.

Over the past years, we, the application developers at VIKTOR, have built many apps for many different industries. We have learned a lot from that and now want to share our experiences with you. The principles and processes we explain in this guide are the same as we apply daily. They have helped us build anything from simple apps, designed to save time, to advanced engineering applications that are used to automatically calculate underground tunnels, dikes, and other large structures. Most importantly, these apps help us make people happy. And is there something better than that?

In this guide, we will provide you with tools, methods, and tips for building your own application that you can start using right now. Don't worry, you do not need to do everything all at once. First, pick the parts that make the most sense to you and start implementing those into your process. Keep in mind that it is important to try things, see how it works out, and use your new insights to keep improving!

How will we go about this? In each chapter, a different aspect of building a good application will be highlighted. First, mapping out the process, then planning and designing, after that coding, then the importance of gathering and processing feedback, and at last a recap of and some tips and trick for the best ways to create real value from your application!

While you're at it, don't forget: Building good apps is all about people. Listening to others, taking their feedback seriously, and using it to improve the application. These people may be your colleague sitting next to you, your manager, or someone outside of your company. It does not matter. Throughout this guide, we will call them 'clients', and we think you should treat them as such.

# An introduction to developing successful applications

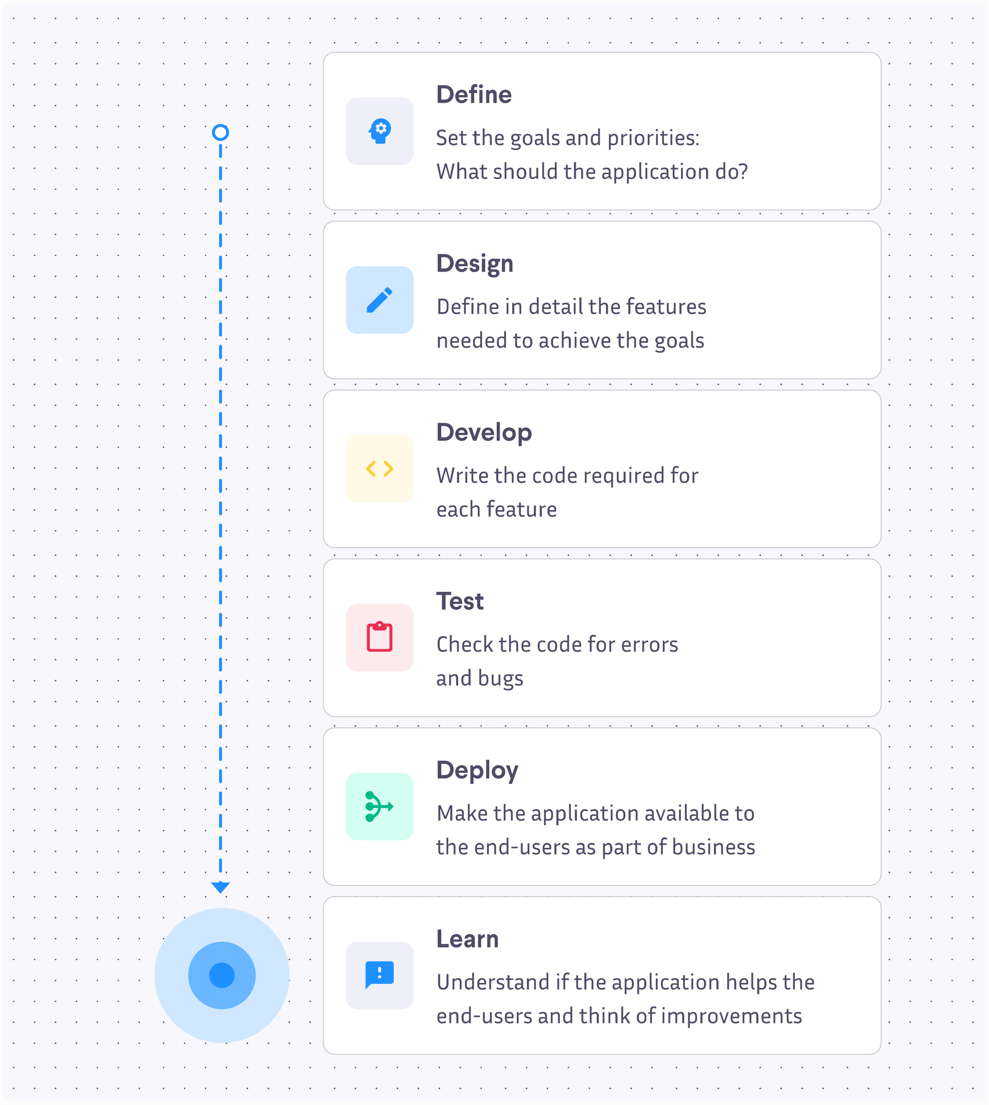
## Introduction

Before you start building an application yourself, you should know everything about the steps that you need to take to do so. In this chapter, these steps will be explained, and you will learn the best method to structure your development process for building a successful application.

## Developing software in 6 steps

When you develop a successful software tool yourself, there are six steps that you need to follow. These are: Define, design, develop, test, deploy and get feedback. Here, we will explain each step in short, and most importantly, elaborate on how you should work through the process of building awesome tools that other people will want to use as well.

First, the steps:


Define
Set the goals and priorities: What should the application do?
Design
Define in detail the features needed to achieve the goals
Develop
Write the code required for each feature
Test
Check the code for errors and bugs
Deploy
Make the application available to the end-users as part of business
Learn
Understand if the application helps the end-users and think of improvements

## Do not follow a linear process

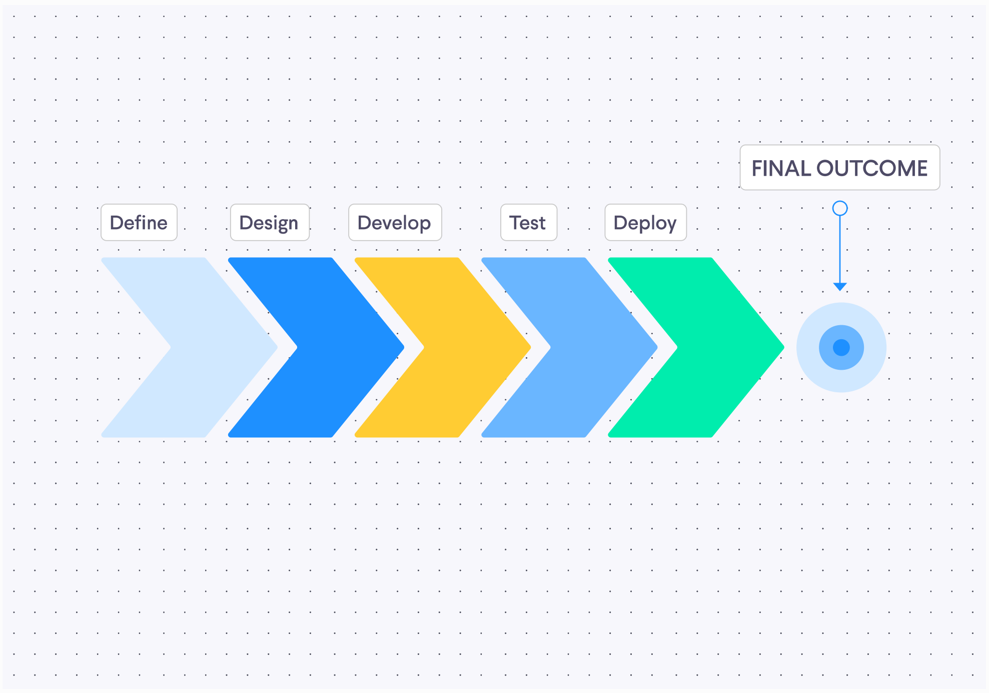
In this world, we build many things following a linear and stepwise process. This means you first define the end goal and then follow the steps that are required to get there. For example, think of a building. First, you decide what you are going to build. Then you lay the foundations, start adding the levels, place the roof, and continue until you have finished the construction. So, what we believe is that if we have a good plan, we just need to follow it to get a good result. But is this also true when building an application?

### The risks of the waterfall method

In the software world, we call this linear process 'the Waterfall Model'. Most people agree that this method does not work so well for building apps due to two reasons:

- Because most end-users (read: clients) don't know the exact requirements for the application beforehand. Only after they see the software in action can they tell what is good and what is still missing or needs to be improved.
- Because it is hard for a developer to predict all difficulties that are going to occur during the development process and if the features that they invented really solve the problem of the client.

That is why we don't believe you can make a master plan beforehand, simply follow those steps, and then end up with a successful application. It is a considerable risk; you invest all your time and energy and then have a 'grand inauguration' which could become either a great success or the biggest fiasco.

