Course Book

DATA SCIENCE SOFTWARE ENGINEERING

DLBDSDSSE01

iu

INTERNATIONAL UNIVERSITY OF APPLIED SCIENCES

DATA SCIENCE SOFTWARE ENGINEERING

MASTHEAD

Publisher:
IU Internationale Hochschule GmbH
IU International University of Applied Sciences
Juri-Gagarin-Ring 152
D-99084 Erfurt

Mailing address:
Albert-Proeller-Straße 15-19
D-86675 Buchdorf
media@iu.org
www.iu.de

DLBDSDSSE01
Version No.: 001-2023-0811
N.N.

TABLE OF CONTENTS

DATA SCIENCE SOFTWARE ENGINEERING

INTRODUCTION

WELCOME

SIGNPOSTS THROUGHOUT THE COURSE BOOK

This course book contains the core content for this course. Additional learning materials can be found on the learning platform, but this course book should form the basis for your learning.

The content of this course book is divided into units, which are divided further into sections. Each section contains only one new key concept to allow you to quickly and efficiently add new learning material to your existing knowledge.

At the end of each section of the digital course book, you will find self-check questions. These questions are designed to help you check whether you have understood the concepts in each section.

For all modules with a final exam, you must complete the knowledge tests on the learning platform. You will pass the knowledge test for each unit when you answer at least 80% of the questions correctly.

When you have passed the knowledge tests for all the units, the course is considered finished and you will be able to register for the final assessment. Please ensure that you complete the evaluation prior to registering for the assessment.

Good luck!

SUGGESTED READINGS

GENERAL SUGGESTIONS

Brookshear, G., & Brylow, D. (2019). *Computer science: An overview*. Pearson Education.

Hunt, A., & Thomas, D. (1999). *The pragmatic programmer: From journeyman to master*. Addison-Wesley.

Martin, R. C. (2008). *Clean code*. Prentice Hall.

Sammons, A. (2019). *Agile project management with Scrum + Kanban 2 In 1: The last 2 approaches you'll need to become more productive and meet your project goals.* M & M Limitless.

Stephens, R. (2015). *Beginning software engineering*. John Wiley & Sons.

UNIT 1

Arif, S. U., Khan, Q., & Gahyyur, S. A. K. (2010). Requirements engineering processes, tools/technologies & methodologies. *International Journal of Reviews in Computing*, *2*(6), 41—56.

UNIT 2

Cesarotti, V., Gubinelli, S., & Introna, V. (2019). The evolution of project management (PM): How agile, lean and six sigma are changing PM. *Journal of Modern Project Management*, *7*(3), 1—29.

Ries, E. (2011). *The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses* (pp. 1—78). Crown Publishing Group.

UNIT 3

Kochhar, P. S., Xia, X., & Lo, D. (2019). Practitioners' views on good software testing practices. *Proceedings of the 41st international conference on software engineering: Software engineering in practice (ICSE-SEIP)* (pp. 61—70). IEEE.

Zhang, J. M., Harman, M., Ma, L., & Liu, Y. (2019). *Machine learning testing: Survey landscapes and horizons.* arXiv.

Available online.

UNIT 4

Gries, P., Campbell, J., & Montojo, J. (2017). *Practical programming: An introduction to computer science using Python 3* (pp. 1—65, 297—317). The Pragmatic Bookshelf.

UNIT 5

Kim, G., Humble, J., Debois, P., & Willis, J. (2016). *The DevOps handbook* (pp. 1—100). IT Revolution Press.

Rao, D. (2019). *Keras to Kubernetes. The journey of a machine learning model to production* (pp. 223—289). Wiley.

LEARNING OBJECTIVES

The course **Data Science Software Engineering** gives a detailed overview of the relevant methods and paradigms that data scientists need to know in order to develop enterprise-grade models. The goal of this course is to give students a common foundation in software engineering so that they can efficiently communicate their requirements with engineering teams, work alongside engineers, and articulate their technical ideas on the job. On successful completion of this course, students will be able to understand the concept of project management approaches, apply Agile approaches in software development, create their own automated software tests, understand various software paradigms, and evaluate the steps necessary to bringing models into a production environment.

You will be introduced to traditional project management, including requirements engineering, the waterfall model, and the rational unified process (RUP). Whatever approach you find yourself using at work, project management is the basis of working productively in teams, and having a good understanding of best practices goes a long way. We will then discuss Agile project management, including an introduction to Kanban and Scrum. Here, you will also learn how to best move a team from traditional project management methodology to Agile, an whether this solution is a good fit for your team.

We will address the process of and reasons for testing your sofware. You will also see example of how to build and run your own tests using pytest. Special focus is given to the topic of testing and the consideration of how to bring a model into a production environment. Next, we will explore relevant software development paradigms such as test-driven development, pair programming, mob programming, and extreme programming. Leading on from this, the final unit explores how to bring your model into a production environment. This brings together concepts from earlier units and introduces concepts such as versioning, DevOps, and MLOps.

UNIT 1

REQUIREMENTS ENGINEERING

STUDY GOALS

On completion of this unit, you will have learned …

- what software project requirements are and what the software requirement engineering process involves.
- some methods for software requirement engineering processes.
- traditional software management methods, such as the waterfall method.
- about the rational unified process model for software development life cycle management.

1. REQUIREMENTS ENGINEERING

Introduction

According to the Project Management Institute (PMI) (2008), a project is "a temporary endeavor undertaken to create a unique project service or result." The development of software for an enhanced business process, the construction of a house or bridge, and the expansion of businesses into a new geographic market all are examples of projects.

Project management is the employment of knowledge, experiences, skills, tools, and methods to perform activities to satisfy the project requirements. Software project management is the art and science of planning and leading software projects (Stellman & Greene, 2005). By using the software project management disciplines, software projects are planned, monitored, and controlled to deliver software solutions according to their requirements.

A failure in software project management results in project shut down or the delivery of a software solution that does not meet the client's requirements. Among the most common causes of software project management failures we can mention

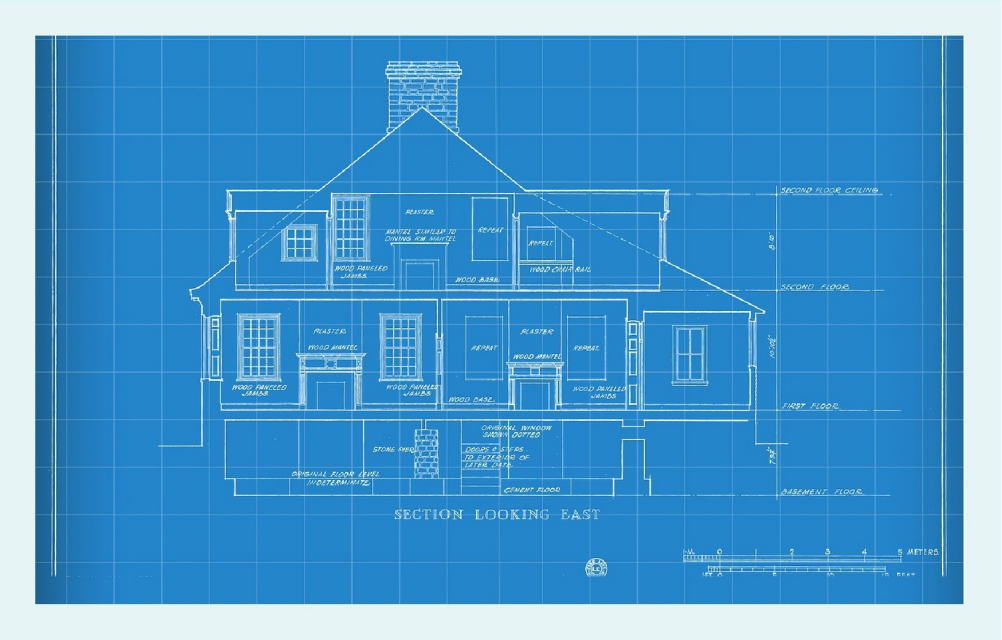
- unreliable or unarticulated project objectives and goals,
- incorrect estimations of required resources (software developers, budget, etc.),
- poorly specified system requirements,
- inadequate reporting of the project's status to the project stakeholders (client, developers, project managers, etc.),
- unmanaged risks,
- application of immature technologies and methods,
- inability to manage the project's complexity,
- poor development practices,
- stakeholder politics, and
- financial pressures (Charette, 2005).

The goal of software project management models is to mitigate the aforementioned causes of failure to deliver the right software solution to the customers with the provided resources. Since the birth of software products, many software project management models have been developed by software companies. In this unit, we will focus mainly on two of the classic (traditional) software management models: the waterfall model (which focuses mainly on matching user requirements to a delivered software solution) and the rational unified model as an iterative software development model. As the project requirements have a critical role in both of these models, we will start this section with an introduction to software requirement management and engineering.

1.1 Requirements Engineering

Every project starts with a statement of requirements. Let us assume that you are going to build your dream house. For this reason, you purchase the land and hire a construction team. However, there is still something missing: a detailed blueprint of your dream house. This blueprint is a set of instructions and drawings, which includes information about the house, such as floor plan, exterior elevations, foundation and basement plans, roof framing plans, and so on. This blueprint gives the construction team a clear idea and image of what your expectations are (requirements) regarding your dream house and also the requirements needed to define their job as completed. The Project Management Institute (2008) defines a project requirement as "a condition or capability that must be met or possessed by a system, product, service, result, or component to satisfy a contract, standard, specification, or other formally imposed documents. Requirements include the quantified and documented needs, wants, and expectation of the sponsor, customer, and other stakeholders." Simply put, project requirements specify the characteristics of the end product of said project.

Figure 1: Blueprint of a Residential House



Source: Hiraeth (2018).

We can apply the general definition of a project requirement to software development projects. The IEEE standard glossary of software engineering terminology defines a requirement in a software project in three categories. A requirement can be considered

1. a condition or capability required by a user to solve a problem or complete a task;
2. a condition or capability that must be satisfied or possessed by a system or system component to complete a contract, standard, specification, or other formally imposed documents; or
3. a documented copy of a condition or capability as in (1) or (2) (IEEE Computer Society, 1990).

The above definition of requirement makes it dependent on the concrete organization and industry it is implemented in, as well as their users. In defining the requirements of a software project, the stakeholders should try to answer primarily "what" should be done instead of "how" it should be done. In the beginning, requirements are formulated as questions and should only express needs, not suggest solutions.

There are numerous approaches to classify requirements and three of them are shown as instances in the table below (Aurum & Wohlin, 2005). The classes shown in this table are only some of the many possible ways of categorizing requirements. The elements in class 1 are divided into functional and non-functional requirements; however, there is often no clear line separating the two. For example, we can assume that security is classified as a non-functional requirement. However, during the implementation of the security requirement, developers encounter the user authorization, which is a functional requirement.

The elements in class 2 are classified from the perspective of different areas such as design, product, goal, and application domain. The elements in class 3 are also categorized according to the type of the requirement: primary (defined by the stakeholders) or secondary (derived from the primary requirements). It should be emphasized that these three sample classes are independent. In fact, a requirement (for example, the security requirement) could belong simultaneously to at least one element in each class: security is simultaneously a non-functional requirement (class 1), a design level requirement (class 2), and could be also a primary requirement (class 3).

Table 1: Some Examples of Requirements Classifications

Class 1	Functional requirements	Define "what" the designed system will do
	Non-functional requirements	Define the constraints on the solutions that will satisfy the functional requirement, such as security and modifiability of the software product
Class 2	Goal level requirements	Define business-related requirements
	Domain level requirements	Define problem area requirements
	Product level requirements	Define product domain requirements
	Design level requirements	Defines what to build
Class 3	Primary requirements	Provided directly by stakeholders
	Secondary requirements	Derived from the primary requirements

Source: Aurum & Wohlin (2005).

Requirement Engineering Process

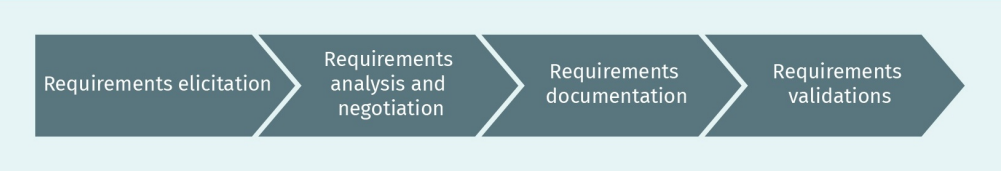
After understanding the meaning of software project requirements, we will briefly discuss the process of engineering the requirements. Requirements engineering is an iterative set of actions that guarantee that elicitation, documentation, refinement, and changes of requirements are sufficiently dealt with through the project life cycle (Coventry, 2015). There are several process models for requirement engineering, such as the purely linear model (Macaulay, 2012), the linear model (Kotonya & Sommerville, 1998), and the spiral model (Boehm, 1988).

The pure linear model proposed by Macaulay (2012) incorporates several requirement engineering activities, such as

- requirements elicitation,
- requirements analysis and negotiation,
- requirements documentation, and
- requirements validation.

In the purely linear model, the requirement engineering process flow sequentially executes each of the four activities as shown in the following figure.

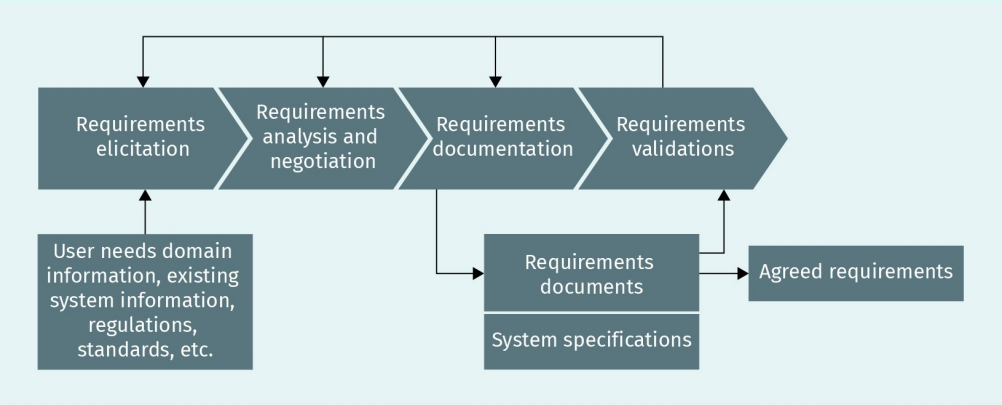
Figure 2: Purely Linear Requirements Engineering Model



Source: Created on behalf of IU (2022).

The linear model proposed by Kotonya and Sommerville (1998) also incorporates the iteration between the same requirement engineering activities as the purely linear model. However, in this model, all of these activities repeatedly overlap during the requirement engineering process, as shown in the following figure. This iteration process continues until all stakeholders agree on the requirements and the requirement specification document is finalized. This model is most appropriate for the cases when the requirements should be pinpoint accurate and validated multiple times by the stakeholders (Shams-Ul-Arif et al., 2010).

Figure 3: Linear Model for Requirements Engineering Process



Source: Alavirad (2020).

Stakeholders in Requirement Engineering

In a daily context, a stakeholder is a person or an organization that has an interest or share in a project or business. The term "stakeholder" is a generalization of the traditional terms of customer or user in requirements engineering to refer to all parties involved in a system's requirements (Glinz & Wieringa, 2007). Pouloudi & Whitley (1997) defined stakeholders as "participants in the development process together with any other individuals, groups, or organizations whose actions can influence or be influenced by the development and use of the system whether directly or indirectly" (p. 3). Usually, instead of considering individuals as stakeholders, people assume roles as stakeholders. Typical stakeholders in a software project include the product managers, different users and administrators from the client and project management sides, and different development teams from the software development side.

The first step in requirement engineering is identifying the stakeholders. For example, let us assume that we are going to obtain the requirements for a software project. According to Glinz & Wieringa (2007), to identify the stakeholders, we should look for people (or roles) who have a keen interest in the system because they will use, develop, manage, operate, and maintain it after its development. In addition, stakeholders can also be the people who design, develop, and test the system, as well as those who manage the development project. Generally speaking, stakeholders are also those people who are involved in the business that the system supports or automates, benefit from it financially, or who are negatively affected by the system. The latter are also called negative stakeholders; for example, the shop floor employees who will be fired after the automation of the production line.

After recognizing the stakeholders, the next step is to prioritize them, as they are not all equally important. For example, it is possible to divide them into critical, major, and minor stakeholders by assessing the risk arising from neglecting them (Glinz & Wieringa, 2007). These terms are defined as follows:

- Critical. A stakeholder is critical when ignoring them results in project failure.
- Major. A stakeholder is major when neglecting them results in a significant impact on the system.
- Minor. A stakeholder is minor when neglecting them results in a marginal effect on the system.

The finalized and agreed requirements (as the end product of the requirement engineering process action) should be documented in the software requirement document (SRD). The SRD—which is a document or a set of documents—ensures that the project's stakeholders are on the same page regarding the software products' goals, scope, constraints, and functional requirements. According to the IEEE 12207.1-1997 standard (IEEE Computer Society, 1998), a typical SRD should contain

- **interfaces**,
- functional capabilities,
- **performance** levels,
- data structures or elements,
- safety,
- **reliability,**
- security or privacy,
- quality, and
- constraints and limitations.

Interfaces
These are shared boundaries between two components within a system or between two systems to exchange information.

