by Nacho Iborra

Development Environments
Block 1
Unit 3: Stages and methodologies in software development

1.3.1. Software engineering

In order to develop software properly, we must follow some steps, or a given approach for this development. **Software engineering** is the branch of computer science that helps us follow these approaches and steps correctly.

1.3.1.1. Stages in software development

In (almost) every software engineering process, we can follow these steps:

1. **Requirements analysis**: this stage includes the communication with the customer to clarify his needs, and an analysis stage to sketch out the main behaviour of the application. It can be divided into two stages:
    - **Requirements specification**: the communication with the customer, in which we will arrange some interviews or other ways of getting information. We must clarify the needs to be fulfilled, and once the requirements have been gathered, we write a document called *requirements specification*, that must be as complete as possible.
    - **Analysis**: from the requirements specification created in previous stage, we must now create a new document where we will use some helpful diagrams to represent the main functionalities of the application and their connections or dependencies.
2. **Design**: from the analysis documents set before, we can now determine how the software will work. Here we will use other diagrams that will help us implement the software.
3. **Implementation**: this stage should be started from previous design stage, using a specific programming language (or more than one).
4. **Testing**: once the product has been implemented, we must test it to check if it fulfills all the requirements and there is no bug. These tests should be checked by someone not involved in the implementation process.
5. **Maintenance**: this last stage consists in improving the performance of the software product, by adding new extensions, or fixing some bugs

To sum up these stages, we could say that software engineering provides an approach that helps us understand the problem to be solved (requirement analysis), design a possible

solution, implement it, test it and then get a better quality or performance (maintenance).

However, all these steps may sometimes be an obstacle, since many developers think that software engineering is something too structured, that do not let them develop software quickly. But we must see sofware engineering as something adaptive, that provides different models and methodologies to be adapted to our development process, as we will see later.

1.3.2. Software lifecycles

A lifecycle is a list of stages through which a system (in this case, a software project) must go, from its birth until it is no longer used. In each lifecycle we set both the stages and the requirements to go from one stage to the next, including the inputs and outputs expected for every stage.

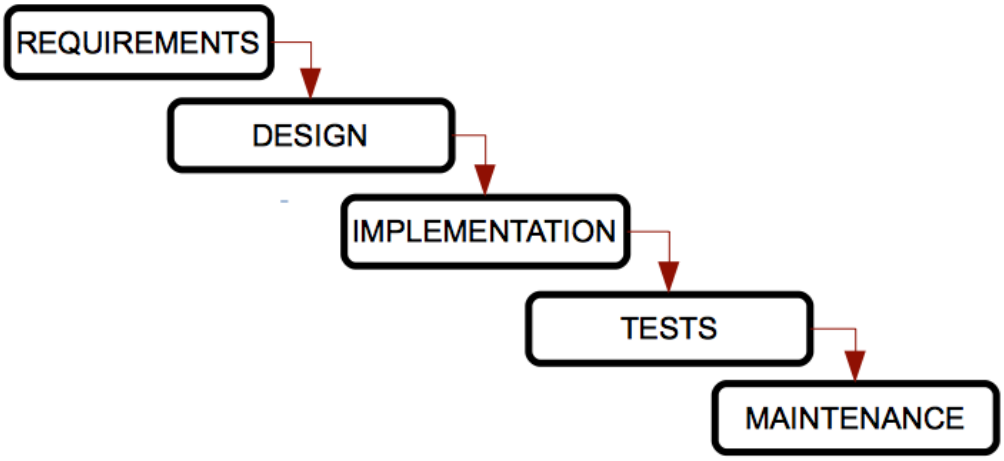
The products generated after each stage are called *deliverables* and they are either part of the input of the next stage or an evaluation of the project at a given point.

Some lifecycles are repetitive, this is, we can go through the same stage more than once, depending on the state of the product. This process is also known as *feedback*.

We are going to see some of the most typical lifecycles in software development, and their pros and cons. In all of them we will find the stages seen before (requirements analysis, design, implementation...), or any variation of them.

1.3.2.1. Cascade or waterfall model

This is the oldest model, and the most widespread. It was created by W. Royce in the 70s, and it sorts the stages of software development rigurously, so that the beginning of a stage must wait for the end of the previous one.