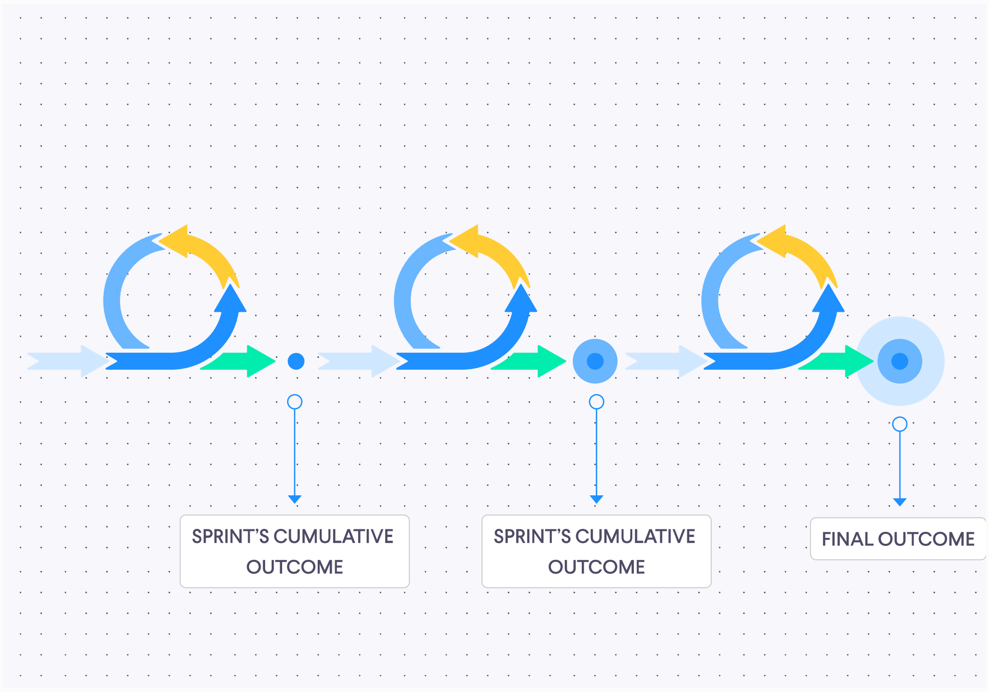


## Be Agile

Instead of a linear method, a more versatile and explorative approach has proven to yield the best results. This one is called the *Agile* method. Its core principle is to collaborate with the client as much as possible and iterate a lot during the development process. This doesn't mean we are saying that you should not have a clear goal in mind. In fact, you really should. It is just that the exact way your software will end up looking and working is something that should develop over time and in collaboration with the end-user. It cannot be defined at the beginning.

### Run sprints, not a marathon

Instead of planning all details and building the app in one sit, the Agile method proposes dividing the project into a series of short sprints. After each sprint, you will deliver the most basic app that fulfils the requirements (the minimum viable product), even if it is not fully functional, so that end-users can test it and provide feedback. With that feedback in mind, you and the client agree on the priorities and requirements for the next sprints. In that way, you will always work on what provides value for the client and end with a much better product than when you would have built everything in one shot.



## How long does a sprint last?

Each sprint usually lasts for about two weeks, followed by one week of client testing and feedback. We do not recommend taking longer than this if you are programming full-time. It is sometimes best to take baby steps in the beginning with new clients or on new projects and ask for feedback even during a sprint to ensure mutual understanding.

## Why a user-centric approach works

Approaching software development in this way ensures that you end up with a better end-product. You also put in less effort since you lower the risk of investing a lot of time in something that clients don't like. Even more important, including end-users into the development process results in tools that people will want to use eventually.

Our experience is that social factors are at least as important as technical functionalities. For example, end-users and especially domain experts can be very critical about the plans. In the beginning, negative thoughts may arise, like 'This is too complex, it is impossible to automate!', 'I have been doing it this way for 10 years, why should I change the process?' or 'Will this tool make me lose my job?'.

However, things change once the first versions of the application appear and people can actually see that their feedback is being taken seriously. The synergy between developers, domain experts  and end-users grows, and the people who once were the most critic end-users will become the biggest ambassadors, promoting the application you have built together.

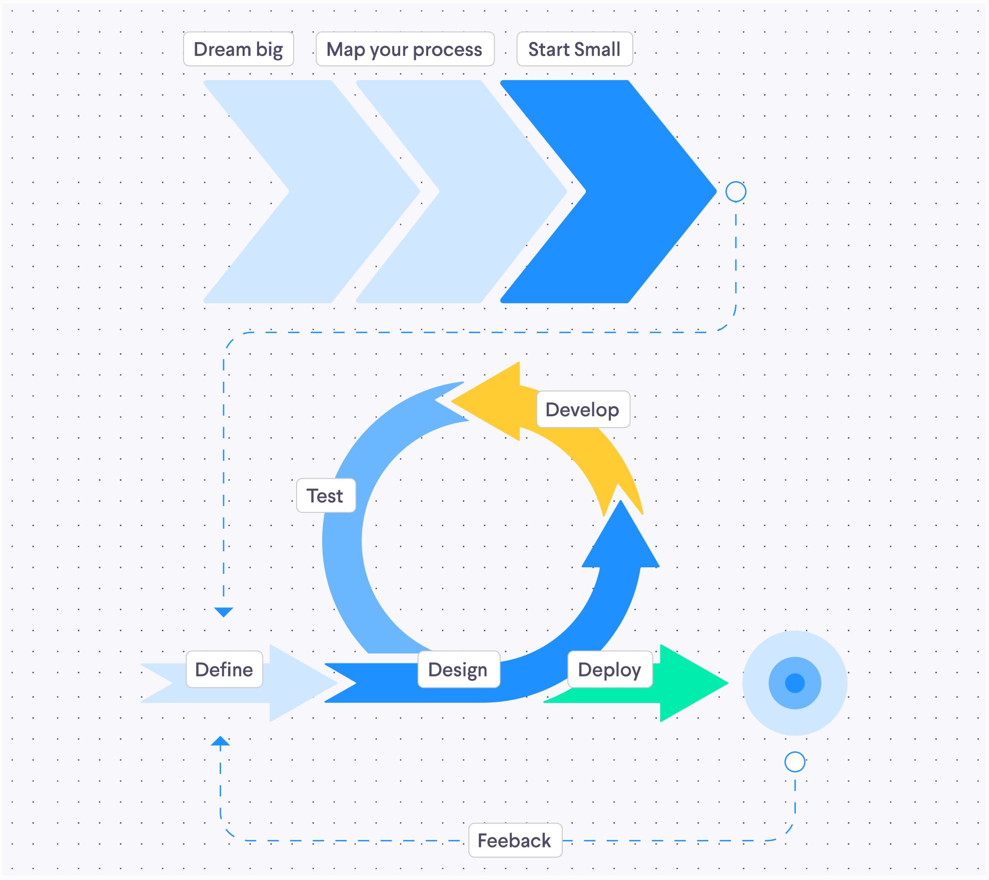
"We see our customers as invited guests to a party, and we are the hosts. It's our job every day to make every important aspect of the customer experience a little better."

**Jeff Bezos**

## Building successful applications in practice (short)

In this section, we will give you a brief overview of the process and explain each step in short.  In the next chapters we will explain the different steps in more detail and give you handy tips.

Building successful applications always start with a big dream: What do you want to achieve ultimately? Next,  you can begin designing the workflow required to get there, all of it. This is too much to start; that is why select a small portion of the workflow to start creating an app. After you have defined all the details, you can start designing and creating the application, not in one shot but in several short sprints.  After each sprint, you will get end-user feedback (clients) and define the following steps to get closer to your big dream.



## Dream big

Everything starts by defining a Big Hairy Audacious Goal (BHAG). Dare to dream, everything is possible. This BHAG will guide the development process. Don't think about all the features yet. Rather think about what you want to achieve; for example, 'I want to automate a tunnel's design fully'.

## Map out your process

To improve your process, you need to understand it thoroughly first. Map out your process and involve everyone you need to be able to understand all the details. Create a workflow diagram and see how the process can be improved to get closer to your BHAG. Organise a brainstorm, think about your ideal process and the benefit of the changes you want to make. Don't overthink possible technical challenges yet.

## Start small

Select a small portion of the workflow diagram and start with that. Which portion this will be depends on two things: Maximum impact and feasibility. In other words, where can you make a significant improvement with little effort? That is always a good starting point.

## Make a plan

This is where you start using the Agile method. Formulate short term objectives in the form of user stories and write them down. For this you can use a product backlog. User stories are statements written from the end-user's perspective. The idea here is to define what the user wants to achieve. Not how they want to achieve it. That part you can leave to the developers.