Performance
As a general definition, performance measures the effectiveness of a software system with respect to time constraints and allocation of resources (Cortellessa et al., 2011).

Software reliability
This is the likelihood of a software running free from failure during a specified time period in a specified environment.

1.2 Waterfall Model

The process that is used by the software industry to design, develop, test, and maintain the software products is known as the software development life cycle (SDLC). The goal of any SDLC is to produce a high-quality software product according to the requirements defined and agreed upon by the stakeholders. The SDLC defines a methodology to improve the quality of the final software product as well as the development process. A common framework for the SLDC has been also defined by the international standard ISO/IEC 12207 (International Organization for Standardization, 2017).

A typical software development life cycle consists of six main steps, as follows (however, these may vary from model to model):

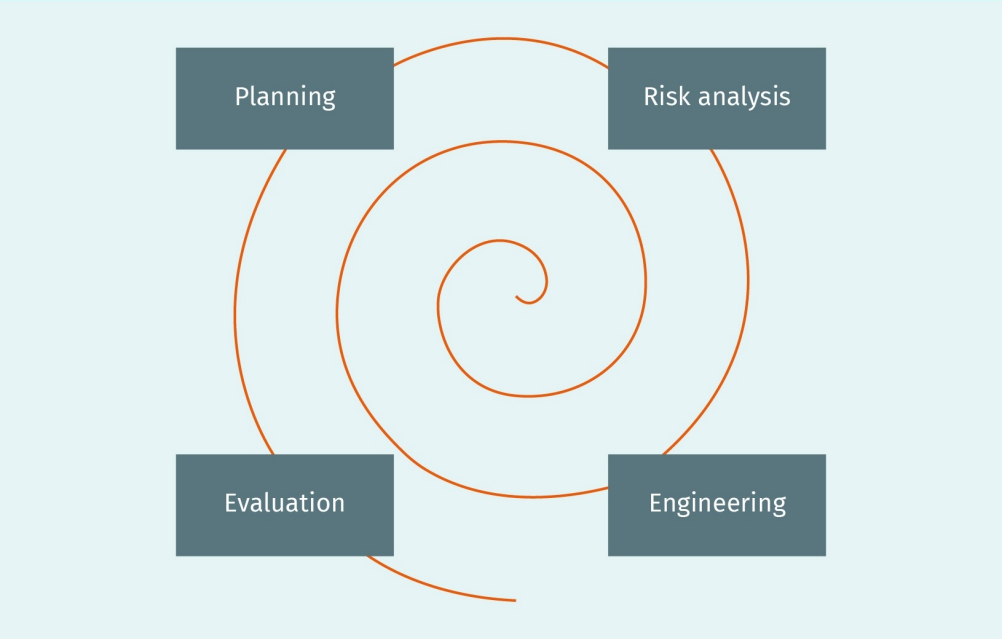
1. requirements engineering
2. system design according to the gathered and analyzed requirements
3. coding the designed software using the appropriate programming languages
4. testing the developed software
5. deployment of the developed software product in the production environment
6. maintenance of the software product during its operational lifetime

There are several different SDLC models, a few of which are defined below:

- The waterfall model is the most straightforward and one of the oldest models for a software development life cycle based on the very simple principle—finish one step before starting the next one.
- The V-shaped model is an extension of the waterfall model (also known as the verification and validation model). In this model, each step will be followed by the corresponding testing phase.
- The iterative model starts with a minimum set of requirements to develop the first version of the software, while a new and more mature version of the final software will be produced in the following iterations. However, the final software product will be delivered at the end of the last iteration, when the product is ready for delivery (Lithmee, 2020). An example of the iterative model is the spiral model. This model goes through four steps (phases) iteratively around a circle (see figure below). The four steps are as follows:
  1. Planning phase to collect the requirements
  2. Risk analysis phase to identify risks and alternative solutions
  3. Engineering or implementing phase to code the software solution using the programming languages
  4. Evaluation phase to evaluate the developed software solutions

In this model, as we follow the spiral from inside to outside (more iterations), the product satisfies more and more of the software requirements.

Figure 4: Spiral Model of Software Development Life Cycle



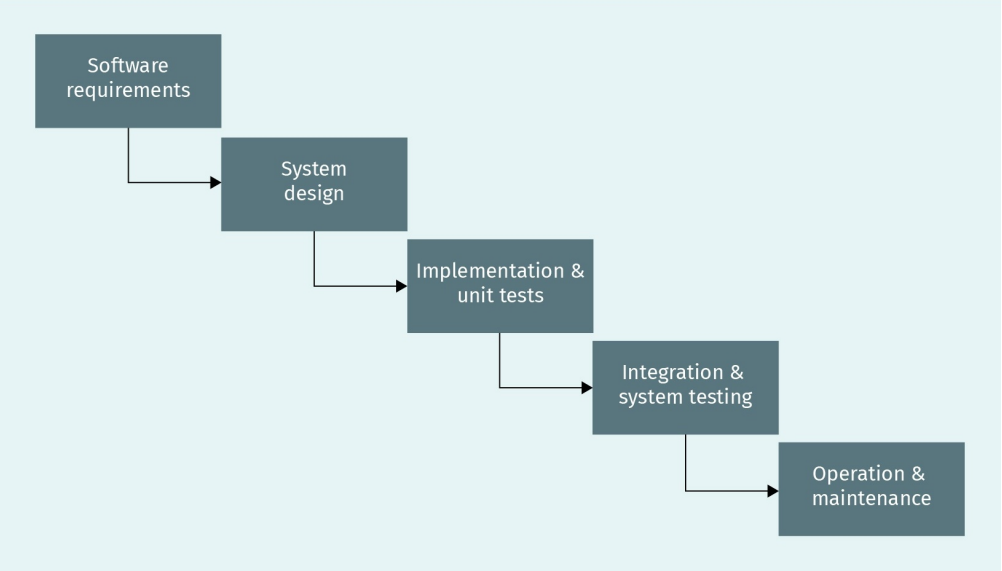
Source: Alavirad (2020).

The Agile model is an iterative development methodology. This model adapts easily to the changes in the requirements. In this approach, the product is divided into smaller incremental steps during which a set of requirements are chosen before a version of the soft-

ware product is developed and discussed with the customer. If the feedback from the customer is positive, then the development team moves to the next set of requirements. The difference between this and the iterative model is that functional software should be delivered at the end of each iteration (Arntz, 2020).

The waterfall model was discussed for the first time by Royce (1970) in a publication in which he tried to explain his personal view about managing large software projects for spacecraft mission planning, commanding, and post-flight analysis. The representation of his proposal for managing the software development life cycle has been depicted in the figure below. In this model, each software project is divided into distinct phases (steps), where each step can be started when the preceding step has been completed and approved. For this reason, this model is also known as the linear sequential model. The author, however, did not exclude the iteration from his model, instead making it so that "there is an iteration with the preceding and succeeding steps but rarely with the more remote steps in the sequence" (Royce, 1970, p. 328).

Figure 5: Steps of the Waterfall Model



Source: Alavirad (2020).

As you can see in the figure above, the waterfall model consists of five steps which are briefly outlined below.

- Software requirement. In the first step of the software development life cycle, the requirements from different project stakeholders will be gathered, analyzed, and documented in the software requirement document (SRD).
- System design. Using the SRD as input from the first step, the development team designs the software system including software and hardware architecture (in simple words, how different building blocks of the software system and the required hardware are put together) and user interface mockups, among other things. The output of this

High-level design
This is the overall system design that covers the system architecture and database design, which describes the relation between different parts of the system.

Low-level design
This is a detailed version of the high-level design that describes the actual logic for each component of the software.

step specifies in detail how the project should be implemented from a technical perspective. The outputs of this step are **high-level design (HLD)** and **low-level design (LLD)** documents.

- Implementation and unit testing. With the system design documents (HLD and LLD documents) in hand, the development team can start coding and implementing the software solution. In other words, this step translates the system design documents into a machine-readable format. To avoid complexity and more easily detect errors and bugs, a software project is divided into smaller units, for each of which one developer or a team of developers is responsible. A unit could be, for example, a single function or a collection of functions and the related codes. After coding each unit in this step, the unit will be tested in isolation from other units to assess whether it performs as expected. Unit testing helps to find the errors and bugs in the very early stage of development.
- Integration and system testing. After implementing and testing individual units, the next step is to put all system parts together to build the desired software system (integration). Now we have the complete system which needs to be tested to see if it satisfies the defined requirements before delivering it to the customer.
- Operation and maintenance. After testing the developed system against the criteria defined in the software requirement document and resolving the possible issues and bugs, the system is deployed into the customer's operational environment (i.e., the infrastructure on which the software solution operates). The maintenance of the system during its life cycle will resolve any issues and failures and also deliver updates to improve the performance of the system.

Now the question is when the waterfall model should be used to manage the life cycle of a software development project. This model is more suitable for

- projects with well-defined and unchangeable requirements,
- projects with well-established and non-dynamical technologies, and
- projects of small size.

The waterfall model has been used historically to develop traditional enterprise software solutions, such as customer relationship management (CRM) systems, human resource management systems (HRMS), and supply chain management systems (SCMS). There are several pros and cons of the waterfall methodology as one of the earliest software development life cycle management methods.

Table 2: Pros and Cons of the Waterfall Model

Pros	Cons
It is easy to understand and to follow the development procedure.	No functional software is produced until the final step.
The start and end criteria are well defined and precise.	Small modifications or errors in the final product are not easy to resolve.
It is suitable for small projects when the requirements are well defined.	For complex projects with changing and dynamic requirements, this approach is not the best solution.

Pros	Cons
Documentation of each step is very valuable for the next step, as well as for future development and improvement.	Documentation sometimes takes a lot of time from the stakeholders.
The client intervention is minimal in this approach and the development team can focus on the development.	As the client intervention is minimal during the development implementation and testing phase, the valuable feedback from the client is missing during the development phase in this method.

Source: Alavirad (2020).

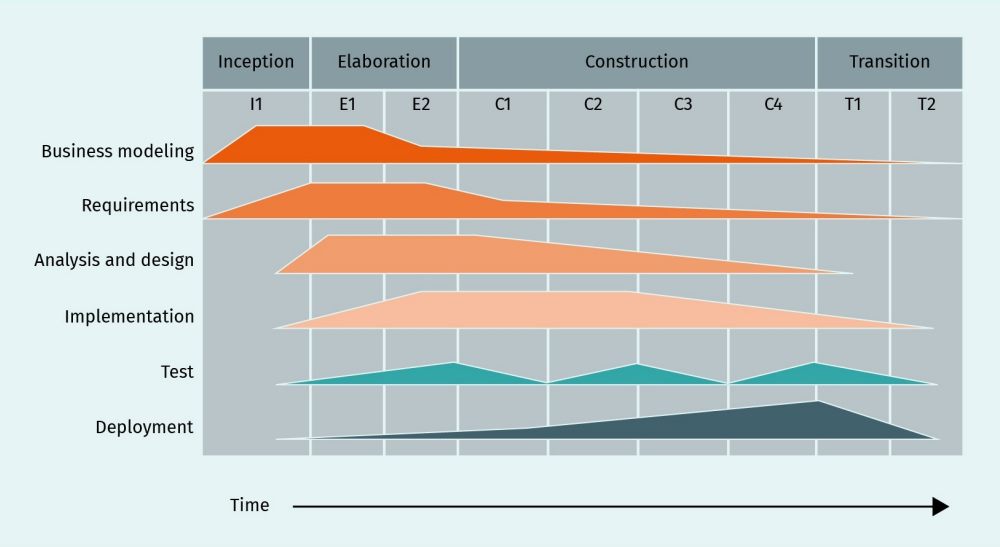
As an example of the failure rate of the waterfall model, we can look to the U.S. Department of Defense (DoD) as one of the most frequent users of this model (Leffingwell, 2007). In the 1980s and 1990s, it was mandatory to manage the DoD's projects using waterfall methodology as emphasized in the published standard DoD STD 2167 (Department of Defence, 1988). The studies showed that 75 percent of the projects failed or have never been used, which resulted in a radical overhaul of the DoD software development standards (Leffingwell, 2007). The failure rate then decreased after revising the internal DoD's standards regarding software project management.

1.3 Rational Unified Process

Rational unified process (RUP) is a software engineering process that manages tasks and responsibilities assignments within a development organization (Jacobson & Bylund, 2000). It was developed by the Rational Software Corporation in the 1990s, which was acquired by IBM in 2003 (Darryl, 2003). Using the modern terminology of software development, it is an iterative software development life cycle (SDLC) methodology.

RUP divides the SDLC into four phases: inception, elaboration, construction, and transition. During these four phases, the core software development activities (i.e., business modeling, requirements, analysis and design, implementation, test, and deployment) happen. To better understand the concept of RUP, we can use a two-dimensional diagram known as the RUP hump, which is shown in the following figure (Krutchen, 2003).

Figure 6: Rational Unified Process Visualization



Source: Dutchguilder (2007).

The horizontal axis represents time and shows the dynamic aspect of the process. It is described in terms of phases (four phases) and cycles (each cycle works on a new generation of the software product and consists of four phases). Each phase is also divided into smaller units called iterations (in the figure above, these go from C1 to C4); each iteration generates a new release of a subset of the final product. Therefore, we have three different time scales: cycles consisting of phases consisting of iterations. The vertical axis represents the static aspect of the process (how it is described in terms of main activities). An activity can happen during different phases.

Next, we will discuss different phases and activities within the RUP model. It is worth mentioning the difference between phase and activities here. Activities are a set of tasks that should be performed to reach a specific goal, whereas a phase is a set of tasks performed over a specific time period, defined by the start and end of the phase. The same activities may occur during different phases of the software life cycle (see figure above).

Time Dimension

The software life cycle is divided into separate cycles, where each cycle works on a new version of the software. One development cycle in RUP is divided into four phases and each phase is concluded with a well-defined **milestone**. The four main phases of each cycle of the software development life cycle could be considered as follows (Gornik, 2017).

Milestone
A milestone is a point in time at which a decision should be made and by which key goals must have been achieved (Boehm, 1996).

1. Inception phase. During the inception phase, all basic business cases for the project should be identified. The business case includes the success criteria, risk assessment, estimate of resources, and the phase plan for the major milestones. The deliveries of this step are a vision document (a document that represents the general vision of the core project's requirements) and a project plan (describing phases and iterations).
2. Elaboration phase. In the elaboration phase, we analyze the problem domain, develop the project plan, and eliminate the highest risk elements of the project. In this phase, we also develop a sound architecture of the system by considering the system's scope and the main functional and non-functional requirements. This phase is the most critical phase of the project because, at the end of it, we should decide whether or not to commit to the next phases. The deliveries of this phase are a software architecture description, a revised risk list of the business case, a development plan for the whole project, and a preliminary user manual.
3. Construction phase. During the construction phase, all component and application features of the systems are developed and integrated into the product. The deliveries of this phase are a software product integrated into adequate platforms, a user manual, and a release note.
4. Transition phase. At this point, the transition phase begins and the software product will transition into the user community. After the product release, there might be some issues that have to be resolved by new developments and bug fixing. We could start the transition phase when the product is mature enough to be deployed into the user environment.

As stated above, each phase of RUP can also be broken into iterations, i.e., complete development loops generating an internal or external release of an executable product or a subset of the final product (such as E1, E2, …, T2 in the figure above). Using the iteration approach in the software development life cycle has certain advantages over the non-iterative models (e.g., the waterfall model), such as an early identification of the risks which allows the team to mitigate them, improve change management, and increase product quality overall (Gornik, 2017).

Process Dimension

The process determines "who" does "what," "how," and "when" and the rational unified process (RUP) is represented through four main modeling elements (Gornik, 2017). They are as follows:

- Workers (who). This element describes the responsibilities of individuals or a team; one individual could have different responsibilities.
- Activities (how). This describes a unit of work that a specific worker should undertake.
- Artifacts (what). This element describes a piece of information produced, modified, or used by a process. If we think of an activity as a function, artifacts are the function's parameters. Source code or a document are examples of artifacts.
- Workflows (when). This describes a meaningful sequence of activities by workers that produce observable values.

The main process workflow of the RUP model can be divided into the following six workflows:

1. Business modeling. In this workflow, we document the business process using business cases (success criteria, risk assessment, an estimate of resources, and the phase plan for the major milestones).
2. Requirements The goal of this workflow is gathering the software requirements from the perspectives of different stakeholders. In this workflow, the use cases (defining the system behavior) are also identified and developed according to the stakeholders' needs. Each use case describes how the system interacts with different actors (such as users, admins, and operators) and what the system does step-by-step.
3. Analysis and design. This workflow defines how the system should be realized in the implementation phase. The output of this workflow is a design model, which is a blueprint for the developer about how they should implement different system functionalities.
4. Implementation. In this workflow, the team defines code organization, implements classes and objects in terms of components (e.g., binaries, source files, and executables), tests the developed units (unit testing), and integrates the individual unit into an executable system.
5. Test. In this workflow, the team verifies the interaction between different units, the integration of units into the software system, and the fulfillment of the requirements. In addition, they also find all issues and resolve them before entering the next workflow.
6. Deployment. At the end of this workflow, the team should deliver the final software product to end-users.