It is called *cascade* model because the stages are placed one below the other, and the process flows from the upper stages to the lower ones, as if they were a cascade.

Pros:

- Suitable for small and well known projects, where all the requirements are perfectly set at the beginning
- Well structured, stages do not mix up
- Easy to use, because of its rigidity

Cons:

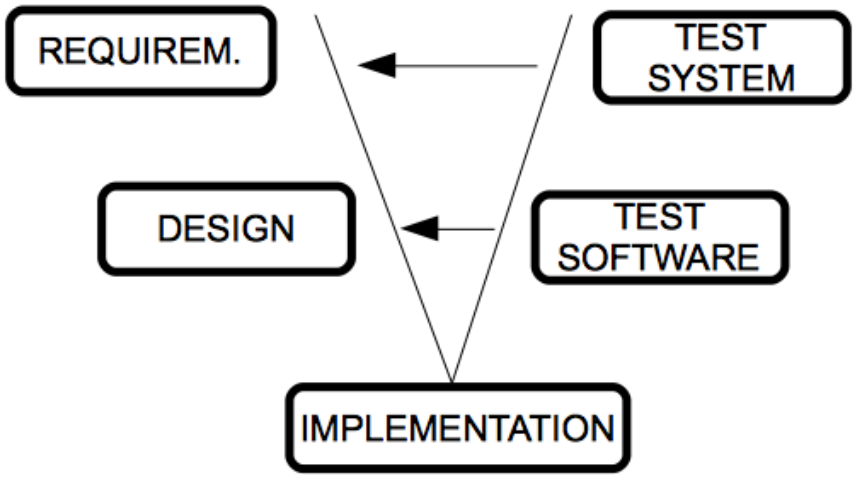
- We can't apply it to most of real life projects, since requirements are barely known at the beginning.
- We can't see any result until the end of the process, so customers must feel worried about that final result.
- It is not usual to have a stage perfectly finished before starting the next one.
- Failures are not detected until the test stage, at the end of the process.

Variations:

There are some variations of this model, like the **Sashimi model**, in which stages are overlapped, like Japanese fish does. In this model, there is some time overlap between two sequential stages; this way, we start the *design* stage while we finish setting the requirements (and we can change them as we go forward with the design), and we start implementing the system while we finish with the design (so that we can improve the design because of some issues detected during the implementation).

1.3.2.2. V model

One of the main problems of the traditional cascade model is that failures are not detected until we reach the final stages of the process. With the V model, tests start as soon as possible, and they are performed in parallel by another work team. This way, tests are integrated in each stage of the lifecycle.



Left branch of the V represents the requirement analysis, design and implementation, and the right branch integrates the tests of each stage. We move along the left branch until we reach the bottom, and then we validate the tests of the right branch, from the most specific ones (unit tests to check some concrete modules of the product) to the most generic ones (integration and system tests). Every time we detect a problem, we go back to the associated stage of the left branch.

Pros:

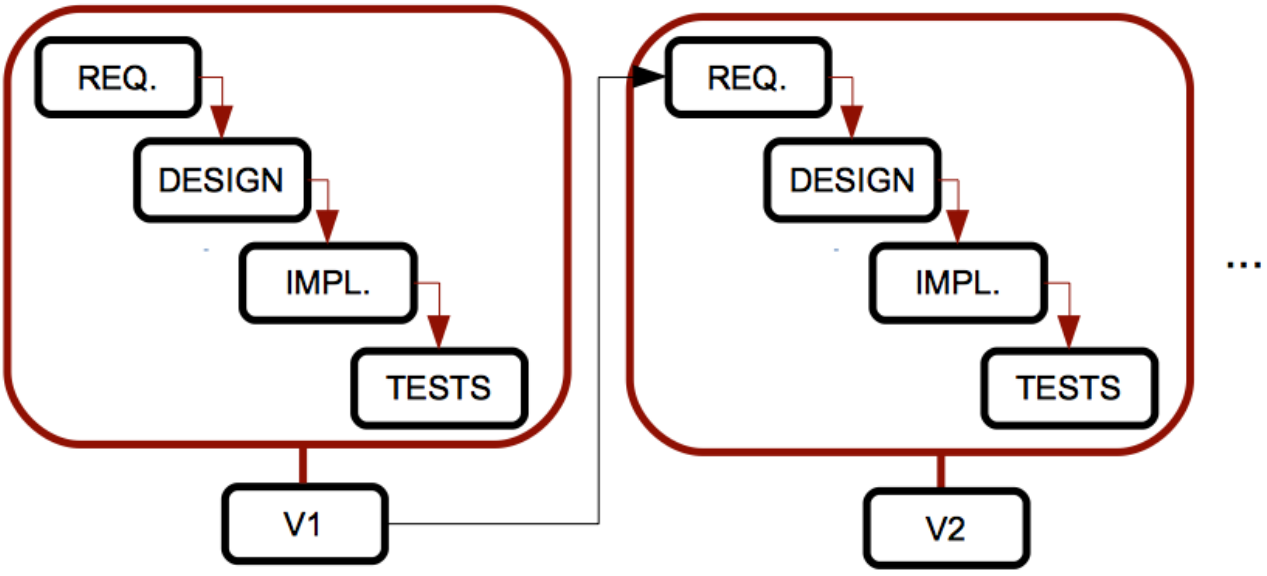
- Easy to use
- There are some deliverables for each stage
- Higher probability of success, due to the tests plans associated to each stage of the process
- Useful for small projects with requirements that are easy to understand

Cons:

- It is also rigid
- User does not see any result until later stages, because we don't develop any intermediate prototype
- Sometimes it is hard to go from the right branch to the left branch to fix the problems

1.3.2.3. Iterative model

The models seen so far are only suitable for projects with easy, well specified requirements, but this is not very usual in real software projects. In order to try to improve this, the iterative model repeats the cascade model, and generates an intermediate version or prototype after each iteration. This prototype can be then checked by the customer, problems can be detected earlier and then we can improve the system.



Pros:

- We don't need to specify all the requirements at the beginning
- Risks are better managed, because we deliver intermediate prototypes after each iteration

Cons:

- If we don't need to have all the requirements at the beginning, they may appear later unexpectedly, and they may affect the design or system architecture significantly.

Variations:

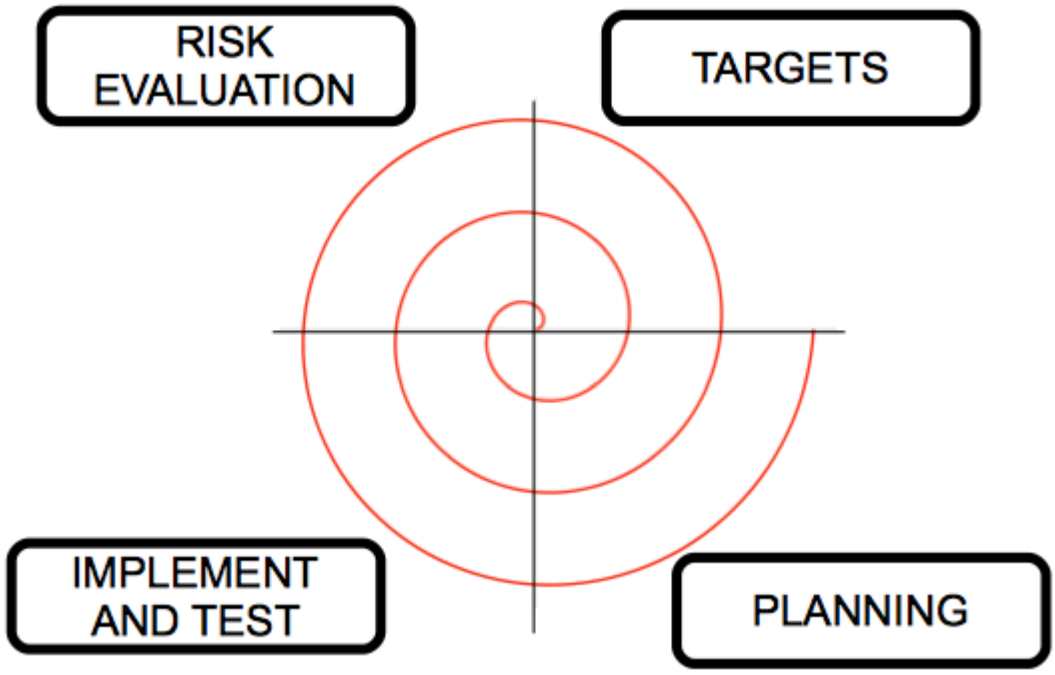
There are some interesting variations of this model, with other names and some particular features:

- **Incremental model**: each prototype only has some few improvements from previous one. This make the prototypes easier to test (we only have to test these small changes). However, we must have a lot of experience in order build these prototypes proportionally.
- **Prototype-based model**: it is based on developing prototypes of the application. At the beginning, we just gather some requirements quickly, make a simple design and have a basic prototype, so that customer can quickly check the application and give us his feedback. The main advantage of this model is that customer is involved in the process from the very beginning. But it may be very costly, since we may develop many useless prototypes. Besides, customer must feel disappointed if he checks some versions of the product that do not work as he expected, and the developer may be tempted to speed up the process to include everything the customer wants, eluding all the quality and maintenance patterns.

1.3.2.4. Spiral model

This model was created by Boehm in 1988, and it tries to combine the cascade and iterative models. Stages are arranged in a spiral divided into four sections, so that each section does one task, and each spiral turn goes through all sections and tasks, generating one prototype after each complete turn.

This model manages the risks of software development. We start from the middle of the spiral, and in every loop we analyze all the development alternatives, we identify the most assumable risks and then perform a spiral cycle. If the customer brings new requirements or improvements, we then evaluate the risks again and perform another cycle, until the product is finally accepted.



- In the **targets** stage we set the final product to reach (requirements, analysis, etc.)
- In the **risk evaluation** stage we identify the possible risks of the project, and choose the options to reduce them as much as possible.
- In the **implement and test** stage we design, implement and test the product, according to the options chosen in previous stage.
- In the **planning** stage we check the product with the customer, and then we decide if we need one more spiral cycle to fix some problems or add some improvements.

Pros:

- Suitable for large and difficult projects
- Risks are minimized
- Implementation and maintenance are both integrated
- We develop prototypes from early stages, so that customer can provide his feedback along the process

Cons:

- We must have a lot of experience to evaluate the risks properly
- This model produces a lot of additional products (reports, prototypes...)
- It can be very costly
- It is not appropriate for small projects

> **Proposed exercises:**

> **1.3.2.1.** Create a table like the one below (in a piece of paper, or in a text editor), and fill each cell with a YES or NO, depending on whether the specified feature is part of the corresponding lifecycle.

	Cascade	V	Iterative	Spiral
It is simple and easy				
It generates intermediate versions of the product				
Suitable for projects with uncertain requirements				
It lets us evaluate the risks of the chosen solution				

> **1.3.2.2.** Our team is going to develop an application to manage the accounting of a cinema company. The customer still doesn't know what he expects from the application, and we want to use a really new technology we don't control yet. According to this information, argue which lifecycle model(s) are the most suitable for our purpose, and which one(s) are NOT recommended.

> **1.3.2.3.** A teacher from the Higher Polytechnic School of the University of Alicante wants to develop a program to help him correct his students' exercises. As he is very busy, he has asked a group of old students to do it for him. We assume that, as he is a computer scientist, he knows perfectly what he wants from the very beginning. It is a small project that should not take a lot of time. According to this information, argue which lifecycle model(s) are the most suitable.

1.3.3. Software development methodologies

As developing software products is not an easy task, there are some different methodologies that we can follow for this development. A **methodology** is a set of techniques and methods that help us face each stage of a lifecycle. This way, we can:

- Optimize the process and the final product
- Use guided protocols for the planning and the development
- Set what to do, how to do it and when, along all the project

1.3.3.1. Elements of a methodology

A methodology is composed of:

- **Stages**: a set of activities that need to be done in order to get an intermediate product, or finish a step of the process
- **Products**: the set of inputs and outputs required/produced by a given stage
- **Procedures and tools**: elements that help us do each task. Examples: code editors, planning software...
- **Evaluation criteria** for the process and the final product, in order to determine if all the targets have been reached

1.3.3.2. Methodology types

Some methodologies are **traditional**, they focus on controlling every part of the process at any time. They set the activities to be done, the deliverables that must be produced in every stage, and the tools and standards that will be employed. For certain projects, these methodologies can be too rigid, and hard to adapt.