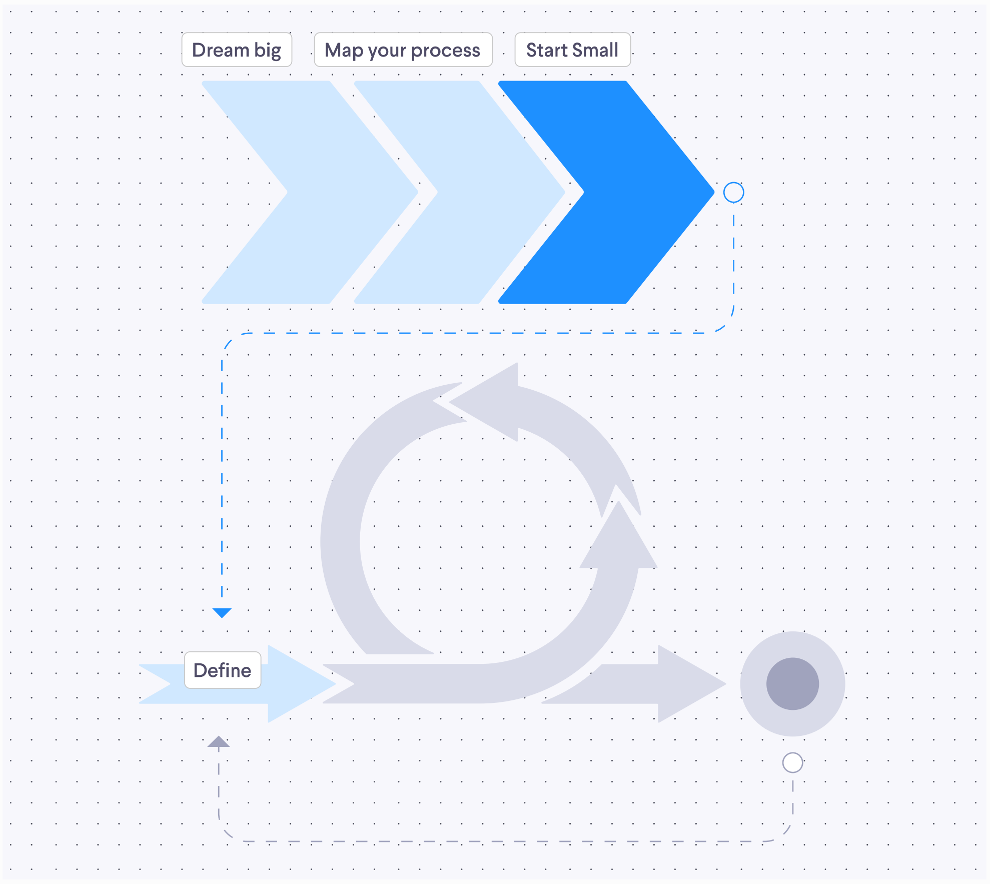
## First sprint, feedback, repeat

For your first development sprint, you should create the simplest app that fulfils the requirements: A minimum viable product. Let the end-user interact with the product and ask them for feedback. Store this feedback on a feedback board and decide the goal for the next sprint(s) together with the end-users. Repeat this until you reach your BHAG.

## Planning and designing your application

# Planning and designing your application

In this chapter, we will explain the importance of and best way to go about the first steps: defining your dream, mapping the process, selecting somewhere to start, and designing and planning the app. But before all of this, we will talk about gathering the right team. Remember, creating successful applications is all about people.



## Gather the right team

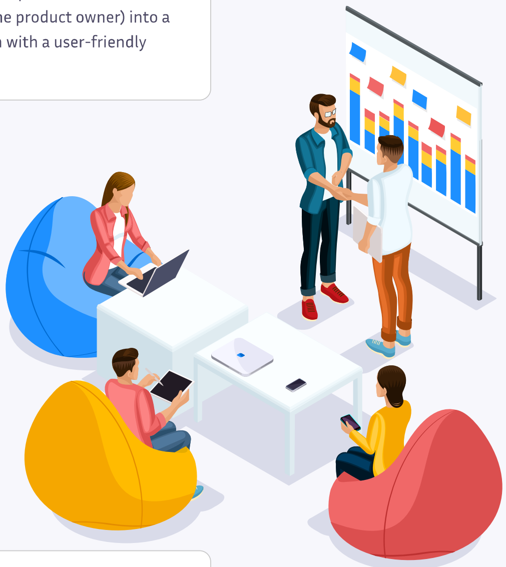
Building an application isn't done by software engineers only. In the ideal situation, it should be a collaboration between the software developers who write the code, the domain experts who have the specialised knowledge (e.g. geotechnical engineering), the end-users who are going to use the application once it is finished, and the product owner who is the bridge between all of them.

**Product owners**

Are the bridge between end-users and developers. They understand the wishes of the end-user and translate them into features that developers can make. Ideally, they have both the domain and software development knowledge.

**Developers**

Have knowledge about software development and coding. They oversee these aspects and turn the wishes and requirements of end-users (provided by the product owner) into a workable application with a user-friendly interface.



**Domain experts**

Have specialized knowledge of the domain that will be applied during the development. For example, geotechnical engineering.

**End User**

Are going to use the application eventually. After each sprint, they test the application and provide feedback about the usage and their requirements to the product owner.

Sometimes the domain expert is also the one building the tools, and that is fine. However, even though it is possible for one person to take on multiple roles, this does not mean that a role can simply be left out or that you won't need to have separate end-users to test your tool. Especially the latter is the first and biggest mistake you can make: Assuming that, because **you** are able to use a tool and **you** find it handy, **other people** will think this as well. So, it is important to involve a representative group of end-users from the very beginning of the development on.

## Dream big, start small

An application can be built for all kinds of processes, such as automating a repetitive task and making people's work more fun. Maybe you have a big idea and want to build a large and complex application. That is nice; we like that! Having such a point on the horizon is great and works very motivating. However, creating such an extensive application can also become very complex and frustrating. Well, we are here to tell you: Don't let this stop you; we have a solution to make it work!

## Think outside of the box

Building an awesome application requires creativity. Therefore, we recommend people to open your mind, get out of there comfort zone and don't be scared to give some new things a try. Think about the big dream you want to achieve. It will drive the development and be the motivation to keep forward. Dream in something like 'I want to automate the design of a tunnel completely', and when thinking about it, don't focus on why it is not possible.

Here we have a short story to make the point. Jaap Wierenga is Lead Engineer Hydraulic Engineering at Heijmans, a large Dutch construction company. He has more than five years of experience making pretty impressive automation tools. One of those is WILMA, a parametric design tool that automates the design of dikes. The funny part? In the beginning, Jaap Wiernega was the first to say: "This is not possible because of many, many, many, many technical reasons!".

"I think it is very important to have a goal in mind, something to work towards, even if it seems very complex or impossible to achieve. Start with the easiest steps towards that goal. By starting this process, you will find your way, but keep focus on the big goals you want to achieve. Applications like this need to start small and grow organically. With each step you will learn more about what is possible and what you need. You will also discover new possibilities and start getting more colleagues enthusiastic. It has a snowball effect, by parametrizing part of the design, and then keep adding small bits and pieces, you get closer sooner than you might have expected."

**Jaap Wierenga**

Lead Engineer Hydraulic Engineering at Heijmans

### Break it down

Developing a complex application doesn't necessarily mean you build a huge application that will take years to develop. It is very well possible to build a complex application without actually making it too complex for yourself and others. How? You just need to break it down into smaller pieces. For example, start by building something small that can solve only part of the problem, just like with the Agile method, remember?

Dreaming of a big and complex application is good, but starting big can also tie the knot around your application's neck really quickly. Many projects have already failed due to this approach. The problem? People are asking too much of themselves and their end-users. Going from 0 to 100 at once is a lot to take in and makes the development process unmanageable.

### Keep thinking ahead

Thus, start small. But while you do this, you also need to think ahead. How else are you going to get there? This can be along the lines of 'How can I make my application grow?'. To answer this question, let's look at an example:

Imagine you are building an application to calculate the structures of a bridge automatically. This is fairly complex, but you have to start building somewhere. This, for example, means that at first, you can only calculate certain parts of the structure. This may not seem much, but if you keep working on that, your application could be able to design an entire bridge at just the push of a button with time. Believe us; **a lot** is possible if you set your mind to it.

## Understanding your process is key

Now that you have a big dream, it's time to flesh out how you will get there. The key to improve and automate a workflow is to **understand it fully**. Start creating a workflow diagram of the process that you want to turn into an application. For your workflow diagram, you will map out the whole process from start to finish with all details included. Remember, while a software developer can be super powerful in making calculations, they are terrible with improvisation.

"When you start looking at a problem and it seems really simple, you don't really understand the complexity of the problem. Then you get into the problem, and you see that it's really complicated, and you come up with all these convoluted solutions. That's sort of the middle, and that's where most people stop... But the really great person will keep on going and find the key, the underlying principle of the problem — and come up with an elegant, really beautiful solution that works.