In addition to the workflows, RUP also provides six best practices that make it suitable for a wide range of software development life cycle management (Gornik, 2017). They are as follows:

1. Develop software iteratively. We continually obtain an understanding of the problem through successive refinements until we obtain an effective solution after multiple iterations.
2. Manage requirements. RUP provides knowledge of how to elicit, organize, and document the required functionalities and constraints.
3. Use component-based architecture. RUP provides a systematic approach to design an architecture using existing and new components.
4. Visually model software. Using the unified modeling language (UML), it is possible to visualize the software model to capture the structure and behavior of architectures and components. This visualization enables us to share the architecture of the system more easily with the nontechnical stakeholders.
5. Verify software quality. RUP assists in planning, design, implementation, execution, and evaluation of reliability, functionality, and system performance tests.
6. Control changes to the software. RUP describes how to monitor and track changes to enable iterative development.

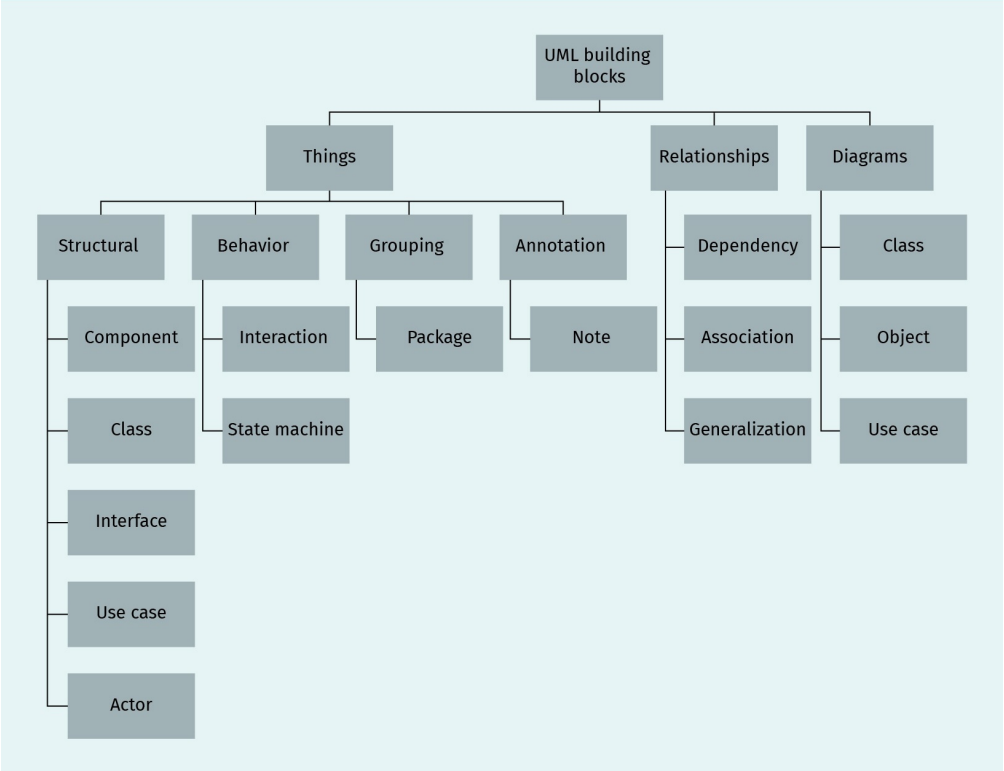
Unified Modeling Language

The idiom "a picture is worth a thousand words" perfectly explains the idea behind the unified modeling language (UML). The main purpose of the UML is to define a standardized approach to visualize the data system design. UML facilitates the modeling, design,

and analysis process for software designers and architects (Bell, 2003). For example, UML is utilized in the rational unified process to visualize this development process. Although UML has been designed for software systems, it is also applicable to other domains, such as visualizing the process flow in a manufacturing unit. We should mention that the UML is not a programming language similar to C++ or Python, but rather a visual language made of visual blocks.

UML was adopted by the Object Management Group (OMG) as a standard in 1997 and the ISO approved the UML standard in 2005 (International Organization for Standardization, 2005). The current main version of UML is 2.0 (Booch et al., 2005). The building blocks of the UML language can be categorized into three main categories, as listed below (Miller, 2003).

Figure 7: UML Building Blocks



Source: Alavirad (2020).

1. Things, i.e., the most important building blocks which explain physical elements
2. Relationships, i.e., representations of the relation (connection) between things
3. Diagrams, i.e., a collection of things and their relations for a specific goal

The “things” building blocks can be categorized the following into the following sub-categories:

- Structural things are nouns that represent the static aspect of the model (physical and conceptual elements). Examples of structural things are the component that describes the physical elements of the system; the class that represents a set of objects with similar properties; the interface that specifies the class responsibilities through a set of operations; the use case that represents a set of actions that should be performed by the system for a specific goal; and the actors representing an entity that interacts with the system.
- Behavior things are the verbs that represent the dynamic or behavior of the model. Some examples in this group are the interaction diagrams representing a group of messages exchanged between elements to perform a specific task, and the **state machine**, representing the state of an object as it is changing during its life cycle in response to events.
- Grouping things bind different elements of a UML model. There is only one concept for grouping things in UML, which is called a package.
- Annotation things are used to add remarks, descriptions, and comments into the UML elements. UML uses the concept of a note for annotating things.

State machine
This is any device that saves the status of an object at a given time.

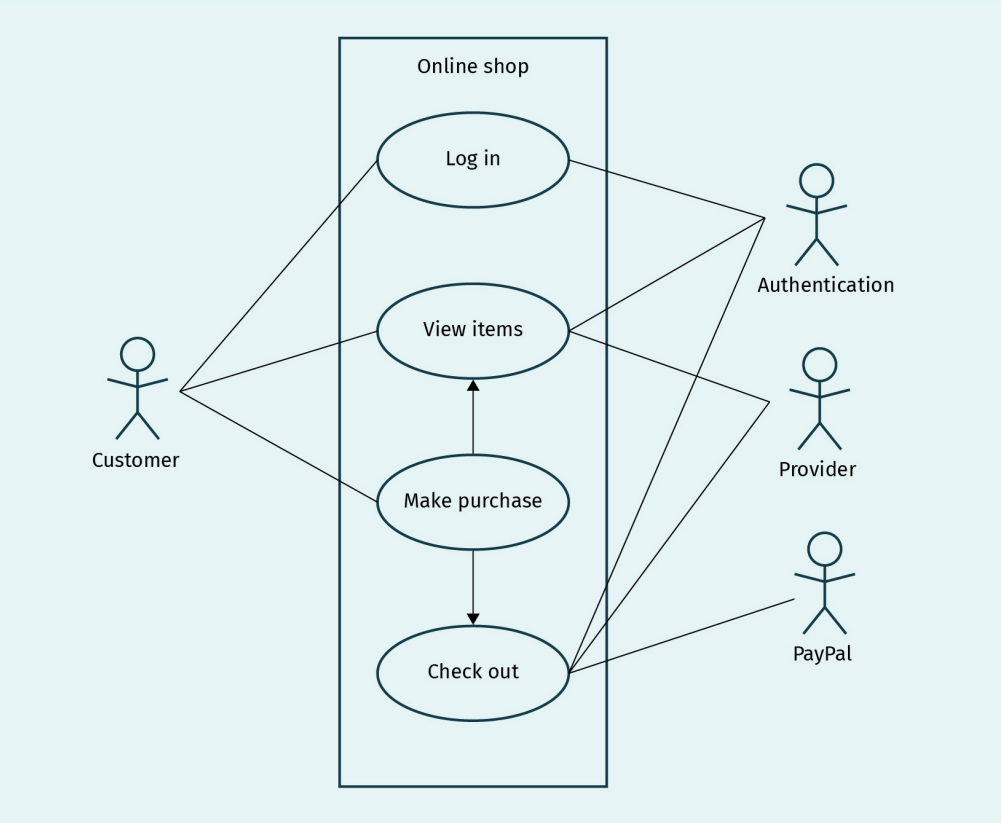
Among the relationships building blocks, we can mention the dependency relationship indicating the connection between the source and target elements, and the association relationship indicating how many elements are involved in that relationship.

Diagrams in UML are a collection of "things" and their relationships, which build a system. In UML 2.0, there are thirteen main diagrams, three of which we will address here (Booch et al., 2005):

1. The class diagram that represents the system classes and their relationships
2. The object class that instantiates a class diagram (i.e., show a concrete example of a general class)
3. Use case diagram that represents the function of a system from the perspective of an external user.

For example, a use case model from the requirement workflow can be visualized by using the UML use case diagram as shown in the following figure for an online shop. In this use case diagram, we have things, such as use cases (“log in,” “view items,” “make purchase,” “check out”) and actors (customer, PayPal), and the relations between these things (such as communication between “customer” and “view item”).

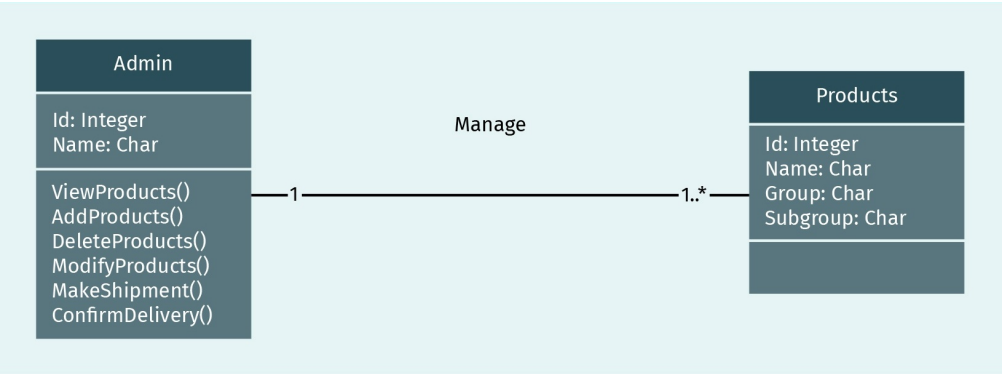
Figure 8: UML Use Case Diagram for an Online Shop



Source: Alavirad (2020).

Another example of a UML diagram for an online shop system is a class diagram for the products managed by an administrator. In this class diagram, we have two classes: “Admin” and “Products” classes with the “Manage” association between them. “1..*” on the “Product” class side means each Admin can Manage one or several Products and the “1” on the “Admin” class side means each Product should have one Admin. The “Admin” class has attributes such as ID (type: integer), Name (type: character), and operations, such as `ViewProducts()`and `AddProducts()`. The “Products” class also has different attributes (ID, Name.), but no operation. We can expand this class diagram to include more classes, such as “Customer” class, “Cart” class, and so on.

Figure 9: UML Class Diagram for an Online Shop



Source: Alavirad (2020).

In Wei & Field (2004), you will find a real case study with real-world experiences of how a development team in Ford Financial successfully developed and deployed an iterative methodology using RUP. The development team set out to resolve several obstacles by using the RUP. One of these obstacles was the difficulty with sharing project deliverables across projects as every individual team had its own set of templates and its own methodology. Another problem was the late involvement of service teams (i.e., the teams that are outside the application team and provide good products and services to projects), as well as the late identification of project risks. Finally, an additional challenge was posed by the fact that framework architects were overburdened by irrelevant questions, such as how to organize a project and what process to use (Wei & Field, 2004).

SUMMARY

In this unit, we learned about the traditional approaches to the management of the software development life cycle. Each project (software or hardware) starts with gathering the requirements from different individuals and organizations who are engaged in the project (the stakeholder) and analyzing them. Clients, developers, operation teams, and clients of the clients are all examples of project stakeholders. First, we defined software requirements and requirement engineering (an iterative set of actions for the elicitation, documentation, refinement, and changes of requirements). We also discussed different software requirement engineering approaches, such as pure linear and linear requirement engineering methods.

Secondly, we introduced one of the earliest and most straightforward approaches to the software development life cycle (SDLC) management models—the waterfall model. This model is based on five steps or phases: requirement gathering, system design, implementation, integration, and operation. In the waterfall model, each step starts when the previ-

ous step has been finished and approved. This model is more appropriate for small software projects with exact and non-changing requirements.

Then, we discussed another traditional approach to software development life cycle management, namely the rational unified process (RUP), an iterative approach mainly developed by IBM. RUP divides the SDLC into four separate phases (inception, elaboration, construction, and transition), during which, the main software development activities are performed, e.g., design and implementation. The changes and the risks in this model are more manageable compared to the waterfall model, as this is an iterative approach which considers changes in each iteration. Finally, we explained the unified modeling language (UML), an approach used to visualize the phases and activities in the RUP model.

UNIT 2

AGILE PROJECT MANAGEMENT

STUDY GOALS

On completion of this unit, you will have learned …

- the different definitions of Agile project management.
- the concept of Agile Kanban.
- the structure of the Agile Scrum method.
- what is important when transitioning from traditional to Agile project management.

2. AGILE PROJECT MANAGEMENT

Introduction

Traditional project management (TPM) models are suitable for managing projects with clear plans, requirements, goals, and a low degree of uncertainty in the project phases. The TPM-based model's primary focus is on the initial planning, prediction, and tracking of the defined phases. During the project development phases of this model, developers implement phases linearly and changes are mostly unacceptable (Coram & Bohner, 2005; Shenhar & Dvir, 2007). Furthermore, the project owner or customer is not involved in the project's development phases (Coram & Bohner, 2005). For example, car manufacturers can use a TPM model for developing cars on their assembly lines where processes are carried out in a sequential phase. TPM models can be used where changes do not come from the customer and the requirements are precise.

However, in areas like software development in which the use of technology is growing and change is inevitable, TPM-based models may not be very useful. In such scenarios, customer requirements can change during the project development phases, projects have a high uncertainty level, and there are no clear goals or plans (Shenhar & Dvir, 2007). Hence, we need a project management model that enables us to develop and manage projects under such circumstances, adjust customer requirements, and support customers to find a clear goal during the project development phases.

2.1 Agile Project Management

Agile project management
This is a model that follows a non-linear process, is flexible to changes, and focuses more on teamwork and collaboration instead of following linear phases.

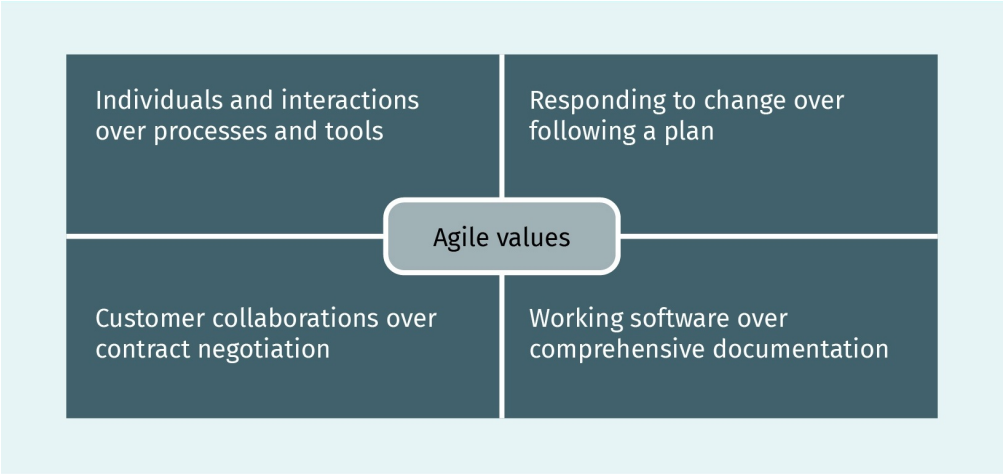
Agility is the ability to effectively and efficiently respond to changes during project development. To complete high-quality projects an organization can apply an **Agile** methodology to different phases of the project, e.g., to developer teams, departments, or shareholders involved in project management. Organizations or groups must consider the following three characteristics to make their system Agile (Sherehiy et al., 2007).

1. Flow. This relates to how the system works. When results are produced at a sustained and constant rate, the system is viewed as a high-rate flow that works smoothly and the team becomes more productive.
2. Learning. This refers to a strategy that allows the system to learn from previous experiences, mistakes, and other people's knowledge.
3. Collaboration. This refers to different strategies or cultures that identify how people involved in project development can work together to achieve the project's goals.

Agile project management has its origins in software development. In 2001, many software developers came together to discuss project management approaches in the software development area. For brand new software development, a TPM-based model like the waterfall method is not the best choice because of the high level of uncertainty, changes in user needs, and goals. The developers tried to create a lightweight and better

software development method called Agile. In the Agile Manifesto written by Fowler and Highsmith (2001) for this methodology, the developers introduced the following four values of Agile.

Figure 10: Agile Values



Source: Asadi (2020).

A software development team of people, i.e., programmers, testers, project managers, and customers, must work together effectively. The most crucial factor for developing software is people and how they interact or work together. For example, a team of skilled developers with good communication but without perfect management tools might work better compared to a junior team with ideal management tools. It is skills and communication that make a project successful, not only a perfect project management workflow.

Another important factor is customer collaborations, since the customer is the only person who can tell developers more about the project's target and about their needs and requests. As a contract is established after the first meeting with a customer, it is possible that in the early stages, they do not have the skills to precisely specify the goals of the project and can change their target when they get more information from developers. Having a close collaboration and educating customers along the way is more important for understanding their requirements during the development phase than only sharing the initial contract with them. For example, in a TPM or waterfall method, customers will have an initial meeting with the project managers to explain what they want to create and sign a contract with the company responsible for creating the project. Then, the project department or team has to deliver customer requirements. During the first meeting it could be the case that the target is not clear for the customer which means that additional communication with the development team will be needed after the initial contract to make the target clear and to explain the requirements. Considering such value will also help understand changes from the customer's side and update the development team so that it can successfully adjust to them.