Other methodologies rely on the personal factor, and the final product itself. These are called **agile** methodologies, and they focus on some aspects such as the communication with the customer, the development with really short iterations, or the adaptability to different project types. They are really welcome for projects with uncertain requirements, or with a critical development time.

Mosf of these methodologies (both traditional and agile) are based on some of the lifecycle models seen before, specially the iterative and spiral models. Both of them apply several iterations over the software development stages, generating some intermediate products along the process.

1.3.3.3. Examples of traditional methodologies

As we have said before, traditional methodologies focus on the planning and the sequence of stages to follow, instead of the product or the customer. Let's see some of the most popular ones.

1.3.3.3.1. Rational Unified Process (RUP)

Rational Unified Process (RUP) is an iterative framework created by Rational Software Corporation, which is part of the IBM company. But, instead of being what a traditional methodology is supposed to be, RUP is not a rigid framework. It can be adapted to different projects, choosing the most suitable elements for each one.

RUP methodology is based on spiral lifecycle model, and it tries to improve the drawbacks of the cascade model. Besides, it also tries to include the object oriented paradigm through UML (Unified Modeling Language). We will talk about it in later units.

Modules

RUP is structured in some elements called **modules**. These modules are:

- **Roles**: they define a set of competences, abilities and responsibilities, so that people with a specific role must have all of them.
- **Work products**: they are the result of a task, including documents and secondary models that may be produced
- **Tasks**: work units, which are assigned to a specific role, and produce a given work product.

These three modules define who does the job (role), what it is about (work product) and how he does it (task).

Stages

According to RUP methodology, the lifecycle of a product consists of four stages

- **Inception**, where we define the scope of the project. We create a model for the business (gathering some documentation about the company, finding out its weaknesses and strenghts, its capacity of producing benefits...), and then a requirements analysis for the project to be developed.
- **Elaboration**, where the project is analyzed more in depth, so its main architecture is defined. We create an initial design an implementation, which are considered a basis. This stage, along with the inception, are targeted on understanding the problem to be solved, skipping the most dangerous risks.
- **Building**, where the application is finally designed and implemented. We can need many iterations of the spiral model, with many intermediate prototypes, to reach our target.
- **Transition**, where the final product is prepared for its final delivery to the customer.

We can think that this methodology is very similar to the cascade model, but one of the strenghts of RUP relies on the iterations that can be performed at any stage. Besides, each stage has a final target to reach.

> **Proposed exercises:**

> **1.3.3.1.** Search on the Internet about all the possible diagrams that we can use in UML (*Unified Modeling Language*), which is an essential part of the RUP methodology. These diagrams can be of two types: structural diagrams or behavioral diagrams. Identify at least three diagrams of each type.

1.3.3.3.2. Microsoft Solutions Framework (MSF)

MSF is a customizable methodology that applies a traditional approach to develop a software product. As you can figure out, it was created by Microsoft, and it can be applied not only to software development, but also to other computer issues, such as network planning, infrastructures...

Stages

As we have seen for RUP, MSF is also based on the spiral model. The process consists of 5 iterative stages. At the eand of each one we must have reached a given target, and completed a set of deliverables.

- **Vision**: we evaluate the business model, the benefits that we can have, restrictions, scopes... We must also get a requirements specification, an initial risk assessment, and a general overview about the company for which we are going to develop the software product. It may be equivalent to the *Inception* stage of RUP.
- **Planning**: the project development and its architecture are planned in this stage, so that we must follow a schedule. This way, we generate a list of tasks to be completed, people

involved in each task, responsibilities, costs... trying to avoid the most dangerous, potential risks.
- **Development**: we begin to impleent the application from the very primary functionalities. We deliver some prototypes to be tested and evaluated by the customer
- **Stabilization**: the product is tuned so that the customer can test it completely. The set of tests is also documented before the customer agrees with the final product.
- **Implantation and support**: the application is installed in the customer's company, and an additional support may also be offered depending on what was initially specified.