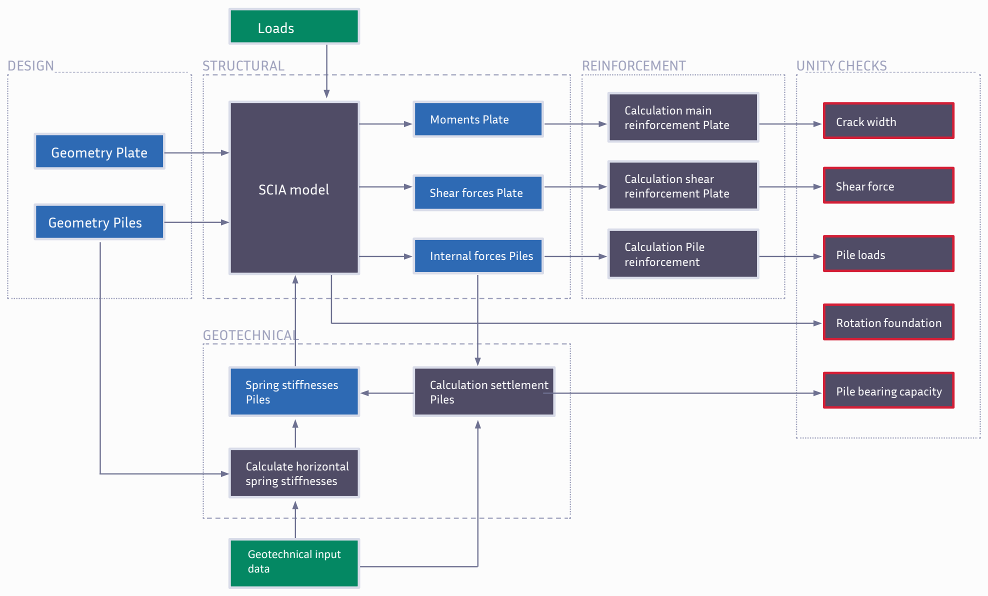
**Steve Jobs**

## The value of a workflow diagram

Here are five reasons why it is worth investing time in a good workflow.

- **Overview and clarity:** The diagram is important for everyone. Not only for the domain expert(s) to keep an overview of the workflow but also for the developer who will turn it into an application. All the steps in the workflow must be incorporated, even if everyone is assumed to know them. It is funny to notice that people often overestimate how well they know the working process of a colleague, while in reality, this is not true at all. It is good that everyone is on the same page.
- **New insights:** Humans are creatures of habit. A field specialist may find it logical to do things as he has always done it. A programmer, however, may look at it from a very different angle. Working together on the workflow and discuss all detail is an excellent opportunity to improve the process.
- **Anticipate critical aspects:** Creating a workflow diagram helps understand the details and identify possible critical elements of the development. This can avoid a big headache later. Some important discussions are, for example, which software packages are used? Does this software have an API? How does information flow? In which format the data is share, and is this always the case?
- **A common language:** By including the terminology of each part into the workflow diagram, you clarify how vocabulary should be used. This is important to avoid misunderstandings when multiple domain experts and developers are involved in the process.

- **Evaluate edge cases:** Most time, a workflow diagram is made for the 'standard' situations that occur in about 95% of the cases. This is fine; you don't want to overcomplicate things at the beginning. Once you are confident about having the proper workflow, try to evaluate some special situations. Maybe you don't want to include these cases at the beginning of the development but making everyone aware of them can make the difference between incorporating them in the future or not.



Example of a workflow diagram

## Planning the first step

Once you have the final workflow diagram, you can figure out which (if not all) parts of the workflow can be made into an application. If the process is already contained, the whole thing can be an application. If the process is very large, start small and make it big.

To select the first step, think about two things: Maximum impact and feasibility. In other words, where can you make a significant improvement or save a lot time with little effort? That is always a good point to start.

## Define requirements with user stories

A user story is an informal, general explanation of a software feature, written from the perspective of the end-user; something that the user should be able to do in the application once it is finished. This helps the developer to understand what the end-user is trying to accomplish when using the application and serves as guideline when creating the different features. This form is chosen because the goal of the application is ultimately presenting value to the end-user.

User stories are usually built in the from 'As a user, I want to...'.

Examples of user stories are:

- As a user, I want to create and manage projects inside the app
- As a user, I want to design different bridges within each project
- As a user, I want to visualise the bridge as a 3D model
- As a user, I want to compare the results of different bridge designs

If you are creating a complex app, the user stories that are given as example here could be too broad for a developer. In that case, it is recommended to create topics and think of more specific user stories per topic. For example, we will take the first user story from the previous list, assign it a topic, and make it more specific:

Topic: Manage and create projects inside the app

- As a user, I want to create projects
- As a user, I want to add properties to a project such as: The clients' name, a project code, and a date
- As a user, I want ...

*Tip:*

*Keep in mind that sometimes it is more important to communicate what you want to achieve and why you want to achieve it than how you are going to do it. Making super-specific requirements can have a negative effect on the development speed and quality of the product. Be clear about what you want to achieve but give the developer the freedom to find the best solution from their own perspective.*

## Make a time planning

After you have gathered your team and decided on which processes you are going to incorporate and how, you should also make a planning on when you are doing all of these things. To ensure success, it is important to think about which steps should be taken at every moment of developing your application and make time estimations based on that **before** you start programming.

Even though the end-goal is not specified completely within the Agile method, a time estimation should be made on when you will reach a finished product, nevertheless. This can be done based on the workflow diagram and user stories. For each of the user stories, a separate time estimation can be made thinking it is an isolated code block designed for a specific function. This is a good method because a time estimation is easier to make when it is for isolated functions that can be implemented next to each other instead of for the whole application at once.

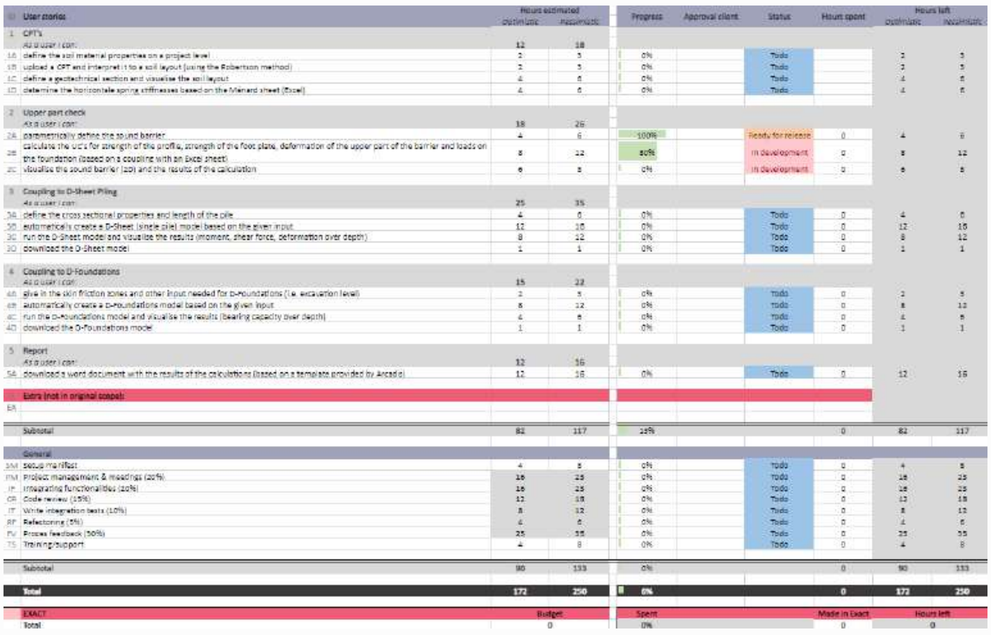
Having a time estimation based on user stories also has other benefits: You can be very transparent about the progress to all parties involved and have a clear way to compare expectations and reality (what was easier than you expected? And what was harder and took more time and why?). This transparency helps generate mutual understanding and trust, which is beneficial when discussing any deviation from the original planning.