It is also essential to consider that the customer prefers to have a working software rather than a heavy document describing the steps towards the achievement of the targets in a software development project. Still, having some guidelines in place is beneficial to the development of the final product (Fowler & Highsmith, 2001). In addition to outlining the four core values of Agile, the Agile Manifesto also introduces 12 principles designed to help project teams better understand what Agile software development is all about (Fowler & Highsmith, 2001). They are as follows:

1. The highest priority is to satisfy the customer through early and continuous delivery of valuable software. In software development, there is no need to plan everything in advance. The software is broken down into small functional elements called increments. Delivering these increments in the early stage of the development phase can help speed up the process of getting customer feedback and planting their request, thereby increasing customer satisfaction.
2. Changing requirements should always be welcomed, even late in the development. Agile processes harness change to guarantee the customer's competitive advantage. As previously explained, conditions will change rapidly, but quick responses to these changes can increase competitive power in the market.
3. Working software should be delivered frequently—between every two weeks and a few months—with a preference for a shorter time scale. Shorter development phases lead to a more frequent delivery of working functions, or increments, to the customers and helps developer teams receive immediate and informative feedback.
4. Business people and developers must work together daily for the duration of the project. A successful project needs regular involvement of the stakeholders. Agile provides companies with tools and techniques that enable stockholders to participate in the development phases of the project.
5. Projects should be built around motivated individuals. Give them the environment, support them, and trust their ability to get the job done. In an Agile team, it is necessary to have people who are willing to collaborate, share knowledge, and learn from each other.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation. A face-to-face meeting is a more efficient way of conveying information when collaborating and it also helps the team to understand the project's critical factors and be involved in all discussions during the development phase.
7. Working software is the primary measure of progress. Delivering working software should be the main goal of the software development team.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely. The customer or stockholder should then be able to maintain the final product.
9. Continuous attention to technical excellence and good design enhances agility. High-quality source codes can play a key role in the project's success.
10. Simplicity is essential. The tasks should be small enough to be doable.

11. The best architectures, requirements, and designs emerge from self-organizing teams. The team should have the skills needed for developing the final product. These will help in making independent decisions and organizing autonomously to achieve the project's goals.
12. At regular intervals, the team should reflect on how to become more effective and adjust its behavior accordingly. This will help an Agile team improve its project management workflow and avoid unnecessary tasks.

Now that we understand Agile project management's values and principles, we can further explore this new methodology's workflow.

Agile Workflow

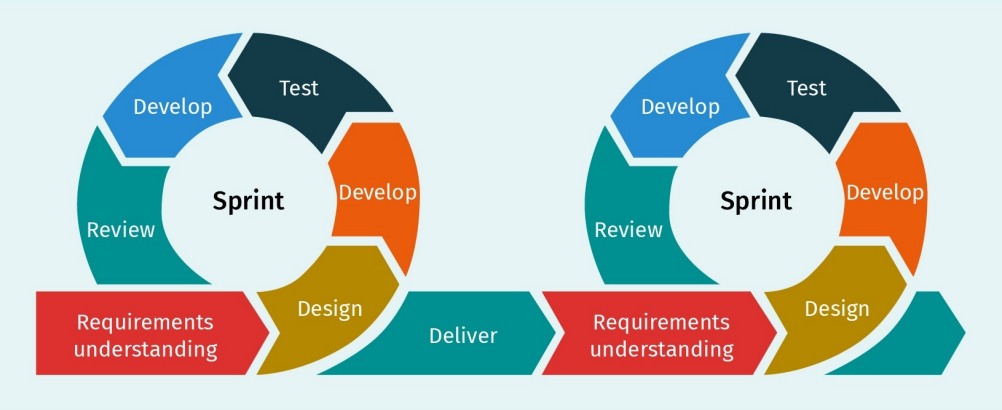
Agile is an iterative project management methodology that helps organizations or software development teams deliver their products and services to their customers quickly and efficiently (Highsmith, 2009). The Agile method is based on an iterative process in which the project is broken down into small functional increments known as user stories. Developer teams will focus and deliver user stories during a short development cycle called a **sprint**. Sprints are at the heart of the Agile methodology. When this methodology is implemented correctly, your Agile team can deliver better software with fewer headaches.

Sprint
This is a short period of time during which an agile team comes together to get a certain amount of work done.

Before explaining the Agile workflow, it is essential to clarify the rules and responsibility of people who are participating in an Agile project.

- User or customer. The Agile team has to understand customers' requirements and support them in solving their challenges. The customer has to keep communication with the Agile team open, while the Agile team has to give high priority to its customers.
- Product Owner. They are the voice of the customer or internal stakeholders in the Agile team. The role of Product Owner entails analyzing and summarizing ideas, gaining knowledge, and giving feedback on a product to the developer team. They are responsible for splitting customer requirements into detailed user stories. Each user story should carry information about who the customer is, the problem being faced, how it may have been solved before, why a new solution is essential for them, and the acceptance criteria at the end of the sprint. As part of an Agile team, the Product Owner will support the developer in creating acceptable products in a feasible and manageable way.
- Product or software development team. In the Agile methodology, a group of developers uses its advanced skills to complement a project and focuses mainly on delivering functional products based on customer requests. To achieve the values and cover the characteristics of Agile, the development team has to have frequent—even daily—internal meetings to assess the progress and struggles and to distribute responsibilities.

Figure 11: Agile Workflow



Source: Asadi (2020).

In the Agile workflow, illustrated in the figure above, each sprint starts with understanding the requirements of the customers and shaping them into user stories. Since the Agile methodology promotes a more supportive group culture, the developer team will design and develop the first solution in close collaboration. During each sprint, every developer will focus on user stories, develop their solution for it, and test it internally to get feedback from other developers. They then focus on the feedback they receive and use it to improve their skills and products. At the end of a sprint, the team will review user stories and deliver them to the customer, who can suggest more changes and give feedback about the products. If necessary, the developer team can consider the customer feedback or modifications in the next sprint.

The Agile project development workflow places primary focus on the process of software or project development and especially on the quality of the final product. To have a professional Agile environment, the Agile team should consider the following characteristics of Agile (Miller, 2001):

- Iterative. Agile focuses on customer requirements, which the developer team tries to satisfy during the current sprint and improve upon in the following ones.
- Modular. During the development phases of Agile, the Product Owner splits tasks into user stories.
- Time-boxing. The developer team completes each module in a sprint cycle, which usually lasts between one and six weeks.
- Parsimony. The Agile team should remove unnecessary tasks and activities to save time, control risks, and reach sprint targets.
- Adaptive. The team will adopt changes and address the new customer requirements before the next sprints.
- Collaborative. Agile culture helps people to collaborate and interact more with each other.
- Customer-oriented. The customer is the core and main focus of an Agile team. The group will increase customer satisfaction by getting customers involved and enabling them to actively take part in the processes, clearing and shaping the targets, and supporting them in achieving their targets.

Impact of Agile on Project Teams

Agile project management is a successful project management method that offers organizations many advantages and makes them more productive (Masson et al., 2007). However, its implementation in a team might be difficult because it is necessary for professional and skilled people to take on all roles and responsibilities in order to have a professional Agile team. Therefore, when moving towards Agile, some team members could be let go, while new ones can be hired to bring the required skills to the team. Besides having skilled people, having an Agile mindset in the organization is essential. The heart of this transmission is moving away from the traditional hierarchical structure and going towards a collective leadership mindset. In such organizations, leaders sees themselves as a part of the developer team and developers see themselves as self-organized. When considering Agile, the following restructuring will happen for a leader, development team, and the customer.

Figure 12: Traditional vs. Agile Team Structure

Source: Asadi (2020).

In fact, the structure of Agile affects the working culture and mindset of people in an organization (Gannod et al., 2018). In this new culture, the main focus is on the customer and their requirements. The developers are more collaborative and supportive and will try to be more productive and deliver a high-quality product. Developers try to create more value for customers during each sprint. The team should hold short feedback loops with customers to understand new changes and requirements. Such collaborations will increase customer satisfaction and let them be more involved in the development phase. In the last few years, different approaches have been introduced based on Agile concepts. Two very well-known methods are Scrum and Kanban, which we will explore in the next sections.

2.2 Introduction to Kanban

In the middle of the twentieth century, the Toyota production system introduced a visual method for managing their production lines' workflow (Sugimori et al., 1977). Before this method, the industry worked based on a **push system**. This kind of system brings additional costs and requires more time and resources because products are manufactured regardless of whether or not there is a demand for them, then have to stay in the show-

Push system
In a push system, the industry will produce products without customer demand for it.

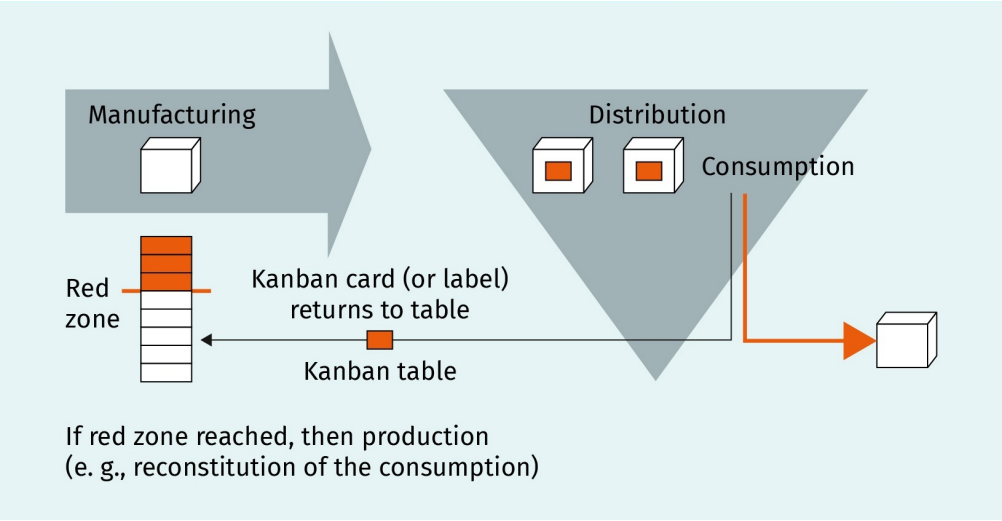
rooms until a customer requests one. For example, when a brand wants to create a new product design in the fashion industry, it will make many new products. Then, the sales and marketing department will start advertising and marketing their new product. If customers are not interested in the latest products, or if some external or unseen factors affect the expected sales, the organization will incur significant costs (Sugimori et al., 1977).

Pull system
This is based on customer demands for actual products. The industry will create a product when there is a demand for it.

Kanban helps to remove additional costs and struggles related to its project development by using the **pull system** (Sugimori et al., 1977). It means they only make products based on customer requests. In the production lines of Toyota, when a customer orders a car, the demand will come to the production line and the needed parts will be collected from suppliers. After finishing the product, the production line will satisfy the customer's request by sending them a new car. In recent years, Kanban has been adapted and applied to many industries. Another good example of a Kanban system is a restaurant: When you order food, your request goes to the kitchen where it will be prepared and, after a short amount of time, your order will be ready.

In the twenty-first century, the software industry realized that Kanban could offer great support when answering customer requests and could help increase productivity (Ahmad et al., 2013). Such a strategy also helps to reduce production costs for the organization and final product costs for customers. Kanban's visual system focuses on the tasks and shows what has to be done, when, and in what quantity. In the following figure, you can see the original workflow of the Kanban structure.

Figure 13: Kanban Structure



Source: Waldner (1992).

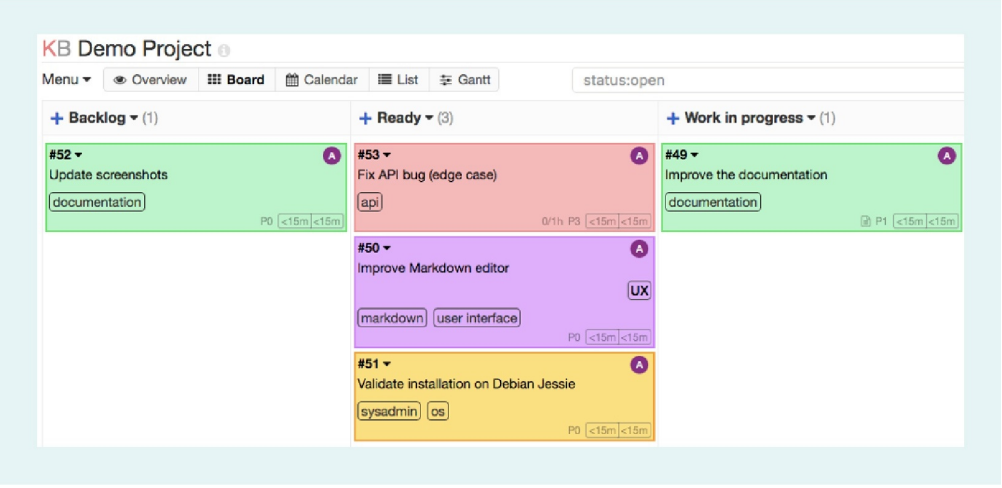
This structure has different components, i.e., the manufacturing or product team, distribution or the contact point with the customer, a Kanban card, and a Kanban table or board. According to this structure, a developer pulls a card from the red zone whenever there is a customer request. Whenever a request comes in, a developer will start a task in the production line. After the production is complete, distribution will deliver the product to the

customer and return the related card to the Kanban board. A Kanban board shows all the tasks the developer team is working on. It also offers a visual representation of the progress of each job by displaying the status of the tasks.

A team splits the Kanban board into different columns to show the current status of the task. Based on the team or project needs, the number of columns can change. The standard number of columns is three: “to do,” “in progress,” and “done,” which show general ideas, work in progress, and completed tasks, respectively. Each Kanban card shows a task and its status to the development team. In a Kanban board, a team can also be assigned different tasks, thereby visualizing the various projects or progress levels running in the team.

The Kanban board helps your team to visualize tasks and maintain transparency regarding the status of the project. It allows for quickly identifying bottlenecks as well as understanding and removing any obstacles or risks during the development phase of projects. Many teams use an offline (on-site) Kanban board, which is easy to understand and visualize, and is accessible for team members (Hammarberg & Sunden, 2014). This type of board is helpful when the team is sitting together all in one location. Nowadays, with the growth of the internet and technology, team members often work remotely and, therefore, need to use an online board that can be accessed from anywhere. Kanbanboard is a free open-source project management software that helps you to run Kanban to visualize your works, limit the works in progress, and drag and drop to manage your projects’ user stories with a simple installation (kanbanboard, n.d.). The figure below shows a demonstration of this online board.

Figure 14: Kanbanboard

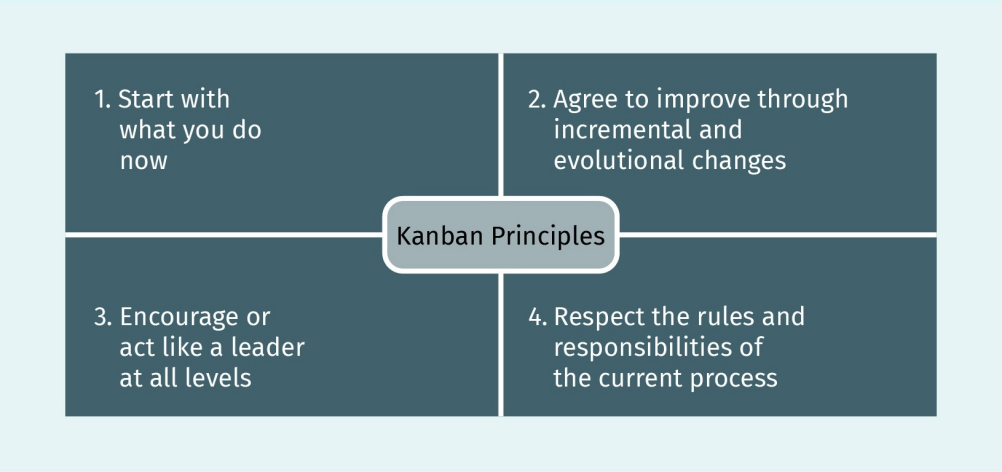


Source: kanbanboard (n.d.)

Kanban Board Principles and Practices

In the original concept, introduced by Toyota, Kanban had four principles and six practices that every team needed to consider when using this methodology in order to achieve successful project development (Sugimori et al., 1977). The four principles of Kanban are shown in the figure below.