The main benefits of this methodology are its adaptability to different projects (as RUP does), and the cooperation with the customer. Their main drawbacks are the excessive documentation that needs to be created, and the dependence on Microsoft products.

1.3.3.4. Examples of agile methodologies

Traditional methodologies are really useful for big projects (in terms of time and budget), but they have some drawbacks for projects not so big, or with a close deadline, or with uncertain requirements. For instance, they produce too much documentation, and they are overplanned.

To face these other projects, many teams have tried with agile methodologies, which are specially suitable for small projects, to be developed in a short time, with teams composed of less than 10 people. With these methodologies, we promote the teamwork, our own responsibility, the customer needs and the targets of his company. Face to face communication among the members of the team, and with the customer, is very regular. This way, team members share their progresses and problems, and they have a quick feedback from the customer.

In order to fulfill its purpose, agile development makes small project increments, with a minimum planning. Each increment performs a whole iteration over the software stages (requirements, design, implementation, tests...), in a short period of time (usually 1 - 4 weeks), which is known as *timebox*. This way, we minimize the general risk, and the project can be adapted to many changes along its development. Documentation is only generated when it is really needed, and the aim is to get a working prototype after each iteration, even though it has really reduced functionalities.

1.3.3.4.1. The agile manifest

The agile manifest is a specification created in 2001 that gathers the main principles that a methodology must have in order to let the team build a software project quickly and facing the changes that it may have in the future.

Some of the most important principles are:

- People are the main success factor of a project. It is better to create a good team and let it configure its own working environment, than building the environment and force the

team to get used to it.
- We should not produce any documentation unless it is needed to make a decision immediately.
- There must be a constant interaction between the customer and the work team.
- The ability to face the changes that the project may have during the development is more important than to follow a rigid planning from the beginning.
- The main priority is to satisfy the customer through the continuous delivery of working software.
- Face to face conversation is the most efficient method to transmit information in a work team.
- Simplicity

1.3.3.4.2. Some usual practices

Besides the principles gathered in the agile manifest, some agile methodologies use certain practices. Let's see some of them.

Pair programming (PP)

Pair programming is a technique that offers many advantages. Two programmers work together in the same computer. One of them is typing code (driver) and the other one is checking what is being typed (observer). Both swap their roles frequently (every 30 minutes, for instance). The observer is in charge of guiding the work of the driver, and providing any ideas to solve future problems.

The main advantages of this technique are:

- We produce shorter programs, with less errors and better designed (since code must be readable for both members of the team).
- It is useful for learning, if one of the members is very experienced, and the other one is a newbie, or if both have different knowledge that can be transferred to each other.
- If one of the members leaves the team, there can be another one who takes his place supported by the old member, so that the deadlines may not need to be modified.
- There are less interruptions along the process, as both members swap their tasks and they can work more continuously.

However, it also has some drawbacks:

- Some engineers prefer to work on their own.
- A newbie may feel intimidate if he works with someone which is more experienced than him, and vice versa (experienced workers may feel boring to help a newbie).
- It is more costly (we have to pay two salaries for a single job).
- There may be some annoying working habits in the team

Test driven development (TDD)

This is a technique that uses short iterations based on test cases previously written. This way, each iteration produces the code needed to pass these tests, and once they are passed, we integrate this code with the previous one and optimize it. So the process is as follows:

1. We add a new test to current set
2. We write the code to pass this last test
3. We run again the test set and check that every test is passed
4. We refactor the final code to optimize it
5. Go back to step 1

It is important to have a set of unit tests that can be launched automatically, so that we can add more tests from time to time, and we can re-launch it at every moment. We will see later in this module some techniques to define test cases and test sets.

So the main idea is to define the tests that the system must pass before writing the corresponding code, so that we make sure that the application can be checked, and we define tests for each feature. This way, we avoid writing unnecessary code (this is, code that is not bound to any test).

1.3.3.4.3. Example: Scrum

Scrum is an agile methodology that can be used in complex projects. It uses iterative and incremental processes, and can be applied to either software products or other scopes. Its name comes from the "melée" that rugby players do.