*Tip:*

*When making a time planning, also take other aspects into account like communication with the end-users, project management, reviewing each other's code, writing test, etc.,*

## Create a product backlog

It is recommended to use a product backlog to keep track of time, budget, and progress. In the product backlog, you will add all user stories organised by topic, a time estimation for each user story, and it's progress status. Examples of status could be: Pending, in development, under review, ready for release, and approved by client. In the picture you can see an example of a product backlog. You can download the product backlog template from the VIKTOR website



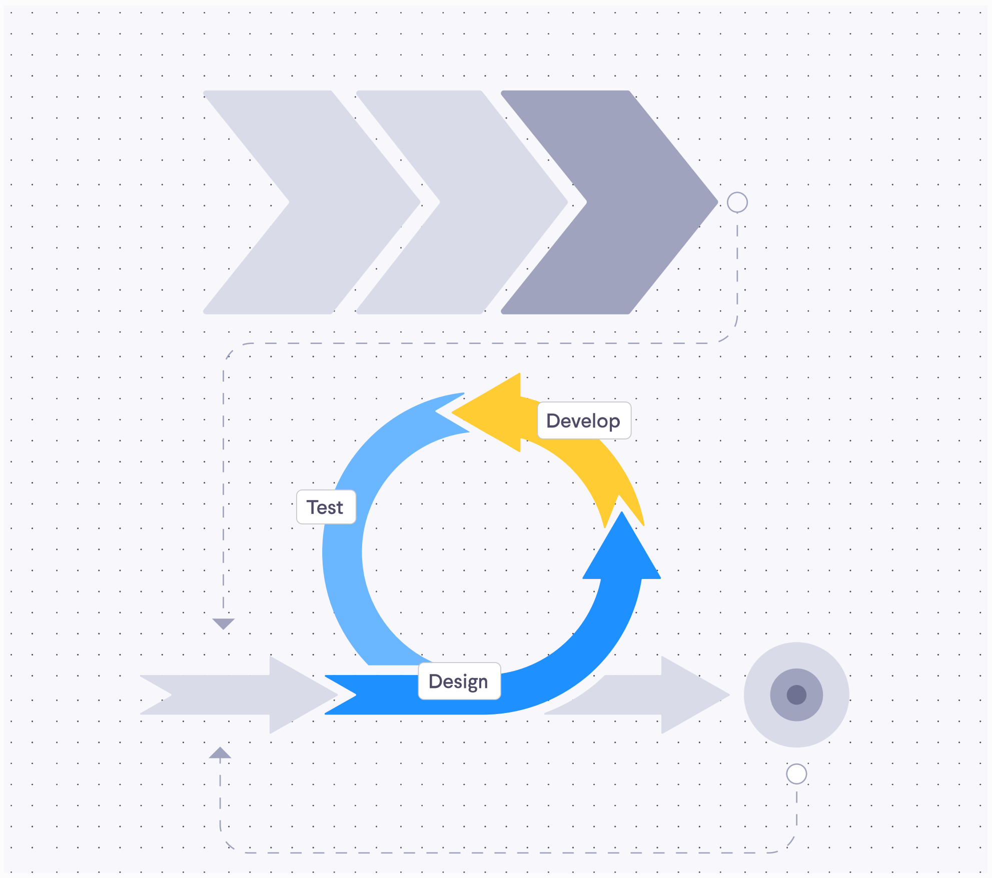
| ID | User stories | Hours estimated | | Progress | Approval client | Status | Hours spent | Hours left | |
|---|---|---|---|---|---|---|---|---|---|
| | | optimistic | pessimistic | | | | | optimistic | pessimistic |
| 1 | CPT's | | | | | | | | |
| | As a user I can: | 12 | 18 | | | | | | |
| 1A | define the soil material properties on a project level | 2 | 3 | 0% | | Todo | | 2 | 3 |
| 1B | upload a CPT and interpret it to a soil layout (using the Robertson method) | 2 | 3 | 0% | | Todo | | 2 | 3 |
| 1C | define a geotechnical section and visualise the soil layout | 4 | 6 | 0% | | Todo | | 4 | 6 |
| 1D | determine the horizontale spring stiffnesses based on the Ménard sheet (Excel) | 4 | 6 | 0% | | Todo | | 4 | 6 |
| 2 | Upper part check | | | | | | | | |
| | As a user I can: | 18 | 26 | | | | | | |
| 2A | parametrically define the sound barrier | 4 | 6 | 100% | | Ready for release | 0 | 4 | 6 |
| 2B | calculate the uc's for strength of the profile, strength of the foot plate, deformation of the upper part of the barrier and loads on the foundation (based on a coupling with an Excel sheet) | 8 | 12 | 80% | | In development | 0 | 8 | 12 |
| 2C | visualise the sound barrier (2D) and the results of the calculation | 6 | 8 | 0% | | In development | 0 | 6 | 8 |
| 3 | Coupling to D-Sheet Piling | | | | | | | | |
| | As a user I can: | 25 | 35 | | | | | | |
| 3A | define the cross sectional properties and length of the pile | 4 | 6 | 0% | | Todo | 0 | 4 | 6 |
| 3B | automatically create a D-Sheet (single pile) model based on the given input | 12 | 16 | 0% | | Todo | 0 | 12 | 16 |
| 3C | run the D-Sheet model and visualise the results (moment, shear force, deformation over depth) | 8 | 12 | 0% | | Todo | 0 | 8 | 12 |
| 3D | download the D-Sheet model | 1 | 1 | 0% | | Todo | 0 | 1 | 1 |
| 4 | Coupling to D-Foundations | | | | | | | | |
| | As a user I can: | 15 | 22 | | | | | | |
| 4A | give in the skin friction zones and other input needed for D-Foundations (i.e. excavation level) | 2 | 3 | 0% | | Todo | 0 | 2 | 3 |
| 4B | automatically create a D-Foundations model based on the given input | 8 | 12 | 0% | | Todo | 0 | 8 | 12 |
| 4C | run the D-Foundations model and visualise the results (bearing capacity over depth) | 4 | 6 | 0% | | Todo | 0 | 4 | 6 |
| 4D | download the D-Foundations model | 1 | 1 | 0% | | Todo | 0 | 1 | 1 |
| 5 | Report | | | | | | | | |
| | As a user I can: | 12 | 16 | | | | | | |
| 5A | download a word document with the results of the calculations (based on a template provided by Arcadis) | 12 | 16 | 0% | | Todo | 0 | 12 | 16 |
| | Extra (not in original scope): | | | | | | | | |
| EA | | | | | | | | | |
| | Subtotal | 82 | 117 | 19% | | | 0 | 82 | 117 |
| | General | | | | | | | | |
| SM | Setup manifest | 4 | 5 | 0% | | Todo | 0 | 4 | 5 |
| PM | Project management & meetings (20%) | 16 | 23 | 0% | | Todo | 0 | 16 | 23 |
| IF | Integrating functionalities (20%) | 16 | 23 | 0% | | Todo | 0 | 16 | 23 |
| CR | Code review (15%) | 12 | 18 | 0% | | Todo | 0 | 12 | 18 |
| IT | Write integration tests (10%) | 8 | 12 | 0% | | Todo | 0 | 8 | 12 |
| RF | Refactoring (5%) | 4 | 6 | 0% | | Todo | 0 | 4 | 6 |
| FV | Proces feedback (30%) | 25 | 35 | 0% | | Todo | 0 | 25 | 35 |
| TS | Training/support | 4 | 8 | 0% | | Todo | 0 | 4 | 8 |
| | Subtotal | 90 | 133 | 0% | | | 0 | 90 | 133 |
| | Total | 172 | 250 | 0% | | | 0 | 172 | 250 |
| | EXACT | Budget | | Spent | | | Made in Exact | Hours left | |
| | Total | 0 | | 0% | | | 0 | 0 | |

Example of a product backlog

## How to start coding

## How to start coding

With a clear idea for an application, all your processes mapped out, user stories and requirements defined, and a time planning prepared, coding can finally start! In this chapter, we will not go into the specifics on **how** to code your application but rather on where a good place to start is and how you can best manage your code throughout the development process, so you end up with a truly successful application.



## A mock-up is a good place to start

Just as with the entire development process of your application, with coding it is also important to dream big but start small. For example, by starting with a mock-up user interface. You can build this interface within a day to get a clear picture of how the elements (parameters, input, and output) interact with each other. You can add some dummy calculations, models, graphics, and reports to complete the picture. Together with the end-users and product owner, you can look at the mock-up and decide on which parts are good and which things should be adjusted or changed. This way, you can quickly get a broad view of what your end-product will look like, without putting lots of effort into completely working out all kinds of different options.

## Start programming the logic

Only after you have built a successful mock-up, the coding of the logic can start. Don’t worry, not all at once. Start with something small and expand from thereon, according to the product backlog you made.

We recommend trying that you try to connect all the islands of information first. Try creating a fully functional but very simplistic code that goes through all parts of the workflow. Just take a very simple calculation, it does not matter if you hard code some things or it only works for one case. The most important thing that you want to prove here is that you are able to automate the workflow as planned. The worst thing that can happen is that you fully develop one part of the workflow and later discover that you are not able to connect that part to the next one.

Once you have verified that the complete system is working, you can start improving each part and make it more complex.

“First, solve the problem. Then, write the code.”