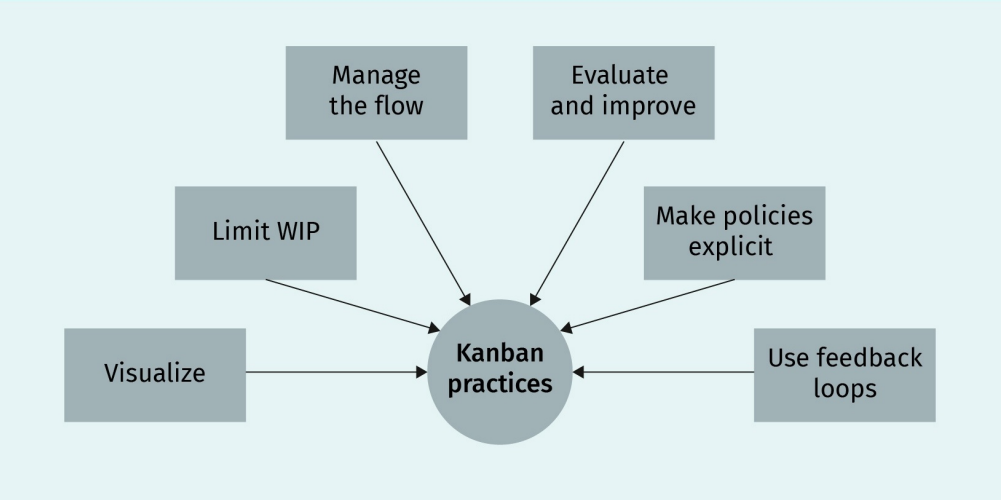
Figure 15: Kanban Principles



Source: Asadi (2020).

Since Kanban focuses on the tasks, team members must visualize their current task and explain what they are doing. The Kanban team can agree to split duties into small increments (also called user stories); they also have to be flexible towards any changes coming from customers. Kanban allows team members to select what they would like to work on and how they want to do it. It can give them freedom and help them to be more self-organized. If you want to reach a positive outcome, acting as a leader who encourages and supports their team members rather than simply managing them is absolutely essential. Furthermore, following the rules and feeling responsible for the current process is key to a thriving work culture. The six practices of Kanban are illustrated in the following figure (Ahmad et al., 2013).

Figure 16: Kanban Practices



Source: Asadi (2020).

Kanban is a visual system, which is useful when there's a need to visualize work, workflows, struggles, and risks. Another characteristic is that the team has to limit the total number of works that it has in progress based on its resources and capacities. While running Kanban, it is essential to ensure that tasks are moving in the right direction within the workflow. When a job is complete, a team member can review it before moving it to the "done" column and giving the developer feedback. Based on this feedback, developers can then improve the quality of work and skills. In order to create a smooth workflow, the rules and policies that the Kanban team creates should be explicit.

Agile Kanban and Its Advantages

A team can mix the Agile principles and practices with Kanban to obtain an Agile Kanban project management methodology. In this new method, the team can use Kanban to manage the process workflow and complete tasks. Each user story in Agile corresponds to a card on the Kanban board containing clear information about the job. Each team member could move the tasks from one column to another on the Kanban board after they have considered related policies. The Agile team has to limit the amount of work in progress. The iterative cycle of the Agile sprint can be applied to Kanban. For example, a software development team can agree that they will have a two-week sprint cycle. When a team member finishes a task before the end of the current sprint, they can select a new card from the Kanban board if they have the capacity to do so; they do not need to wait until the end of the current sprint to get a new task. The team can also add new requests from customers directly to the board as a new card.

The Agile Kanban methodology helps to deliver high-quality products quickly and efficiently. The collaboration inside the team can speed up the delivery of Agile Kanban. Agile Kanban has the following benefits for the project team (Lage Junior & Godinho Filho, 2010).

- It has a flexible approach towards accepting customer changes.
- It reduces wasted work, time, and costs.
- It helps to focus on the continuous delivery of customer requests and products.
- It increases the productivity of the team.
- It increases knowledge of team members.
- It brings about a better collaboration culture for the team.

A pull structure-based industry, where the production will only start after a customer request, can use Agile Kanban. For example, suppose you work on an automotive company's data science team where you have all data ready and accessible from the start. The technical department regularly asks your team questions on how to improve their product that need to be answered based on the data sources. Each of these questions will constitute a task on the Kanban board; once you have met all requirements and finished your current job, you can pull a new one from the "to do" column of the Kanban board, thereby answering all questions from the technical department one by one.

2.3 Introduction to Scrum

Scrum is a framework that helps the developer team to develop, deliver, and improve different types of projects. Takeuchi and Nonaka (1986) first introduced this model in the *Harvard Business Review*, a general management magazine covering various management and business-related topics. The Scrum authors published their article with the title "The new product development game," in which they used the term "Scrum" for the first time in the field of software development. Originally borrowed from rugby, the word "scrum" refers to a type of teamwork and collaboration in which the team tries to move the ball while simultaneously staying united (Takeuchi & Nonaka, 1986). The following figure shows the concept of scrum in rugby.

Figure 17: Scrum in Rugby

Source: Baucherel (2019).

Like a rugby team practice, Scrum as a project management method encourages teams to collaborate closely, learn from feedback, be self-organized, and express struggles and challenges in order to get support and improve products. The initial emphasis of Scrum for developing projects is in the software development area. During the last few years, different industries have also made attempts to adopt the Scrum methodology in project management (Sutherland, 2004).

Scrum is an iterative project management method for managing and supporting the team and enables it to develop a high-quality product in **increments**. This framework provides a structure, principles, and practices that a team can follow to gain knowledge, learn how to adapt work, and leverage product development experience (Sutherland, 2004). In order to satisfy customer needs, a Scrum-based team should know that the customer's mindset, requests, and goals might change over time. These unpredictable changes are not suited to prediction-based project management methods, e.g., the waterfall method. Instead, the Scrum team has to consider these changes and adapt to the new customers' requests. The Scrum team must also be aware of the project's challenges but cannot define the project's concrete targets upfront. Rather than focusing exclusively on delivery, this model also focuses on responding to changes and adapting to new technologies quickly and efficiently.

Increment
This is any work done as part of the current and previous sprints that meets the Definition of Done. The increment should provide value and it should not just be a list of tasks or features added to the products in the previous sprints.

Agile Scrum Framework

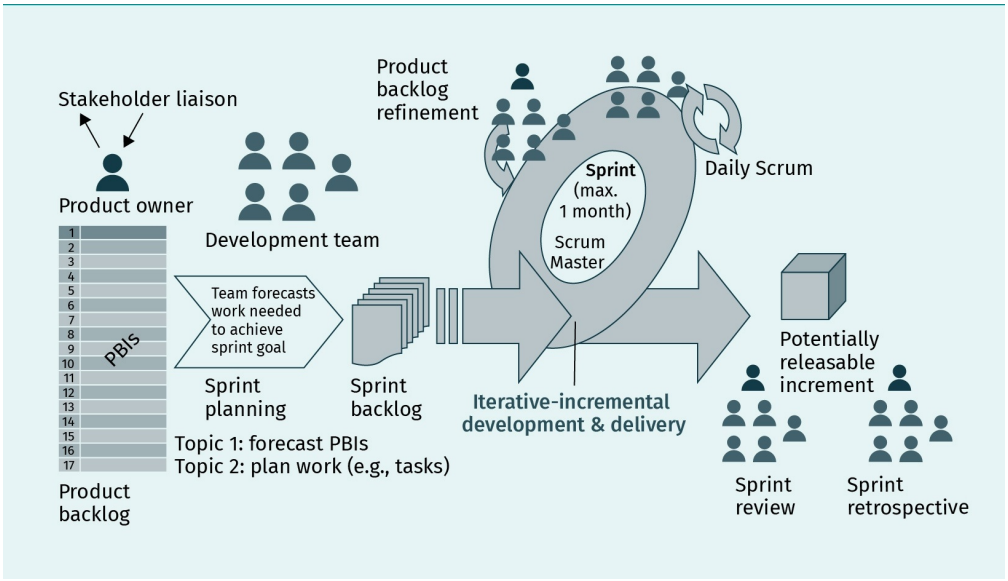
Agile Scrum
This is an iterative sprint-based project management method that supports the team in delivering a high-value product based on agile and Scrum rules and practices.

Based on this model's regular iteration, team members can focus on the tasks, organize themselves, and improve their delivery without interruption. During a sprint, the team needs to communicate and collaborate closely to support each other. This model is suitable for projects or companies that can break down their product development phases into small increments that team members can complete during a sprint. The **Agile Scrum** method aims to build valuable increments and deliver a functional product at the end of each sprint. To create a successful Agile Scrum team, it is essential to consider the following practices (Schwaber & Beedle, 2002):

- Transparency. The Agile Scrum team should create a culture and environment that enable members to share knowledge about the process, struggles, and obstacles of the project so that they are able to solve them together.
- Inspection. The Agile Scrum team needs to regularly assess or review the tasks to improve the final product's quality. All team members should agree to receive and give feedback.
- Adaptation. In an Agile Scrum team, team members should continuously check and revise items that do not add value to the project.

Agile Scrum consists of different elements, different rules, and responsibilities for each team member. You can see an effective Scrum structure in the figure below.

Figure 18: Agile Scrum Structure



Source: Created on behalf of IU (2022).

Agile Scrum Sprint

In Agile Scrum, the project development length is broken down into equal parts known as sprints. Each sprint is a time when the Scrum team plans and works together to deliver functional increments (Takeuchi & Nonaka, 1986). If the task is unknown or complex, it is better to have a shorter sprint. The team can plan a sprint's duration of one to four weeks but, if necessary, it can change the sprint's size. All events in an Agile Scrum team will happen during the sprint. A sprint can also become a project if the developer team delivers all parts necessary to run the projects within the sprint.

Consistently adjusting the size of the sprint will help in the estimation of the project costs. For example, in four-week-long sprints, the price for developing and delivering a project will be equivalent to the team members' monthly salary. Many teams try to use a fixed-length sprint as a best practice—usually not longer than one or two weeks. During the sprint, it is essential to finish all identified tasks. Therefore, planning the right sprint affects the efficiency and productivity of the team. For healthy sprint planning, you have to be able to answer the following questions:

- What is the objective of the sprint?
- How does the developer team achieve goals?
- Who will be doing and delivering a task?
- What is the input for developing a valuable outcome?
- What is the output of a task?

Agile Team Structure

An Agile Scrum team is a self-managing, cross-functional, and focused team that tries to achieve the project targets. In an Agile Scrum team, people can take on different roles based on their skills and experiences. The following roles are necessary for a successful Agile Scrum (Sims & Johnson, 2012):

- Product Owner. This is the same role as an Agile Product Owner. They represent the customers and stockholders when determining the product's vision for the development team. During the sprint planning, the Product Owner, with the team's agreement, identifies the objective of the sprint.
- Scrum Master. This role helps to facilitate Scrum and ensures the team follows Agile values and principles. Scrum Masters work with each Scrum team member, guiding and coaching them to ensure that the team will achieve the sprint goals. They are not team leaders, they ensure that the development team completes all the required tasks. They are also the owners of the processes and responsible for managing events. Since a Scrum team should be able to focus during a sprint, the Scrum Master is the point of contact of the team, together with the product owner.
- Developer. If you are passionate about product development and have skills that enable you to deliver high-quality products quickly, which makes customers happy, this would be a perfect role for you in an Agile Scrum team.

Agile Scrum Artifacts

Artifacts
These are pieces of information or tools that we make or use to solve a problem.

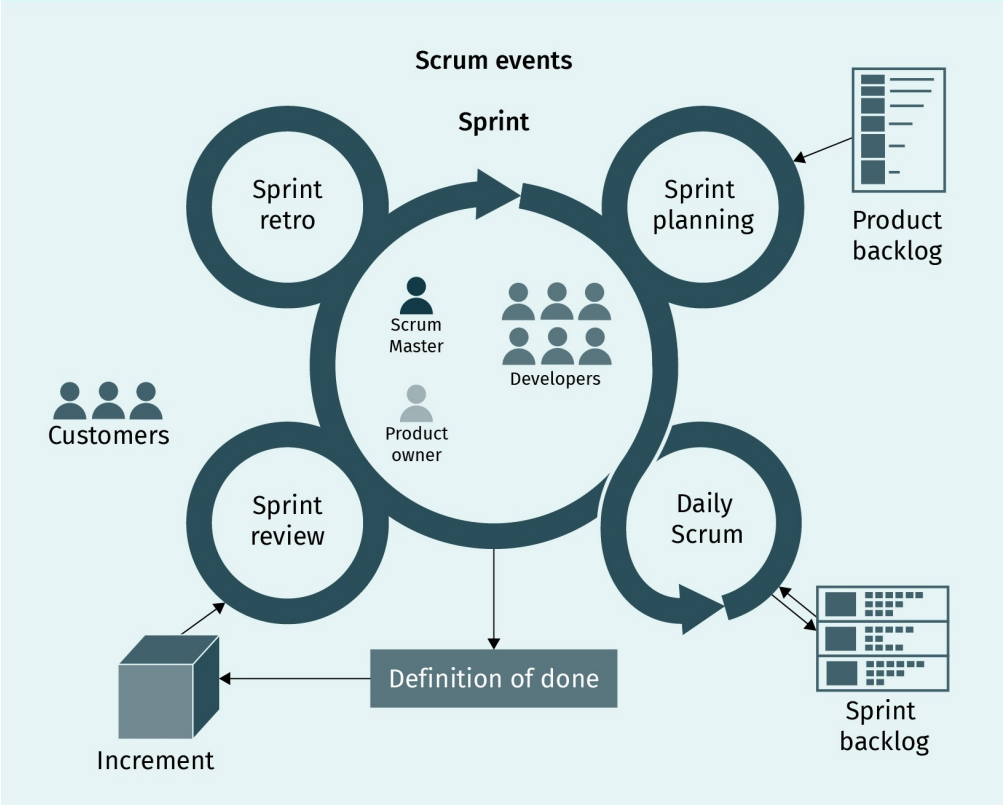
In an Agile Scrum, **artifacts** are the main part of the workflow. The team and customers use it to show detailed information about products and actions to perform in order to deliver the developed project (Jongerius et al., 2013). These artifacts are briefly outlined below:

- Product backlog. A product backlog is a broken-down master list of the tasks that the development team must complete and it is maintained by the Product Owner. The maintenance of the product backlog is the Product Owner's responsibility, as is the identification of each item's business value. The task is divided into user stories that are options for the development team to work on and can be selected for the next sprints. There is no guarantee that the development team will deliver all user stories created by the Product Owner. Therefore, the Product Owner focuses on the tasks which have a high business value and removes the ones with a low value. Each user story gives a full definition of the job, features, requirements, owner name, developer name, etc., which will be an input for the sprint backlog.
- Sprint backlog. A sprint backlog contains a list of items or user stories that the development team should work on in the current sprint. These items determine the sprint goal or what the team wants to achieve at the end of the sprint.
- Increment. The sprint goal is collecting tasks or product elements that can represent a reliable function or create a usable product from a sprint. It must meet the Definition of Done and be deliverable to the customer.
- Definition of Done. These are standards, roles, or acceptance criteria common to every user story in the sprint backlog. The development team created this definition and agreed with the Product Owner to accept items in the sprint backlog at the end of the sprint. For example, a software development team can deliver a new version of a product at the end of each sprint. Therefore, the Definition of Done is realizing a new version with a new functionality that has undergone a code review and a unit test, and that is immediately deployable on the customer side.

Agile Scrum Events

Scrum defines four events or ceremonies that must take place during an excellent sprint cycle (Jongerius et al., 2013). These help Scrum team members to improve communication and transparency. In the following figure, you can see these events and their relations to the other elements of Scrum.

Figure 19: Events in Agile Scrum



Source: Asadi (2020), based on Takacs (2018) .

Sprint planning is a sequential meeting or event that kicks off the start of a sprint, where the product owners and developer team identify the plan and tasks that will be carried out during the sprint (Boschetti et al., 2014). The Scrum Master leads this meeting, which usually lasts more than one hour. The team selects items from the product backlog and pushes them to the sprint backlog. These items are aligned with the goal of the sprint and agreed upon with the product owner. Every Scrum developer has to be clear about what they can deliver at the end of the sprint.

The daily Scrum or stand-up is a short daily meeting lasting about 15 minutes, taking place at a specific time (typically mornings) and which should be as quick as possible. Everyone in the team shares updates on what they have done and what they plan to do in the next 24 hours. Daily stand-ups help the team to be on the same page and remain aligned with the sprint goals.

At the end of a sprint, two meetings take place: the sprint review and the retrospective. In the sprint review, the entire team can view a demo of the product in an informal environment or review all the team's tasks completed in the current sprint. In this meeting, stockholders or customers involved in the project can get an update about their products, give feedback to the developers, and suggest additional changes to the products. The Product Owner can consider these new changes as new product backlog items for future sprints. They can also decide whether the team can realize the latest version or not. At the end of

the sprint review, the development team, including the Product Owner, comes together to talk about their performance during the sprint, what worked, and what did not (James & Walter, 2010). The retrospective is related to the project sprint, items, tools, people, relations, and all other things that affect the team's efficiency and productivity (James & Walter, 2010). It will also help the team to create a healthy environment and successful Scrum.

There are many tools and software which can support an organization in creating an Agile Scrum workflow. Jira Software is an Agile project management tool that helps the Agile team to run Scrum. This software offers an Agile board to report, plan, track, and manage all steps of your Agile project with a single tool. You can review this software on their official website and try to create an Agile Scrum for your project management workflow (Jira, n.d.).

Agile Scrum Benefits and Limitations

Scrum is a simple project management workflow with rules and events that are easy to use and understand. The conversion of complex tasks into manageable user stories makes this model suitable for complex tasks. This model is beneficial for industries with complicated project processes which care about customers and results (Srivastava et al., 2017). Quick realization and continuous delivery of the products increase the Scrum team's motivation and customers' satisfaction. Furthermore, this model adapts quickly to customers' changes, which helps the organization to remain competitive. In the Scrum method, there is no boss to tell the team what to do and when to do it, instead, the team is self-organized and members communicate often to support each other in finishing tasks. In Scrum, a software development team can also use pair programming: In this technique, two developers work together, one writing the code and the other one supporting and reviewing each line of the code at the same time.