Roles

The main Scrum roles are:

- **ScrumMaster**, it is something equivalent to the *project manager*, although this role does not implicitly exist in Scrum, since the own work team is self-managed. His main task is to make sure that the Scrum method is used properly, and there are no external influences that may disturb it.
- **ProductOwner**, who represents the customer, although he doesn't need to be part of the customer's company. He may even be someone from the work team.
- **Team** of developers.

Process

To begin with, the work team must gather the user requirements from the customer (both managers and employees who are going to use the application). The application features are gathered through **user stories**, which are basically a set of paper cards in which the customer describes the features of the system. Each story must be understandable and concrete, so that the team can implement it in a few weeks. Each user from the customer company is asked to write what he expects from the application. The whole set of stories are packed in a collection called **backlog**. From this backlog, some stories will be finally part of the application (the rest will be discarded).

From this point, we start the development process, based on iterations called **sprints**. Each iteration takes 2 - 4 weeks to be completed, and produces a prototype or operative version of the product. In this increment, we add to the application some requirements extracted from the *backlog*. The set of requirements to be added in each iteration is decided in a planning meeting, where the *Product Owner* chooses some items to be added, and the team decides which ones can be added in next iteration. Then, during the iteration, the *backlog* freezes, this is, we can't change any previous requirement until next iteration begins.

Besides the meeting at the beginning of the iteration, there are also daily meetings in Scrum, where people discuss about the state of the project, what has been done and what is about to be done. There is also a final meeting at the end of the iteration to check the version or prototype obtained.

> **Proposed exercises:**

> **1.3.3.2.** In this video you have a summary of Scrum methodology. After watching it, try to answer to these questions:
>
> 1. What is a *sprint* in Scrum?
> 2. Who is in charge of prioritizing the tasks to do in every *sprint*?
> 3. How can the final delivery date of a *sprint* be calculated?

1.3.3.4.4. Example: Kanban

Kanban is another agile methodology, which is really easy to apply. Its name is a combination of two Japanese words: *kan* ("visual") and *ban* ("card"), so we can figure out that the main component of this methodology consists in using cards, that represent the different tasks that we must complete in the development process.

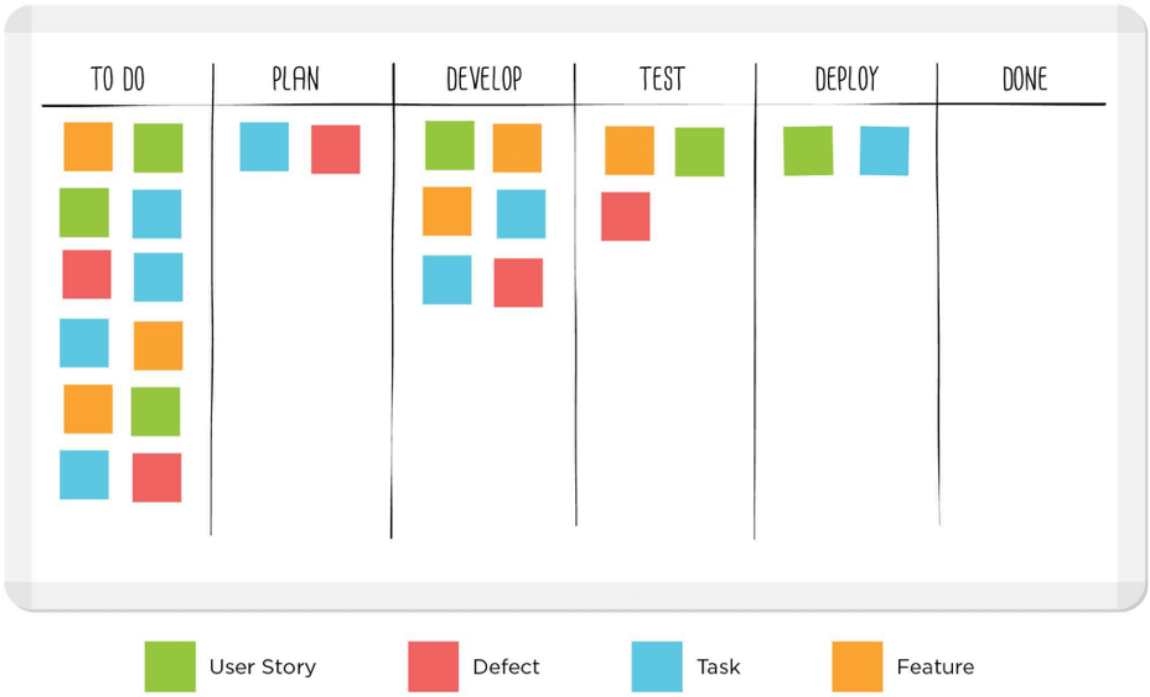
The origins of Kanban methodology date back more than 60 years. In the late 1940s Toyota began to optimize its engineering processes, based on the same model used by supermarkets to optimize their stock. As inventory levels should match consumption patterns, the excess stock can (must) be controlled. This way, Toyota could align their inventory levels with their actual consumption of materials. Workes passed a *card* between teams when a bin of materiales had been emptied, indicating the exact amount of material needed. The warehouse would have a new bin of material ready to be delivered to the factory, and then they would send a new kanban to the supplier to provide a new bin.