**Stijn Jansen**

Head Application Development at VIKTOR

## Working with other developers

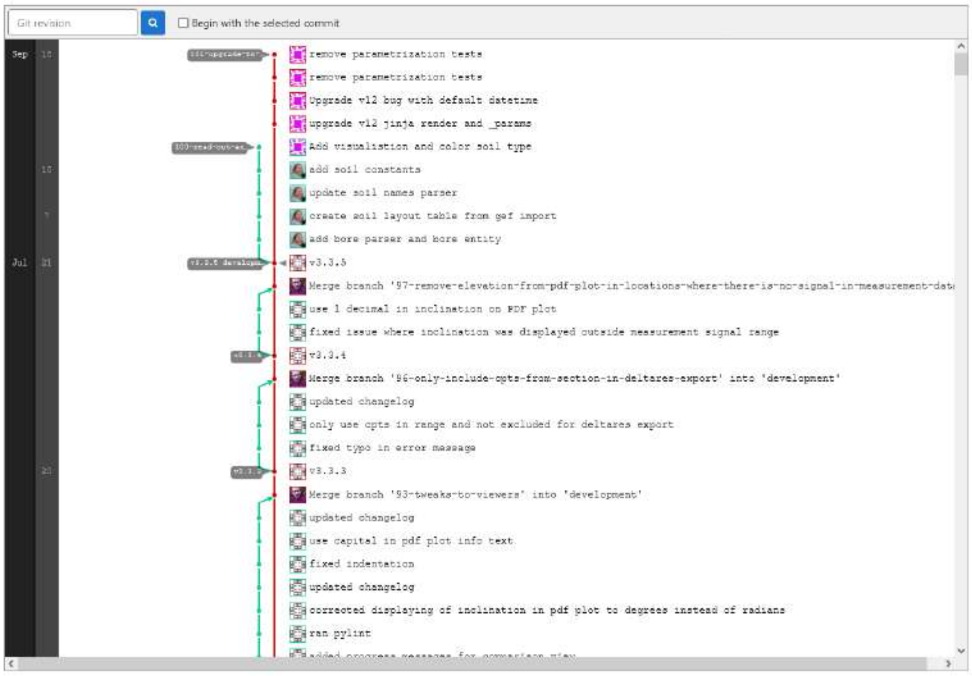
### Find a system to work with

On bigger projects, there are often multiple developers working on writing the code. To keep a clear overview, it is good to have a structured way of working and are somehow able to keep track of everyone's progress and adjustments. This is also vital to keep all parts of the code compatible with each other.

We recommend using a code sharing platform with a version control system, such as GitHub or GitLab. These platforms have several benefits, such as:

**Version control:** Everyone that is working on a code will always work with the right of it. No version 1, version 2, version 3, flying around anymore. If someone changes a part of the code, the system immediately tracks this change, and everyone is immediately able to see it as well.

**Improved collaboration:** Code sharing platforms allow people to work on one big project, but still have their own separate 'branch' (part of the code) within. You are able to change your own code and later merge the branch into another branch and make sure everything stay compatible. You can also see and check the code that others are working on.



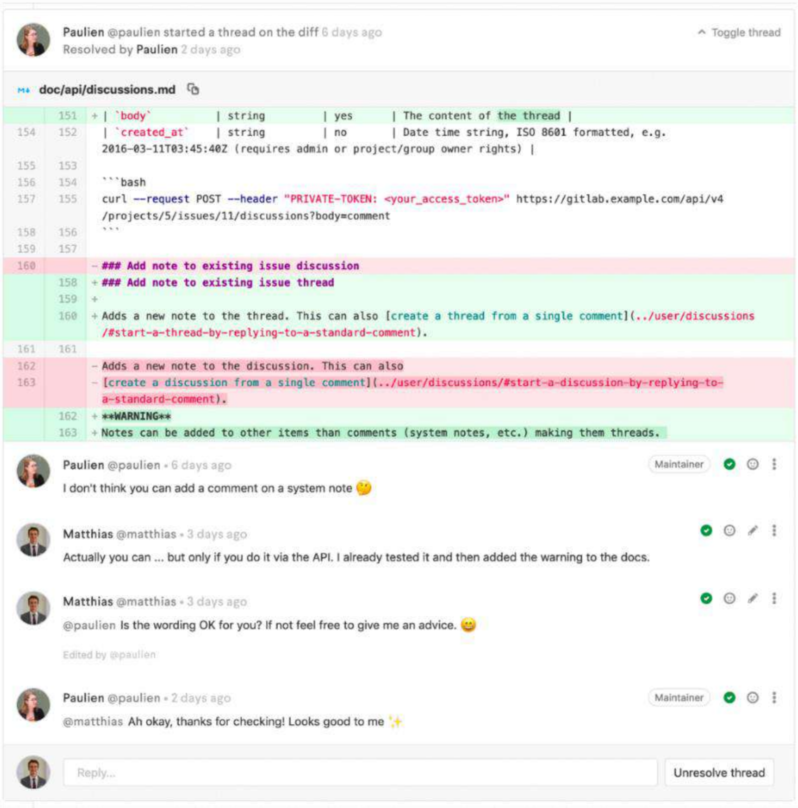
```
Sep 18  remove parametrization tests
        remove parametrization tests
        Upgrade v12 bug with default datetime
        upgrade v12 jinja render and _params
        Add visualistion and color soil type
    10  add soil constants
        update soil names parser
     7  create soil layout table from gef import
        add bore parser and bore entity
Jul 21  v3.3.5
        Merge branch '97-remove-elevation-from-pdf-plot-in-locations-where-there-is-no-signal-in-measurement-dat
        use 1 decimal in inclination on PDF plot
        fixed issue where inclination was displayed outside measurement signal range
        v3.3.4
        Merge branch '96-only-include-cpts-from-section-in-deltares-export' into 'development'
        updated changelog
        only use cpts in range and not excluded for deltares export
        fixed typo in error message
    20  v3.3.3
        Merge branch '93-tweaks-to-viewers' into 'development'
        updated changelog
        use capital in pdf plot info text
        fixed indentation
        updated changelog
        corrected displaying of inclination in pdf plot to degrees instead of radians
        ran pylint
        added progress messages for comparison view
```

Development branches visualisation

**Issue management:** To remain in control over the different parts of the code that are worked on, different issues can be created for specific features. Each of these issues can be linked to one of the user stories. On a code sharing platform, you can link deadlines to issues as well. This way, the planning and controlling the project is done easily and all in one place.

**Sustainability through sharing:** Because it is possible to check each other's code when you're using a code sharing platform, developers are more likely to write code that other people will be able to understand. This benefits not only other people but developers themselves as well. When others have difficulties understanding your code, you will have difficulties understanding your own code too in a couple of weeks. Due to the sharing-aspect, working on a platform basically forces developers to write futureproof code that enables them and others to work with on the long term. So, even after you may have taken a break from it for a while.

**Sustainability through commenting:** Another functionality of the platform is that it allows people to add comments in the code. This way, you can ask specific questions, suggest changes with a clear focus, or just look at it from different perspectives. The ability to check on each other's code ensures for a higher quality code. Bugs are detected and solved quickly. You can also help each other with difficult parts and have discussions on about how things can be done differently or more efficient. When you have to explain your code to a colleague, it makes it clearer for both parties. This way it also becomes more concise with rest of the code.


Paulien @paulien started a thread on the diff 6 days ago
Resolved by Paulien 2 days ago

doc/api/discussions.md

~~~
+ | `body`          | string        | yes     | The content of the thread |
  | `created_at`    | string        | no      | Date time string, ISO 8601 formatted, e.g.
  2016-03-11T03:45:40Z (requires admin or project/group owner rights) |

  ```bash
  curl --request POST --header "PRIVATE-TOKEN: <your_access_token>" https://gitlab.example.com/api/v4
  /projects/5/issues/11/discussions?body=comment
  ```

- ### Add note to existing issue discussion
+ ### Add note to existing issue thread
+
+ Adds a new note to the thread. This can also [create a thread from a single comment](../user/discussions
  /#start-a-thread-by-replying-to-a-standard-comment).

- Adds a new note to the discussion. This can also
- [create a discussion from a single comment](../user/discussions/#start-a-discussion-by-replying-to-
  a-standard-comment).
+ **WARNING**
+ Notes can be added to other items than comments (system notes, etc.) making them threads.
~~~

Paulien @paulien • 6 days ago
I don't think you can add a comment on a system note 🤔

Matthias @matthias • 3 days ago
Actually you can ... but only if you do it via the API. I already tested it and then added the warning to the docs.