In general, Scrum manages to bring commitment, courage, focus, and respect to an organization. Start-ups, such as WordPress, Dell, and Airbnb, have also implemented Agile Scrum into their environment (Kahootz, 2015; Poskitt, 2018). However, Scrum also has some limitations. For example, for teams that work remotely, part-time, with special skills, which have projects with many stakeholders, or which have many external dependencies, the Agile Scrum methodology is generally not the best approach to adopt (Srivastava et al., 2017). In the end, it is up to every individual team whether it wants to use Scrum or not; if team members see some limitations, they can also optimize it to suit their needs.

2.4 From Traditional to Agile

Many organizations are still using traditional methods for their project management, as they see the benefits in using these models. In today's world, technology is growing fast and customer requirements are changing quickly; therefore, it is necessary to focus on Agile-based models. The main focus of Agile is on customers and their requirements. During an Agile process, the project team tries to incorporate the customers' needs into the development process and supports customers to reach their targets. The following table compares traditional project management with Agile.

Table 3: Comparison of Agile and Traditional Project Management Methods

	Traditional project management	Agile project management
Requirements	Clear requirements with low change	Clear requirements with high changes
Customer	Not involved in the process	Close collaboration
Documentation	Formal documentation required	Implicit knowledge
Scale of project	Large-scale	Small and medium
Organization structure	Linear	Iterative
Model preference	Adaptation to changes	Anticipation of changes

Source: Asadi (2020).

Nowadays, many big and small companies, including start-ups, are using Agile to successfully develop their projects (Yau & Murphy, 2013). Suppose you want to set up your start-up and introduce your product to the market. After your start-up has been growing for a while, you have to find a suitable project management methodology. Agile can be a perfect fit for you in this case. You may need to combine different Agile methodologies to find the best solution for your team based on Agile values. This methodology enables you to improve flexibility, accelerate the time-to-market, improve product quality, improve collaboration, and respond quickly to customers' requests for changes. If you want to implement Agile successfully, you need to have a reason to change to Agile before you communicate it to your team members and make sure they are willing to support you through this transition. It is essential to demonstrate the importance of Agile and the method you want to work on, i.e., Scrum or Kanban. As your team is the primary driver of Agile, it is crucial to keep all members motivated. The team structure should be clear since Agile, unlike traditional methods, requires a clear team structure and identification of Agile roles and responsibilities. You need to get your team to agree to do Agile events and identify the tasks that need to be completed. Based on the model you adopt, you can select a proper tool specifically tailored to your needs. During a test run, you can find and optimize the Agile model for your team.

When it comes to managing complex projects, many organizations work based on the waterfall method, which ensures that the team will follow the planned development phases and reach the target on time and under the budget (Petersen et al., 2009). This model is especially well-suited to projects consisting of a series of dependent steps that start with a thorough plan and run the phases based on the plan. This model works particularly well with simple and straightforward projects, however, when a task is more complex and the target is not clear, it would be better to choose a different methodology. For example, if you want to create a website for an online business, you have to consider the fact that technology is growing quickly and that you do not know your users' requirements or what particular feature they like, which makes it difficult to plan in advance. Such digital projects contain a lot of unknown steps because, in this field, it is impossible to predict the project's final output and the project requirements. In these projects, you need to learn

from the test run in order to create a clear plan and product, and to gain experience. As in the waterfall method, testing is not part of the development process and there is no room to incorporate feedback which is why adding changes can result in additional costs. As we have already mentioned, Agile methods can perform better in a complex project, as they break down projects into user stories; however, there are some points to consider when transitioning from a waterfall method towards Agile (Carilli, 2013). They are as follows:

- Team capacity. Figure out your team capacity or working hours.
- Individual capacity. Planning each team member's workload individually helps you to better understand their potential and communicate changes to them based on their ability.
- Estimation. The proper assessment of the effort required to complete a task affects your team members' productivity.
- Sprint length. Find the right sprint length based on your needs.
- Client transition. Your clients need to participate more in the project development processes. You have to help them understand the break-down framework and collaborative processes, the value of embracing uncertainty, the need to have the willingness to give feedback, and flexibility towards time, cost, and efforts.

Transitioning to Agile means changing the work culture and collaboration style of your organization. It is essential to support culture changes and take Agile values into consideration. It is also important to remember that, in order to change the work culture, Agile will also change management and leading processes and measure the team's progress and achievements (Carilli, 2013). For example, in Scrum, the target is to finish all user stories in a sprint backlog. In order to achieve that, you have to find a way to measure team performance based on Agile values. During this transition, the organization needs to have experienced people taking on roles and responsibilities in the team. To do that, you can choose whether to hire a new member or train the team members to fill the need for specific skills.

A sudden move from traditional to Agile project management can be tough for your team, especially for team members who are not familiar with this model and who need to learn a whole new project management approach. Smoothly aligning your legacy project management method with an Agile framework requires skills and patience. You need to spend time and money bringing the right knowledge and model to your team. Furthermore, in addition to changing rules, events, and responsibilities, you also have to bring a new language and working culture to your organization. Agile requires agility, but it's worth the effort to try. The transition from waterfall to Agile is a lengthy process, but it will improve your team collaboration and work quality. However, bear in mind that Agile is a journey, not a destination, so in the end, you have to make sure the transformation does not hurt your team, but rather helps it to be more productive.

SUMMARY

This unit provides an overview of Agile-based project management approaches for developing high-quality products. We presented the essential characteristics of these models and covered how they can help organizations move towards successful project developments. We first talked about the main Agile values, principles, roles, and characteristics. This model brings a new working and collaboration culture between team members, customers, and stakeholders. It breaks down projects into doable tasks and tries to cover customer changes to increase customer satisfaction. Next, we explored Kanban as an Agile model that visually illustrates the tasks on a board to give quick information about the work status to the team members. We discussed the pull and push system and showed how Kanban, as a pull system, could support the organization to improve its project development experience.

We introduced Agile Scrum as another important Agile-based model mainly used in the software development field. This model is based on iterative development sprint processes. In Agile Scrum, tasks are distributed among three prominent roles: Product Owner, Scrum Master, and developer. These roles help teams and organizations to become more productive and focused on the customer's requirements and changes.

Finally, we assessed the advantages of Agile in comparison to traditional project management models. The Agile methodology benefits small and medium-sized projects and increases customer satisfaction thanks to its ability to implement customer change during the iterative development phase. We mentioned some features of Agile that need to be considered when first adopting this method or changing from a traditional model to Agile. The main change that Agile brings is cultural change. From the developers to the top managers, they all have to adopt Agile values to have a successful Agile experience. Every organization has to find a perfect model based on their requirements; there is no one-size-fits-all model for all projects in the project management field.

UNIT 3

TESTING

STUDY GOALS

On completion of this unit, you will have learned …

- why testing is useful in any software project.
- what unit and integration tests are and how you can use them in your projects.
- how software development teams approach testing.
- how machine learning systems can be tested effectively.
- how to continuously assess the quality of your code base.

3. TESTING

Introduction

Testing software is an integral part of the software development life cycle. What it means to test something can vary widely from having basic, automated sanity checks to complicated manual test procedures to see if the workflow of a user on a website works as intended by the programmers. For instance, a company building an e-commerce platform will spend enormous amounts of time and effort to ensure their payment system is very solid and cannot be exploited, and they will do so both by testing their fundamental payment functionality in isolation and by having test users check out actual items in the shopping cart. Testing isn't easy and, in this unit, we will explore the basic paradigms of software testing in a pragmatic fashion.

As a data scientist, you might not face the same problems as software developers, but any code that you touch that ultimately finds its way into the product of a company must be tested with the same scrutiny as any other component. For instance, when you have finished training a machine learning model on predicting the likelihood of users clicking on an ad on an e-commerce platform, it is not enough to simply trust that this model will function as expected. You need to have basic and advanced safeguards in place to make sure your model will perform in practice, too. In this example, if you continually overpredict the click-probability of users and sell ad space on your site proportional to this probability, then this prediction is linked to someone's ad spending and overpredicting means that they pay too much. In the worst case, your model will assign high value to an ad placement that is essentially worthless, eventually leading to your company losing advertisers' trust and potentially a lot of money. However, with proper tests in place, scenarios such as these can often be avoided completely, or at least be spotted before the issue becomes too large to effectively deal with.

Ideally, testing is a team effort supported by all parts of a company. Often you find dedicated testing teams working with and supporting the core development team. Automated software testing and implementing testing pipelines that only allow new code to be shipped to production environments when all tests pass have become a staple of modern software development. While there is no such thing as a perfectly tested piece of software, assessing how much of its functionality is covered by tests is worthwhile. Larger companies go as far as to create whole testing strategies and prioritizing testing according to the project's risk factors in an effort to mitigate them. There are many levels at which you can test software, and projects in general, and also many stakeholders with different interests. For instance, backend developers might have a vested interest in ensuring their application servers have few to no outages, while a frontend developer is more focused on having as little friction in the user flow as possible.

In this unit, we will investigate the value of testing from a data scientist's perspective, discuss how to write different types of tests in Python, how to approach testing in practice, and look into how machine learning systems can be tested properly.

3.1 Why Testing?

In the introduction to this unit, we talked about how ubiquitous testing is in software development and hinted at a few reasons for this, but it is good to take a step back and ask: Why are we testing software systems in the first place?

Validity and Correctness

In spite of our best intentions, humans make errors all the time. By extension, even the best programmers in the world err, and they do so quite frequently. We test software to make sure our programs execute and run correctly, even in edge cases. This problem becomes even more prominent when several programmers work on the same project and modify each other's code, which is the norm.

Let's have a look at a basic Python example that shows that correct behavior of a program needs to be properly defined and scoped out. Let's say you want to add two numbers and return the result to the user. A function with that functionality could look like this:

Code

```
def add(a, b):
   return a + b
```

Simple enough, but does this program run as intended? If we provide two numbers as input to this function it will indeed return their sum. However, Python has **operator overloading** and the plus operator "+" does not only work on pairs of numbers, but also on strings or lists, which allows you to misuse the add function above for other types:

Operator overloading Many programming languages allow you to overload operators, such as "+", "-", or "=", to other contexts and custom functionality, while others strictly forbid it. One of its benefits is that it makes for very readable code.

Code

```
add("test", "this")
# 'testthis'
add([4], [3, 2, 1])
# [4, 3, 2, 1]
```

What is worse is that you can actually break this function by providing mixed input types. For instance, if you input one integer and one string argument, you will get a Python `TypeError`:

Code

```
add("3", 42)
# TypeError: can only concatenate str (not "int") to str
```

While in this artificial example the consequences may not be too serious, you can easily imagine more complicated functions used in a large code base, consisting of thousands of lines of code, in which unintended usage can cause disaster. It's best to prevent this kind of problem altogether by introducing type checking and catching incorrect inputs.

In the concrete example of the add function, we could check that both input arguments to the function are either of float or int type in Python and make sure to throw an informative error if that's not the case. Python has the built-in assert keyword that you can use in such situations. A slightly improved version of the add function could look like this:

Code

```
def is_number(value):
   return isinstance(value, float) or isinstance(value, int)

def add(a, b):
   assert is_number(a) and is_number(b), "Both inputs must be numerical"
   return a + b
```

Now, if you try to use this function with lists or strings as input, executing the code will fail, as expected:

Code

```
add([4], [3,2,1])
# AssertionError: Both inputs must be numerical
```

Note that we improved our implementation by adding a basic safeguard. Doing so allowed us to better scope the input arguments, thereby making this function more testable. Still, the testing itself was manual in the sense that you have to execute the above code in a Python runtime and see if it returns the result you wanted. Automating this kind of procedure is what software testing is all about, and we will see more on this topic in the next section. For now, keep in mind that testing is used to ensure the following:

- Correctness. Does the program execute without any errors when provided valid input arguments and does it return the correct result in all cases? For instance, does the program correctly compute the sum of its arguments?
- Validity. Does the program make the right assumptions about the context and scope in which it is to be executed? For instance, do we restrict input arguments to the right types?

Requirements and Documentation

In the example of the above add function, it became clear that software functionality is a result of the requirements, i.e., the intended behavior we impose on it. We required the add function to only take in numerical values as input arguments, which is a design choice. It is conceivable that we could also have chosen otherwise and purposefully allowed add to take various other arguments.

Companies do **requirements engineering** to match their code with the expectations of their users and customers. Testing software can be one way of ensuring these expectations are met. In the add function example, we know from manually testing it that it only

works on numerical inputs. In fact, the error message that gets returned when this is not the case highlights this within the function body, i.e., the numerical input requirement is documented in the function itself.

Requirements engineering
The process of defining the specific functional needs for a piece of software, as agreed upon by a group of stakeholders, is called requirements engineering. It precedes the implementation of any code and is the basis of it.

The tests you write can more generally be understood as an expression of the requirements and documentation of intended behavior. Without explicitly testing "add" it would be unclear how to use it. When a new programmer joins a company and starts working on a new code base, it is considered good practice for them to check out and run the tests first.

Lower Support Costs and More Trust

When software is properly tested, a company will need to spend less time on support, as there are fewer bugs in the code base and the applications will run more reliably. In general, it is advisable to invest in good tests upfront rather than to let your users discover bugs for you. In an evolving system, it is virtually impossible to eradicate all sources of error entirely, but, with good **test coverage**, you minimize the risk of something failing fundamentally. It's usually unfeasible or impractical to aim for 100 percent test coverage, but a high percentage should be the goal. In the example of our add function, we've essentially covered 100 percent of the functionality, but, for more complex functions, it becomes increasingly difficult to do so.

Test coverage
In software testing, coverage is a measure of how much of the functionality in your code base is tested for correctness, i.e., "covered" by tests.

Ultimately, good testing is a requirement to stay in business. If your customers trust you to continually produce high-quality software with every release, there is a much higher chance that you can retain them. Reliability and quality are prerequisites for good user experience and a necessity to grow your customer base.

Development Frameworks and Best Practices

The trust your customers gain is reflected by the trust your developers have in their own software, internally. Having a good testing framework, accompanied by a set of engineering best practices, will help development teams to be more productive. For instance, junior staff members will find it easier to go through their onboarding process when they can rely on the benefits of a tested code base (e.g., by understanding its intended behavior faster).

By contrast, untested software is often difficult to deal with, in particular when you want to modify and extend it. This is because a lack of testing means that you can never be entirely sure if a modification breaks part of the existing functionality.

Refactoring

Code bases naturally evolve over time as customer requirements change, technology becomes obsolete or gets replaced, and new functionality gets added. This implies the necessity to change existing code, to **refactor** it. When you start to refactor a project, you need to be careful not to break things. Good tests can guide a developer in their refactoring efforts by giving vital feedback. If your project has good test coverage for a specific function and you rewrite this function from scratch to cater to an extended use case, after

Refactoring
In software engineering, we speak of refactoring as the practice of modifying or rewriting existing code in order to maintain or extend it.

you've finished the rewrite all tests should still pass, i.e., the previous functionality should still work. Without tests, it is largely impossible to keep large software projects alive, as they will need to be refactored constantly. Once a code base grows beyond what a single person (or group) can reliably monitor, automated tests are the only way to ensure modifications don't introduce new bugs.

3.2 Unit and Integration Tests

When talking about testing, it is useful to distinguish between different types of tests. One of the most common differentiations is between unit tests and integration tests.

Unit Tests

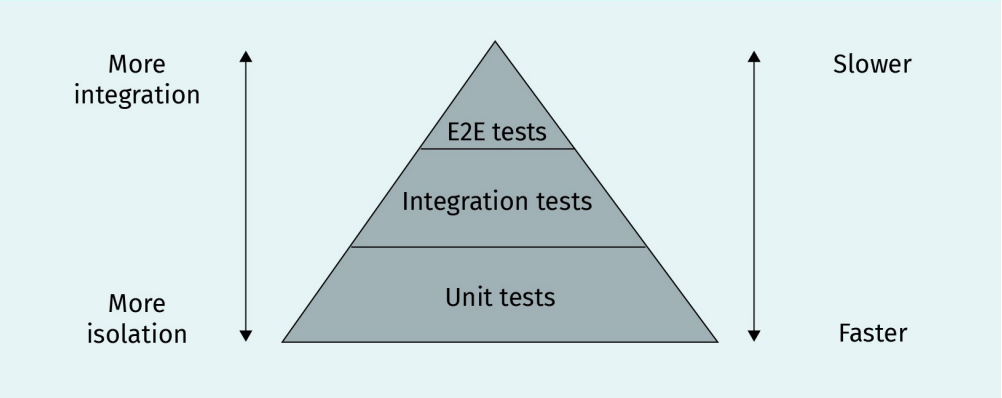
These tests check smaller, standalone code units in isolation. Unit tests will usually run very fast, so developers can execute them frequently, e.g., whenever saving a file they just modified. Unit tests give developers confidence of the validity of the "building blocks" of the code base, but do not test how components interact with each other in complex and intricate ways.

Integration Tests

In contrast, integration tests are aimed at testing how various components or modules interact with each other in the grand scheme of things. Integration tests will often run much slower, due to the added complexity, and developers cannot afford to run them as frequently as unit tests. Often, integration tests can be so resource intensive that developers cannot afford to run them on their local machines at all. However, integration tests are an integral component of testing, as they are less isolated than unit tests, and are more concerned with a higher level interaction that resembles the type of interaction users will face when using the final product.