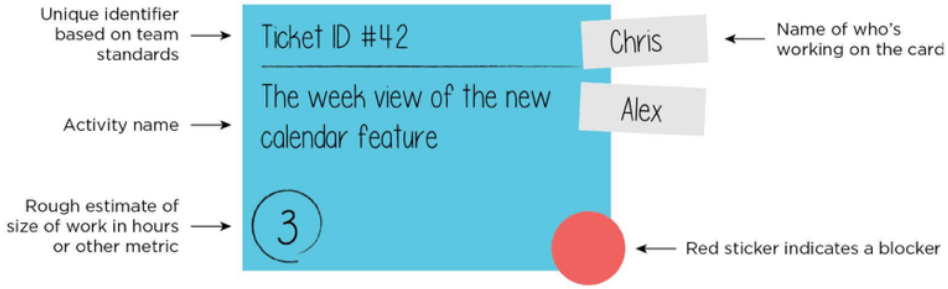
Applied to software development processes, Kanban allows teams to match the amount of *work in progress* (WIP) to the team's capacity. This gives teams more flexible planning options, faster output, and clearer focus.

Process:

The general term to refer to Kanban methodology is *flow*, since work flows continuously through the system instead of being organized in timeboxes, as Scrum does with its *sprints*.

Kanban uses visual mechanisms, such as **Kanban boards**, so that team members can see the state of every piece of work at any time. These boards can be either physical and/or virtual.





The main function of the Kanban board is to ensure that team's work is visualized and all blockers and dependencies are immediately detected. The most basic board has three sections (*To Do*, *In Progress* and *Done*), but we can add as many columns and states as we need for our particular case.

Any section or column of a Kanban board can be filled with **Kanban cards**. They reflect critical information about a particular work item, giving the whole team information about who is responsible for that item, a brief description of the job to be done and an estimation of how long it will take

Principles:

Some of the main principles of Kanban methodology are:

- **Limited *work in progress***, or *stop starting* and *start finishing*, this is, the team should not start another issue until the current one has been finished.

- **Guaranteed quality**: everything needs to go well at the first opportunity, there's no error margin. This way, speed is not as important as quality, because fixing bugs may be costly.
- **Waste reduction**: we must do just what we need, and do it well.
- **Flexibility**: next step is decided from the *backlog*, by choosing the next task to be completed. So we can prioritize the chosen task depending on the needs of each concrete moment.

Kanban vs Scrum

Kanban has some similarities with Scrum methodology, since they are both agile methodologies: both require collaborative and self-managed teams, and both focus on releasing software very often. However, there are some important differences between them:

Kanban	Scrum
No prescribed roles	ScrumMaster, ProductOwner...
Continuous deliveries	Timeboxed sprints
Changes can be made at any time	No changes allowed during sprint

Scrum is more appropriate in situations where work can be prioritized in batches, so that each batch can be solved in one or some sprints. But Kanban can be more suitable in envinroments with a high degree of variability in priorities.

> **Proposed exercises:**

> **1.3.3.3.** In this video you have an example of Kanban methodology. Try to answer to these questions after watching it:
>
> 1. What is the main function of WIP limits? What happens when we try to move a task to a column that exceeds this limit?
> 2. How can Kanban and Scrum be combined? Which is the main purpose of adding Scrum to Kanban?