Matthias @matthias • 3 days ago
@paulien Is the wording OK for you? If not feel free to give me an advice. 😀

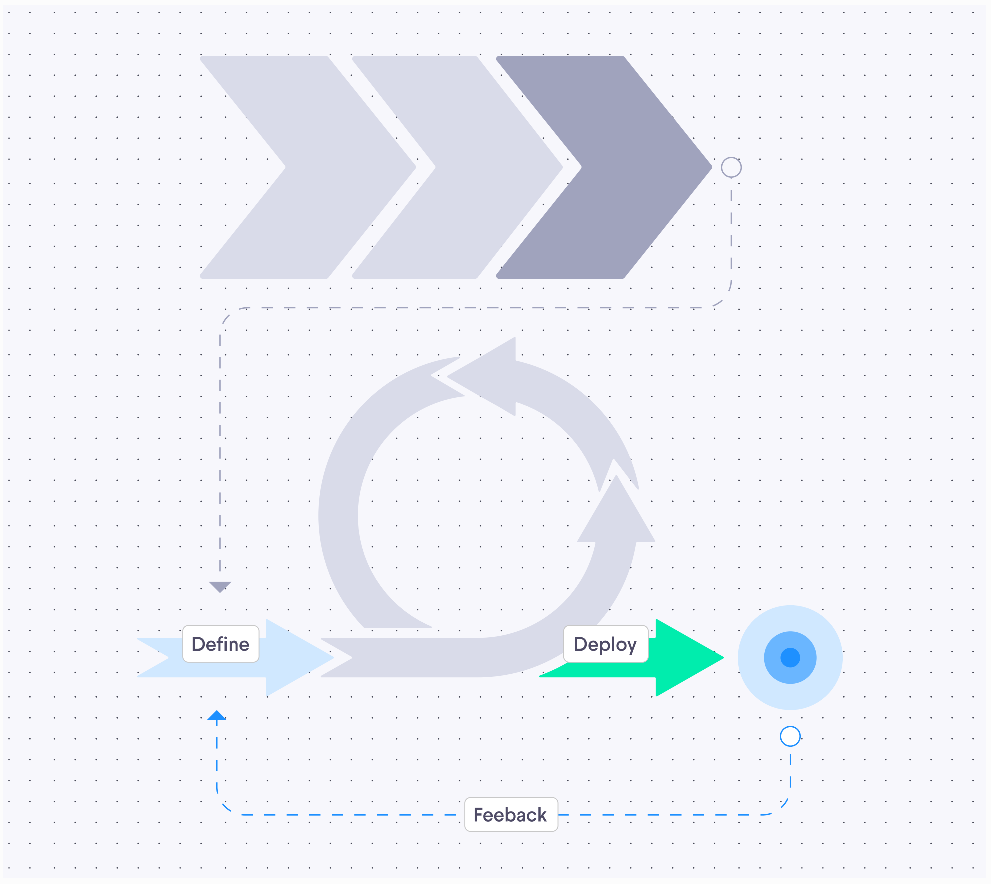
Edited by @paulien

Paulien @paulien • 2 days ago
@matthias Ah okay, thanks for checking! Looks good to me ✨

## Gathering and processing feedback

## Gathering and processing feedback

It can be difficult to determine if the underdeveloped tool is generating real value for the end users and which of the many requested features should be made first. Of course, you already made your planning and have outlined what you are going to do. Still, it can be difficult to choose which user story requires the most attention and if the feature under development really solves the problem. The solution: actively search for feedback of the end-users and process it carefully. In this chapter, it will be explained how you can do this best.



## End-users should help testing

People underestimate the importance of end-users testing the app and the time that this requires. After all, what could go wrong, right? Well, developers are humans too! So even if they take their best effort to deliver a bugs-free app, we guarantee there is still a way to make it malfunction. This is especially the case when the developer does not have any domain knowledge, which make it extra hard for them to check the results and think of all kind of unusual but still realistic cases. An experienced domain expert, on the other side, can come up with all kinds of special cases and is able to identify in a split second that a results is a bit off.

“It’s hard enough to find an error in your code when you’re looking for it; its even harder when you’ve ASSUMED your code is ERROR-FREE."

**Steve McConnell**

Author of bestselling software engineering textbooks

## Plan testing in advance

Everyone has a busy job. Experience has shown that not making a time-planning and setting deadlines to test the application, leads to people not testing it at all in the end. Testing the app and giving feedback will be low on people’s to-do lists and only happen once they have a good reason to do it. This often reads: In case something really goes wrong. Trust us, you really don’t want to be in that situation, especially close once you get closer to the deadline.

## Four reasons to get feedback

Getting feedback on a regular basis really helps improving the application. With rounds of feedback every two weeks, bugs, errors, and other mistakes and/or misunderstandings can be solved quickly.

Throughout the process, you want to test your application so you can receive feedback on both objective aspects, such as whether the correct answer is presented or not (for example with calculations), and subjective aspects, such as the layout and usability.

Benefits of testing are:

## Reason 1: Issue prevention

If feedback is given in time, problems later on can be prevented and anticipated. You don't want to build on loose soil. Otherwise, things can get really ugly. It is crucial to schedule not only the coding sprints, but also the tests and following feedback moments with everyone involved. Such feedback moments are not just 5 minutes of talking but rather take up several hours of testing and reviewing the application thoroughly to add the most the value.

## Reason 2: User-friendliness

Most times it is clear (but maybe laborious) how you get the right numerical solutions to a problem. However, designing the interface for a good user experience can be a much more complex and abstract task. The layout and usability of an application are subjective aspects and therefore differ from user to user. To get the most optimal results, it is important to gather feedback from as many end-users as you can get. All of this information should then be combined to build the final concept version of what your application is going to look like, tending to the most common and important needs of every user.

## Reason 3: Gaining insights

Not only end-users but also developers benefit from this feedback. In sessions with domain experts, developers often gain the best insights into the technical processes that are happening and then come up with creative ways to solve problems!

## Reason 4: Making things clear

Additionally, it is possible that you find out a user-story appears to be too broad because it is not formulated clearly enough for developers to understand well. During these feedback moments, problems like this can be discussed and anticipated. Now you know your user-story needs to be described in more detail, so that the developers have a clear image of where they are headed and what concrete steps they need to take to get there.

If you start building without having feedback rounds in-between, an assumption you made at the beginning may turn out to be wrong in the end. This means you can throw away a lot of code and start all over again. A frustrating and time-consuming way of developing, in our opinion.

"Your most unhappy customers are your greatest source of learning"

**Bill Gates**

## How to give feedback

When giving feedback, try to be as specific as possible. Developers cannot read anyone's mind and certainly are no domain experts. So, don't take anything for granted. To make sure feedback is as specific as possible, always take the following aspects into account:

- **Who**: Developers want to know who provided the feedback so they can reach out to them to find a good solution together. Mention the feedback came from you.
- **Location**: To which part of the app is this feedback relevant? In the input part of the interface? Or some visualisation? A calculation? The menu? Etc. You could also add a screenshot to clarify it even more.
- **Condition**: Did you find a bug? Please also report under which conditions this bug happens so it can be reproduced and checked by the developer. For example: I get error X when using input X when toggle button Z is activated.
- **Priority**: Feedback is good, but too much can be confusing. Please let people know how important this feedback is. For example: not needed – nice to have – should have – must have – urgent to have.
- **User story**: Can you relate this feedback to a user story of the backlog?

## Processing feedback

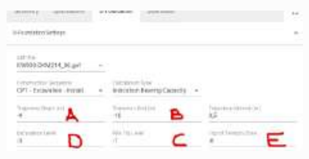
It is recommended to use an issue board to keep track of all the feedback you have gathered from your end-users. In this board, you can indicate which items of feedback already have been processed and which items should still be worked on.

In a small application team, sending feedback directly to the developer seems more convenient than making a whole issue board. However, this isn't the best practice. The more people that get involved in the application, the more reason for why you need to have an issue board to keep track of all the suggestions. Sending the feedback directly via email to the developers creates a lot of extra work for everyone, since that way there is no clear overview of the to-do's.