While unit and integration tests make up the most relevant part of the testing tool chain for data scientists, they cannot cover everything that is needed, so there are other types of tests to fill these gaps. For instance, companies building complex graphical user interfaces (UI), for example, for a web application, often require manual testing of the workflow of UI components. This can be quite expensive, as human labor is much more expensive than running tests on computers. Tests that automatically check the complete workflow of a user in an application are aptly called end-to-end (E2E) tests and can be considered an extension of integration tests in the sense that they are even more integrated. The drawback is that E2E testing is even slower and more expensive (in terms of compute resources) than integration testing. To summarize what we have discussed so far, we can define a test automation pyramid as explained in Cohn (2009) in the following diagram.

Figure 20: Test Automation Pyramid



Source: Pumperla (2020), based on Humble & Farley (2015).

Unit and Integration Testing in Python

As data scientist you need to be able to test the software you write. Here, you will get a short introduction to the unit and integration test framework `pytest` available for the Python programming language. There are other tools to choose from, like `unittest` or `nose`, but `pytest` is a very popular choice and likely the easiest to pick up. To get started with `pytest` for your projects, you must install it first with a package manager, such as `pip`:

Code

```
pip install pytest
```

In this example, we are creating a file called `arithmetic.py` and putting the latest version of the `add` function we defined in the last section into it. So far, we have only manually tested this functionality, but it's now time to automate the process by writing tests into a new file called `test_arithmetic.py`. Let's start with a simple correctness check:

Code

```
import pytest
from arithmetic import add

def test_add():
   assert add(2, 3) == 5
```

If you run this test file with

Code

```
pytest test_arithmetic.py
```

the output on your system should look as follows:

Code

```
collected 1 item
test_arithmetic.py .                                [100%]
```

In this case, `pytest` was able to collect one test and run it without failure, leading to 100 percent correctly executed tests. This is good news, as it enables us to automate all the tests we ran manually before. For instance, remember that we didn't want to allow mixed-type inputs and non-numerical types as input to the `add` function. With testing, we can check, and simultaneously document, the intended behavior of our little arithmetic library. Here's how to check for incorrect types:

Code

```
def test_mixed_types_error():
   with pytest.raises(AssertionError):
       add("3", 42)

def test_list_type_error():
   with pytest.raises(AssertionError):
       add([4], [3,2,1])
```

Note how we catch the assertion errors emitted by the incorrectly used `add` function calls in both new tests with `pytest.raises`, which takes the type of error to catch as argument. When you run `pytest` again, after adding these two tests into `test_arithmetic.py`, you will now see three passing tests.

An interesting feature of `pytest` is that you can easily parametrize tests with it, i.e., instead of writing essentially the same test over and over again, but each time passing different values into it, you can provide a list of input parameters in a **decorator**. A parametrized version of our first `test_add` test would look like this:

Decorator
In Python, decorators are used to provide existing functions with additional functionality, like wrapping a single test case into a parametrized version.

Code

```
@pytest.mark.parametrize(
   ("a", "b", "expected"), [(1, 2, 3), (2, 3, 5), (42, 7, 49)]
)
def test_add_params(a, b, expected):
   assert add(a, b) == expected
```

Adding this parametrized test to our test suite and running `pytest` again will result in a total of six successfully executed tests, as we provide three sets of parameters to our test case. Specifically, what `pytest` does here is

- uses a Python decorator (indicated by the "@" sign) from the `pytest` library called `pytest.mark.parametrize`;
- specifies parameters strings in a Python tuple, here the three parameters "a," "b," and "expected"; and
- provides a list of parameter values that will successively be passed as arguments into the test function, whose input parameters have the same name as those specified in the decorator.

Up until this point, we have only scratched the surface of what the testing framework `pytest` can do. For parametrization, we have already seen markers, which are important for many different aspects in `pytest`. For instance, custom markers allow you to build a suite of tests that can distinguish between unit and integration tests. To do so, let's define a file called `pytest.ini` that we put next to our library and test code files, with the following content:

Code

```
[pytest]
markers =
   unit: fast-running unit tests
   integration: slower integration tests
```

This defines two custom markers, namely "unit" and "integration." The crucial part of this definition are the names of the markers. What follows after the colon (":") is just optional documentation. When you type `pytest --markers` into your shell you will see your new, custom markers prompted on top of the list of built-in markers from `pytest`:

Code

```
@pytest.mark.unit: fast-running unit tests
@pytest.mark.integration: slower integration tests
…
```

Using these test markers is now straightforward, you simply annotate your tests with the above decorators. For instance, if we wanted to make our very first test a unit test, we would just have to mark it accordingly:

Code

```
@pytest.mark.unit
def test_add():
   assert add(2, 3) == 5
```

Running only unit tests and running all non-unit tests has the following syntax, where the "-m" flag stands for module:

Code

```
pytest -m "unit" # run the single test marked as unit test
pytest -m "not unit" # run the other five tests not marked as unit test
```

Likewise, you could mark a test as a unit test by using the `@pytest.mark.integration` decorator and run the respective tests with

Code

```
pytest –m “integration”
```

This covers the very basics of `pytest` as a unit and integration test framework. It is worth noting that `pytest` is modular and comes with many interesting extensions, such as `pytest-cov` for analysing test coverage.

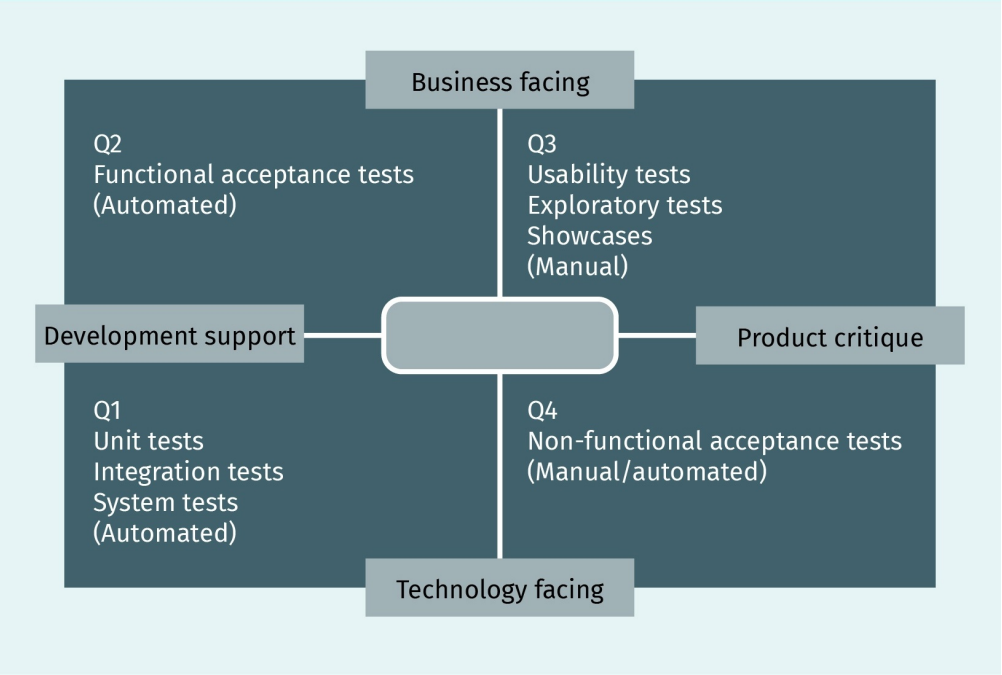
Other Testing Dimensions

We introduced unit and integration tests by essentially distinguishing them using two dimensions, namely how integrated or isolated they are, and how fast or slow they execute. There are other interesting dimensions relevant for implementing successful software testing strategies, among which are the business-technology and the development-product dimensions. They are defined as follows:

1. Business-technology dimension. Is the test in question coming from a business requirement or is it purely a test of the underlying technology stack?
2. Development-product dimension. Is the test designed to aid the development team or is it built to check the workflow of the product?

The axes of these two dimensions span across four quadrants known as Brian Marick’s quadrant (Humble & Farley, 2015). Not every product team needs to put equal emphasis on all quadrants, but checking each of them before a new release is worthwhile.

Figure 21: Marick's Quadrant



Source: Pumperla (2020), based on Humble & Farley (2015).

Technology-facing tests that support development (Q1)

It is no coincidence that these are precisely the type of tests we have already looked at: automated unit and integration tests. To many developers and data scientists who write these tests, these are the most relevant tests in their daily work. Note that the system we test, also called the system under test (SUT), might have external dependencies, such as third-party application programming interfaces (APIs) or data sources. In systems testing you will usually **mock** the external dependencies and provide test data for them to test your internal system exclusively. Systems testing tests the system as a whole, while integration and unit tests apply to parts of the system.

Mocking a system
To isolate and test components with external dependencies, providing an implementation that emulates the external system on a basic level, is called mocking.

It is important to note that all of these tests are functional tests that describe precisely what is supposed to happen in the software according to the requirements. Opposed to that, non-functional requirements are concerned with how things do what they are intended to do. For instance, a bottle's functional requirement is the ability to hold fluids and the amount of fluid to hold, while its design is entirely non-functional.

Business-facing tests that support development (Q2)

Tests in this quadrant are usually referred to as functional acceptance tests. In an acceptance test you predefine criteria that have to be met so that the test can be "accepted" by the business stakeholders of the project. An acceptance test that is also functional by the above definition is a functional acceptance test. As these tests are business-facing, the corresponding acceptance criteria can easily be formulated in plain English.

E2E tests, as we previously introduced, are an example of tests in this quadrant, as they are functional by nature and test the whole system, thereby covering high-level business-facing requirements. In E2E tests the goal is to simulate real user scenarios (Katalon, 2020). E2E testing differs from systems testing in that the latter mocks the environment, while the former tests all connection to external systems like APIs as well.

Business-facing tests that critique the product (Q3)

These tests are mostly manual and check that the product is shipped in the way customers expect it to be. They not only check for the underlying functionality, but also take into consideration non-functional aspects, such as usability. For instance, a test in this quadrant might consist of a demo of showcase that demonstrates a concrete case from a **user journey**, by stepping through the product. Also, tests in this quadrant can often be of an exploratory nature, trying to surface bugs that only occur in rare edge cases and other similar situations. If an exploratory test led to a new insight and is considered to have long-term value, said test might make it into the automated test suite in Q2.

User journey
In software quality control, the term "user journey" means the experience a user has when using a product.

Quite often the manual part of the testing in this quadrant and in quadrant 4 are done by dedicated teams of testers whose job it is to design and run complex test scenarios and report the result back to the stakeholders (both technical and business).

Technology-facing tests that critique the product (Q4)

The tests in this quadrant are acceptance tests, but are non-functional in nature. For instance, such tests might check for reliability, availability, or scalability of a service. Testing such requirements is very important, as downtime of a service or very long response times of a website can lead to customer dissatisfaction or lower sales. Non-functional acceptance tests can be automated but are usually only executed at the very end of the testing pipeline, usually after having made sure that all tests from Q1 pass.

3.3 Approaching Testing

Now that we know some of the basics of testing in Python, it is worth investigating how to approach the topic of testing in practice. While some companies take a very structured approach to testing with whole teams dedicated to the various aspects of testing as introduced in Marick's quadrant, others take a more pragmatic approach and test as much or as little as they deem necessary. Among the more structured approaches, we will discuss three software development and testing techniques in more detail, namely

- test-driven development (TDD),
- acceptance-test-driven development (ATDD), and
- behavior-driven development (BDD).

Test-Driven Development (TDD)

In test-driven development the basic idea is that you start by writing the tests first, and then provide an implementation that makes those tests pass. To inexperienced developers this technique often comes as unnatural, as you are required to devote all your initial time writing tests for which there is nothing that they could be checked against. In our running example of the add function, applying the TDD paradigm would mean to write the suite of six tests first in order to specify and document the requirements for the yet to be implemented add function.

One of the benefits of this technique is that it is much harder for developers to deviate from the originally imposed requirements, which are directly expressed as tests. This additionally ensures that developers are familiar with business requirements in the first place, a useful safeguard. On the other hand, TDD can sometimes lead to confusion on the part of the development team, as members can find it difficult to figure out where to start, how to set up a test, how much to test in an iteration, or how exactly to provide the first implementation, respectively figuring out why a test fails (North, 2020).

Once the test suite is written and all stakeholders agree on it, it is the responsibility of the development team to provide an implementation that satisfies these tests. Only once all the tests, reflecting the requirements, finally pass is the implementation considered valid. According to Beck (2014), the TDD process can be expressed in the following steps:

1. Write tests for functionality that has not been written yet. By construction, this test will fail.
2. Write the simplest code possible that makes the tests pass. Your code should not anticipate future requirements and thereby complicate the design. Making the predefined tests pass is what matters.
3. Run all tests to check that they do, in fact, pass.
4. An optional last step is to do some refactoring and clean up your code base. This might not be necessary in the first iteration, but once you go back and define your requirements and write more tests, it is likely that there are some redundant code parts or code that can be refactored to be more elegant.

As indicated in the last step, this TDD process is actually a life cycle, in the sense that the above steps, once completed, are repeated in a new cycle. While initially setting up a project with TDD might appear unintuitive as you don't have any functionality yet, just tests. Once you have an established code base and regularly enter the above cycle of steps, the development in TDD often doesn't look too different from a code-first and test-later approach, and development teams tend to get used to this style of programming quite quickly.

Acceptance-Test-Driven Development (ATDD)

TDD usually refers to a specification of unit and integration tests, and hence take a very developer-centric and functional view of the requirements held by a company. By contrast, in acceptance-test-driven development (ATDD) you go through the same four life cycle steps as in TDD, but you start with an active discussion between stakeholders, often

involving and focusing on the needs of clients, which results in a set of acceptance tests to be implemented. The ATDD methodology is, therefore, more business-facing than TDD and mostly corresponds to testing in Q2 of Marick's quadrant (as acceptance tests are usually functional in nature), while TDD falls into Q1, which is why both approaches complement each other.

Behavior-Driven Development (BDD)

Behavior-driven development (BDD), introduced by Dan North (2006), is another development paradigm related to TDD and ATDD, but with a clear focus on putting user behavior first. In BDD, new functionality is specified by examining concrete examples of users' behavior and examples are formulated in scenarios which follow a simple "given-when-then" structure. This means "given" an initial context, "when" an event occurs, "then" a certain outcome is expected. One of the main advantages of BDD is that its requirements are expressed in plain English, which helps bridge potential communicational gaps between software developers and business units and makes sure all stakeholders have a common understanding of the expected functionality of the software they are building.

Domain-specific language (DSL)
In software development, a DSL is a language that has been designed and crafted to work with the abstractions common in a specific application domain.

Gherkin is a **domain-specific language (DSL)** implementing the semantics of the "given-when-then" structure of modelling behavior (Gherkin, n.d.). Sticking with our simple example of adding two numbers, a Gherkin feature file would look as follows:

Code
```
Feature: Addition
   Functionality for adding objects
   Scenario: "+" should add two numbers
       Given the number 5
       When I add 3 to it
       Then the total should be 8
```

In Gherkin, you define a feature by giving it a name (in this example, `Addition`) and an optional description, and then move on to define one or several scenarios, which follow the aforementioned "given-when-then" logic. If you have multiple clauses, you can chain them with an `And` qualifier. For instance, the above Gherkin feature file, which we store as `addition.feature`, can functionally equivalently be written like this:

Code
```
Feature: Addition
   Functionality for adding objects
   Scenario: "+" should add two numbers
       Given the number 5
       And the number 3
       When I add them
       Then the total should be 8
```

Of course, this example is extremely simplistic, especially since we only defined it for two concrete numbers. Gherkin only really shines when describing more complex scenarios, but this example gives you a good enough idea of how this DSL is built. If we wanted to implement a BDD test in Python, we could leverage the BDD extension of `pytest` called `pytest-bdd`, which you install with

Code
```
pip install pytest-bdd
```

Having installed the library and given the above feature file, we can now define a BDD test for the `add` function (which would not have been written at the point of specifying the test):

Code
```
from pytest_bdd import scenario, given, when, then
from arithmetic import add

@scenario('addition.feature', '“+” should add two numbers')
def test_add():
   pass

@given("the number 5")
def first_number():
   return 5

@when("I add 3 to it")
def add_to_it():
   return add(first_number(), 3)

@then("the total should be 8")
def result():
   assert add_to_it() == 8
```

Note that this test will only work if you specify the correct feature file and choose the exact same naming conventions in the `scenario, given, when,`and `then` decorators provided by `pytest-bdd` as in the respective fields of the feature file.

3.4 Testing Machine Learning Software

Machine learning (ML) is a crucial part of any data scientist's work, which, on a high level, requires the selection of suitable algorithms and data to train these algorithms. The training process itself, i.e., letting the machine learning algorithm learn from the data, then produces a trained model that somehow has to be put into production. Once the model is live in a production environment, it needs to be continuously monitored to ensure it does

what it is intended to do. All of these steps need thorough testing to ensure the final product delivers the value expected. This section is focused on the testing aspects involved in machine learning, but is not an introduction to machine learning itself.