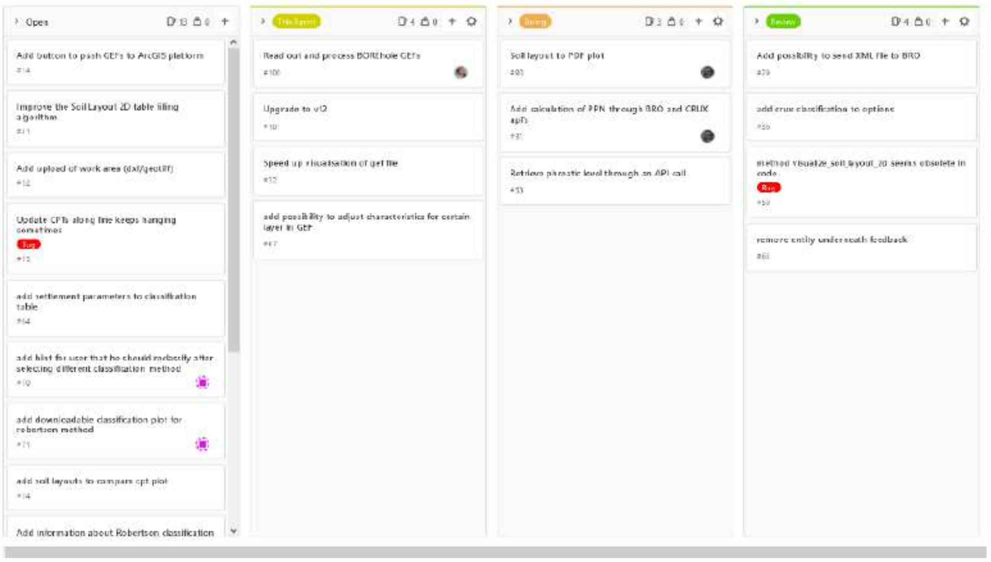
## Creating an issue board

If you want to keep it simple, you can create an issue board in a worksheet. This sheet contains all the feedback, problems, and other comments that you have gathered up to this point. Also, relevant information such as when the feedback was given, who gave the feedback, an extra description, the priority, and if the feedback has been incorporated and approved.

Here is an example of such an issue board in a worksheet:

| # | Date | Author | Description feedback<br>What is the current situation and what is the desired | Priority | Approval | Status |
|---|---|---|---|---|---|---|
| 1 | 26/07/2021 | John Doe | Coordinates are given in [m] but are actually in degrees latitude and longitude. | Must have | | Ready for release |
| 2 | 26/07/2021 | John Doe | Using a download button in blue will fit better with the house style | Nice to have | | Finished |
| 3 | 26/07/2021 | John Doe | Show coordinates of the CPT on a mapview | Must have | | Finished |
| 4 | 26/07/2021 | John Doe | Add the pile tip levels (and other levels) in the geometry visualization | Should have | | Ready for release |
| 7 | 02/08/2021 | John Doe |  <br>To avoid wrong results, following logic should be incorporated in the program:<br>A < D and A>=C<br>B <= C<br>E > A | Should have | | Finished |

If you want to bring it to a more professional level, you can use platforms such as GitHub, GitLab, or others. Here you can make *issue*s for each feedback point, assign it to a developer, classify them, give them labels, and plan when they will be solved (sprints planning). You can also start a discussion on each issue, where you can ask for more input from clients or developers. Additionally, you create a new *branch* to solve each issue or a collection of them. A branch is a copy of the code on which you can work solving the issue without affecting the original code. Once you have solved the issue, you can merge the branch with the 'main code'. This also makes it possible to solve several issues simultaneously with different developers, without breaking the main code.



Open 13 0

Add button to push GEFs to ArcGIS platform
#14

Improve the SoilLayout 2D table filling algorithm
#11

Add upload of work area (dxf/geotiff)
#12

Update CPTs along line keeps hanging sometimes
Bug
#13

add settlement parameters to classification table
#64

add hint for user that he should redraw after selecting different classification method
#10

add downloadable classification plot for robertson method
#71

add soil layouts to compare cpt plot
#14

Add information about Robertson classification

This Sprint 4 0

Read out and process BOREhole GEFs
#100

Upgrade to v12
#101

Speed up visualisation of gef file
#12

add possibility to adjust characteristics for certain layer in GEF
#47

Doing 3 0

Soil layout to PDF plot
#93

Add calculation of PPN through BRO and CRUX api's
#31

Retrieve phreatic level through an API call
#33

Review 4 0

Add possibility to send XML file to BRO
#79

add crux classification to options
#36

method visualize_soil_layout_2d seems obsolete in code
Bug
#58

remove entity underneath feedback
#61

Overview of issues on GitLab

## Increasing adoption of your app

## Increasing adoption of your app

There are several things you should consider throughout the development process that help you add even more value to your application. In this chapter, we will provide a quick recap and some tips and tricks that help you increase adoption of your application even more!

### Tip 1: Work as one team

A good way to add more value to your application is by having clear communication and efficient collaboration between your team members, thus the product owner, domain experts, developers, and the end-users.

Here are some tips you can use to boost teamwork to make the most out of your app:

- **Map your process in detail:** Understanding your process in full detail is vital to a successful application. Bring all people involved together, create a workflow diagram, and discuss all the details. Give everyone enough time to speak. The greatest improvements often arise thanks to this process.
- **Bring developers and end-user together:** It is impossible for developers to build a perfectly good and correct application that fulfils all end-users' wishes without talking to the end-users on the regular. This is also true the other way around. If an end-user has a great idea for an application but is not able to talk to a developer about this, their ideas will never be considered in the first place. Doing everything through an issue board does not lead to the best results. Instead, bring everyone together so they can communicate directly.
- **Have independent testers:** In parallel to your developments, you should have independent testers that test and provide feedback on the developed features. End-users that are experienced with your application may only think in advanced features. New people, however, bring new insights. For example: What is missing for people to have a quick start? You need both kind of testers to create a great application that others will want to use.
- **Create advocates:** The best thing that can happen for the adoption of your application is having an enthusiast end-user who wants to tell everyone the benefits of using your app. People like this do not simply show up in the blink of an eye but come from a close relation to the

product throughout the whole process. After you found testers, help them to own this role. Think about how you can help, maybe it is as easy as simply inviting them to do a presentation or talking to their manager so they have more time for this process.

## Tip 2: User-friendliness is king

It is simple: People will not use your application if it is unfriendly, no matter how wonderful the code in the background is. Don't overcomplicate things. Less is more. Here are a few things to think about:

- **The workflow inside the app should be recognisable** by the end-users. Try to emulate the current workflow as much as possible and include the same steps wherever you can.
- **It should be clear** what an end-user is supposed to do to get results. Use clear descriptions for all element that the end-user interacts with, such as the input fields. Provide additional info (wherever needed) using tooltips or other means.
- **The application should react/calculate fast** otherwise people will lose interest and patience quickly. As a developer you can do a lot to your code to improve performance. Yes, sometimes calculations are just heavy, like running a FEM model. In that case try to divide the fast parts of the calculations from the intensive ones. Maybe it is possible to at least show part of the results quickly?
- **Make your app robust** and avoid that the user gets errors. If they get, make sure they have a clear error message that helps them make the app work again. There is nothing more annoying than having to solve 10 errors before getting any results.

## Tip 3: People don't like black boxes

We often hear that people don't want to use an app because it is a 'black box'. This is understandable. For example, put your self in the shoes of an engineer that needs to use the application to calculate a bridge. He doesn't know what the app is doing, but he is supposed to put his signature on the design without having any idea about all the calculations and processes behind it. Yea, sure... we bet he prefers his old and trusted Excel sheet to do the work instead.

Here some things you can do to help end-users understand what is happening behind the scenes:

- **Use verified calculations.** When looking into how to automate a workflow, you will discover that there are all kind of spreadsheets that people have been using for years, and most important: That they trust. Avoid reinventing the wheel, just incorporating those same models into the app. It will give people more trust.
- **Show additional information** that is relevant for the end-user and include intermediate results. Thus, don't just give the final result but also include the information used to get to there. For example, which material properties, geometry and forces are used in the calculation. Maybe you could also generate a calculation report?
- **Make models available**. Maybe the end-user does not need to know all the ins and outs of your code, as long as the calculation can be verified. For example, does the app optimise a design based on a FEM model or Excel sheet? Make it possible to download the final model/sheet, so that the end-users can check it independently.

## Wrapping it up

## Wrapping it up

Like it is explained throughout this manual, developing an application is a circular process without a set-in-stone end goal. Just as the definition of the word development, the process of building an application is something that is “growing and improving along the way”. You start with a big idea, a great complex process, that you try to make easier. If you think big and clearly define the point on the horizon that you are working towards, you have a good starting point. Start little, with easy and small steps. Then, become more practised as you gain experience and review your previous thoughts and creations. Only after doing that you can continue to the next step. If you develop your application according to this way, you can continuously keep a good overview, make clear progress, keep receiving and processing feedback, and add the most value in the end!

We hope this guide has been proven useful throughout your journey of building your own application. If you have any questions or if you are interested in the possibilities an application development platform can offer you, don’t hesitate to contact us. More information about VIKTOR and ways to get in touch can be found on the next page.

## About VIKTOR

VIKTOR makes it really easy to build powerful web applications and share them with everyone you want. The apps have a user-friendly and interactive interface, where you can present results with all kinds of visualisations, like 3D models, graphs and maps.

You don't need to worry about integrating with different software packages, the front-end, back-end, database management, user-management system, etc.. We automate all the boring stuff,  so you can focus on creating awesome apps that provides real value.

### Automate the boring.
### Engineer the awesome!

Ready to streamline your process? Contact us!

Contact person:

**Anande Bergman**

**Developers Relations**

abergman@viktor.ai