To give an example of the need for testing in machine learning, imagine you are working for an online marketing company that wants to give users of an e-commerce platform smart product recommendations using machine learning to yield more customer satisfaction. To do this well, you have to test several steps and assure high quality of each. These steps are as follows:

- Data collection and processing. Data collection describes the process of retrieving and storing data in the first place, while data processing entails transforming the data suitably so that it fits the given use case. For instance, to train a recommendation model, if your data set does not include the dates of previous sales, your model cannot account for seasonal effects like Christmas sales.
- Model training. Once you make sure you have the right data and have selected an algorithm, you will have to ensure that the training procedure is running smoothly and produces good recommendations.
- Model monitoring. After successfully training your model and putting it in production, you need to ensure that it performs well on live data, e.g., when a new user visits the e-commerce site you are working on for the first time, do they get good recommendations from the start?

These are just examples, but they give a good first impression of what testing for machine learning can look like. In the rest of this section, we will introduce you to both testing mechanisms for model training and deployment.

Testing Machine Learning Training Processes

From a software engineering perspective, one of the aspects that makes machine learning paradigmatically different from traditional programming is that in ML, models are not explicitly programmed by humans, but rather learned from data. Hence, there is a fundamental difference in the output of both approaches, and an ML model can be somewhat of a black box. Of course, any ML project also still contains code in the traditional sense, namely the code to set up the ML model in the first place, prior to letting it learn.

What's more, ML models are often nondeterministic, meaning that given the same inputs you might not get a single, deterministic answer all the time, but rather a range of possibilities that have to be interpreted. As an example, an ML model that has learned to classify pictures as having either dogs or cats in them and that is shown a picture of a previously unseen dog will assign a probability to that picture corresponding to the likelihood that this picture does indeed have a dog in it, according to the model. If the model predicts a very high probability, human operators might be confident in this result, but what can you expect for an example picture that has 51 percent probability of having a dog in it? That's essentially a coin flip and should be investigated more closely.

It is thus necessary that testers of a machine learning training procedure test several aspects, such as the data, the code to set up the ML model, the ML framework that is responsible for the training (usually a third-party library), and the choice of algorithm. We formally define a learning program to be the algorithm chosen for the use case, plus the code used to implement it in a given ML framework and all the configuration needed to set it up. Essentially, it's all that you need to train a model, minus the data.

Testing these aspects can be very time-consuming and require a lot of domain-specific knowledge (Zhang et al., 2019). While testing the code falls under the realm of traditional software testing, testing the ML framework is considered the responsibility of the respective framework vendor.

Testing data

As indicated in the introduction to this section, testing the data required for an ML model training procedure is not to be taken lightly and requires testers to check for several key questions, which we list with examples from the above cat-and-dog classifier:

1. Is there enough training data available for training? To train an image classifier, you likely need several thousands of images, for instance.
2. Does the data need to be cleaned or preprocessed? For example, images might come in different resolutions and aspect ratios, and they first have to be **normalized** to "fit" into the same ML model.
3. Does the model fairly represent future data? For example, if our data set contains only one breed of dogs, the trained model might not be very accurate for predicting other types of dogs.
4. Does the data contain a lot of noise? For instance, are the objects we are interested in (cats and dogs) partially occluded by other objects?

Data normalization
In machine learning and related fields, data normalization describes the process of transforming potentially heterogeneous data into the same shape or form for further processing by an ML model.

It is not possible to give a generally valid prescription of how to test data in ML model training, but there are a few best practices that you can follow (Heck, 2020). They are as follows:

- Schema and type validation. Whenever possible, impose a strict schema on data and check for data types. Relational database systems do this for you, for instance. Doing to helps ensuring good data quality and prevents mal-formatted inputs (Kim et al., 2018).
- Checking constraints and sanity checks. Oftentimes you can check for patterns and constraints in subsets of your data and how they relate to other subsets (Kim et al., 2018). For instance, all purchased items on an e-commerce platform must previously have been in the shopping cart, but not all cart items will lead to a purchase. Quick sanity checks can help immensely in spotting that something is wrong in your dataset.
- Using testing platforms. There are cloud service providers that help you check whether you are properly testing your data quality. For instance, Google has a data validation system that generates synthetic test data for a given schema, so that you can test your models with it (Breck et al., 2019).

Testing the learning program

When training an ML model, you need to evaluate its performance, keeping several metrics in mind (Zhang et al., 2019). These metrics are as follows:

- Correctness. The model should be as correct as possible given a predefined measure to test it against. For instance, in the example of classifying cat and dog pictures, you might be interested in accuracy, defined as the number of correct predictions divided by the number of all predictions, usually reported as a percentage.
- Interpretability. As mentioned before, ML models can often have aspects of a black box, but there are efforts to make the reasoning of these models more understandable and explicit. For instance, the ML Interpretability module available for H2O aims to give users feedback on the causes underpinning predictions (H2O, n.d.).
- Fairness. Ethical aspects such as fairness of predictions, e.g., not to discriminate against certain demographics, become more and more important in ML (Dwork et al., 2012). For instance, when trying to predict human attributes or behavior, a bias in data can lead to unwanted outcomes. One example of a tool used to mitigate discrimination and bias is the AI Fairness 360 framework (IBM, 2020).

Machine Learning Model Monitoring

Once your ML model makes it into production, it should be constantly monitored to prevent unwanted behavior. Though there are many others, we distinguish between three basic types of monitoring in ML systems (Gade, 2019). They are as follows:

1. Input data monitoring. You need to make sure that the data feeding into your model is stable over time by tracking its statistics. For instance, if you observe that the "onsite duration" of the users on your e-commerce platform goes down significantly, that might not only lead to a shift in your model's performance, but also hints at a bug in your website.
2. Model operations monitoring. You have to keep track of factors, such as latency, response time, and throughput of your models. If the predictions take too long, your ML service might slow down the entire operation.
3. Model performance monitoring. Do your predictions remain of high quality, or can you observe a drop-off, e.g., in accuracy over time? If this is the case, you might have to go back and retrain your model.

3.5 Performance Monitoring

So far in this chapter, we have discussed how testing of software components is done on an individual level, meaning what aspects you as a data scientist have to look out for. When working in a company that has to ensure its products are stable and well-tested, you need to make sure that all the tests that have been set up by individual contributors like yourself are properly automated. This is done by setting up a development pipeline with the following components, each of which we will discuss in more detail:

- continuously contributing and automatically building new code to the team's code base and delivering this code to production environments on a regular basis
- continuously testing the changes committed to the code base
- applying the first two points to an ML workflow, continuously training your models

Continuous Integration

Continuous integration (CI) is a software development paradigm that was introduced in 1999 which enables developers to integrate new code into existing code bases as quickly as possible in order to iterate faster (Beck, 1999). One of the core ideas of CI is that it is easier to spot software quality problems when making small, frequent changes as opposed to large, infrequent changes. Continuous integration is based on the following building blocks (Humble & Farley, 2015):

- Version control. Your software, including code, configuration files, scripts, and other files, needs to be checked into a version control system like the ubiquitous Git (Git, n.d.).
- Automated builds. For cases in which your programming language has a compilation step or other required build steps, there should be tooling to automatically rebuild your code when it changes. Modern integrated development environments (IDEs) can take care of this for you.
- Team agreement. Continuous integration is a philosophy based on teamwork, which requires commitment from each team member. For instance, all team members need to agree to use the same version control system and use automated builds in their toolchain.

Continuous Delivery

While CI is focused on developers and ensuring that they can quickly introduce new code, continuous delivery (CD) is a method for continually delivering software in a timely manner, so that companies can deliver value to their customers more quickly (Chen, 2015). We often speak of CI/CD together, as parts of continuous integration are also found in CD and the two methodologies naturally fit into common frameworks. To implement a CD pipeline, the development team of a company has to set up the following development and testing steps (Chen, 2015):

1. Committing code. Providing new code, e.g., with a version control software like Git introduced in the CI subsection, is the first step in the CD pipeline. Upon committing code, the unit tests of the project can be run, which is not always the case in this step and might only be done in the next. As with each step in this pipeline, if an error occurs, the pipeline stops and gives the developers feedback about what went wrong. Once the issue is fixed, the pipeline can proceed to the next step.
2. Building and testing the software. After building the software (if necessary), all unit and integration tests are run. Only if all tests are passed can the pipeline move into the next phase. This step will usually also include various reports, such as code coverage, formatting hints or code smells, which automatically tell the developers that some parts of the code base can be improved in the future. A build step will generally

produce some artifacts, e.g., in Java, compiling your project's source code will produce a JAR with compiled classes that can be used in production. These artifacts are then uploaded to a repository for deployment in later steps.

3. Running acceptance tests. Once the software has been built and the artifacts are uploaded, an environment emulating the production environment is built, i.e., setting up servers, installing the software, and properly configuring the service. This environment is often called the staging environment, preceding production. Once that is done, acceptance tests can be run to ensure that the software is also ready from a user's perspective.
4. Running performance tests. This step checks for non-functional requirements, such as latency and throughput of requests, and generally makes sure the new update does not degrade performance of the service.
5. Shipping to production. Finally, the last step is to ship all artifacts to the production environment and start a new service with your software updates.

Figure 22: Continuous Delivery Pipeline

Source: Pumperla (2020).

Continuous Testing

A topic related to CI/CD is that of continuous testing (CTe). As we have discussed before, there are several kinds of tests involved in a testing pipeline, e.g., unit, integration, and acceptance tests (Sakolick, 2020). In CTe we make sure that automated testing is an integrated part of the CI/CD pipeline we just introduced. For instance, we indicated that unit tests are both part of a regular CI setup and belong to the first step of a CD pipeline. More complicated tests that require a production-like server setup, such as integration, acceptance, and performance tests, should be part of later steps of the CD pipeline, following the build step.

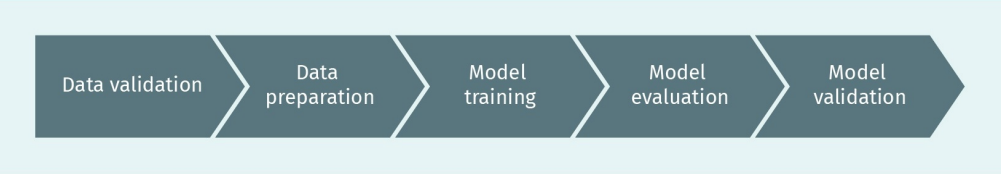
Another important type of test is regression testing, in which you compare the test results of the currently deployed solution against that of the intended update (Basu, 2015). If the new update would lead to worse performance compared to the production environment, we would speak of a regression in performance and reject the update in the CD pipeline.

Continuous Training

The last aspect of automated and continuous testing is that of continuous training (CT), which refers to automating the training and deployment steps of an ML project. According to Google Cloud (Google, n.d.), the following steps need to be executed to build a CT pipeline from ML projects:

- Data validation. This step checks if the data follow the expected schema and have been cleaned.
- Data preparation. In this step, the data are pre-processed to fit the requirements of the ML algorithm that the data will be fed into.
- Model training. After the data have been checked and prepared, the model training step is automatically triggered.
- Model evaluation. After the training procedure is finished, the model is passed some hold-out test data to compute performance metrics on them, like accuracy. Only if the model passes a certain threshold does it go into the next phase.
- Model validation. An additional model validation step might then be required to make sure that the trained model passes standards for production deployment, which might entail a manual testing procedure to gain confidence in prediction results.

Figure 23: Continuous Training Pipeline



Source: Created on behalf of IU (2022).

Additionally, the pipeline should store and manage metadata for each pipeline run, such as data and model versions, when the pipeline got triggered and when the process ended, and what parameters have been used to execute it.

A new CT pipeline run could be triggered for various reasons, e.g., when a human operator triggers a new run manually, following a predefined schedule, when new data becomes available, or when the model's performance regresses. A CT pipeline should have the following characteristics (Google, n.d.):

- Rapid experimentation. To ensure a fast ML experimentation cycle, the five steps of the CT pipeline should be executed quickly and seamlessly.
- Training on production data. It should be guaranteed that the ML model is trained automatically in production using the latest data available to avoid staleness.
- Experimental-operational symmetry. The exact same pipeline should be used for locally training models, e.g., on a data scientist's laptop and in the production environment. This ensures that local experimentation and production training will not deviate and will produce comparable results.
- Engineering best practices. The components of an ML pipeline should follow software development industry standards, such as reusability, modularity, and composability.

SUMMARY

This unit introduced you to the basics of software testing for data scientists. First, we investigated the many reasons why thorough testing is worth the effort it requires, including gaining and retaining trust from both customers and your development team, checking basic correctness and validity of your code, and the ability to safely refactor code. Secondly, you learned about unit and integration tests and how they relate. We explored basic testing in Python with the pytest library to show you how to automate testing for your own work and introduced Marick's four test quadrants as a more general classification of testing in business contexts.

The third section introduced you to the topic of how to approach testing in practice. We looked into the paradigms of test-driven development (TDD), acceptance-test-driven development (ATDD), and behavior-driven development (BDD) and gave examples of them in Python. Then, you were confronted with the crucial problems in testing machine learning software, including the necessity of cleaning and normalizing data and testing the learning program.

Finally, we considered production requirements for testing, in particular, setting up automated build and test environments such as CI/CD pipelines, aspects of continuous testing and what it means to do continuous training for ML experiments.

UNIT 4

SOFTWARE DEVELOPMENT PARADIGMS

STUDY GOALS

On completion of this unit, you will have learned …

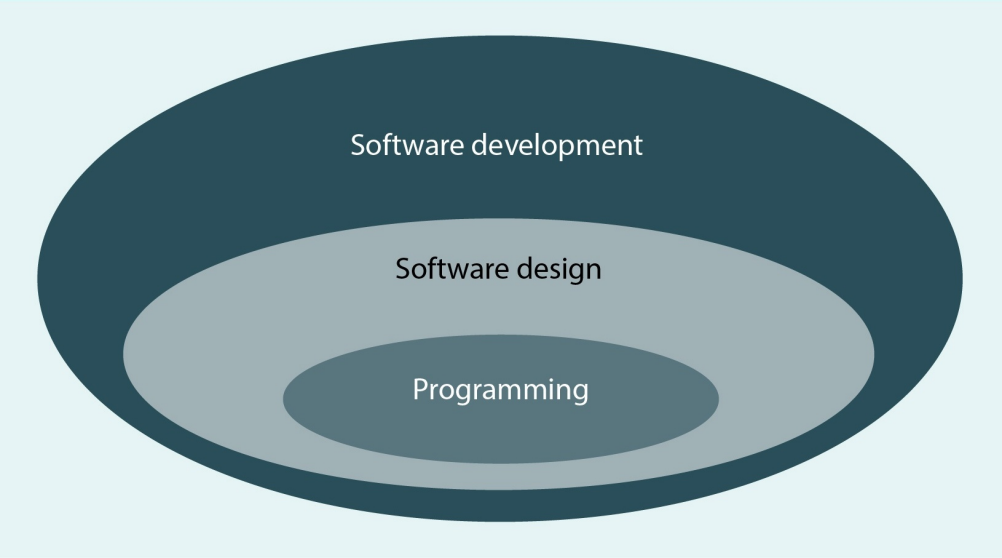
- the idea behind the software development paradigm.
- the difference between imperative and declarative programming.
- the major programming paradigms.
- different methodologies of software design.
- the definitions and concepts of pair, mob, and extreme programming.

4. SOFTWARE DEVELOPMENT PARADIGMS

Introduction

Software development paradigms refer to the procedures and steps that are taken during software design and development. There are several approaches suggested and currently in operation, so we are going to discuss these paradigms in software engineering. These can be combined into different groups, although they would likely overlap. Roughly speaking, programming is a subset of software design, which is itself a subset of software development, as illustrated in the following diagram (Tutorialspoint, n.d.).

Figure 24: Software Development



Source: Kobdani (2020).

In this unit, we will first discuss the most important programming paradigms. Then, we will take a closer look at software design paradigms. Finally, we will explore a few well-known software development frameworks.

4.1 Programming Paradigms

A programming paradigm describes the basic style in which a program is designed. It defines which principles are applied and which approaches are used (Guigova, 2009). There are several kinds of programming paradigms; however, there are four major paradigms: imperative, declarative, object-oriented (which is considered a subset of the imperative paradigm), and functional (which is considered a subset of the declarative paradigm).

Most programming languages are able to work simultaneously with different programming paradigms. Usually, it is not mandatory for programmers to commit to one programming paradigm and stick to it throughout the code. It can be shown that if a problem can be solved using one paradigm, it is also possible to solve it using the others. However, some types of problems lend themselves better to particular paradigms (UCF, n.d.). In this section, we will discuss some of the major programming paradigms.

Imperative versus Declarative

If we simplify the situation so that there are only two different ways to write source code, then we end up with an imperative and a declarative approach. These two approaches are, in a way, based on different philosophical views, which differ in ways that we will address below.

Imperative programming

The point of this is to give the machine a precise sequence of events through which to achieve a desired result. In other words, you tell the compiler what should happen step by step. For example, the following code is an imperative Python code used to calculate the sum of an array.

Code

```
my_list = [1,2,3]
sum = 0
for x in my_list:
   sum += x
print(sum)
```

Examples of imperative programming languages are C, C++, Java, Kotlin, PHP, Python, and Ruby.

Declarative programming

Here, it is a matter of conveying to the machine what one wants to achieve and to let the computer find out how this can be done best. In other words, you write code that describes what you want, but not necessarily how to get it done. The declarative form of the imperative example above for calculating the sum of an array can be as follows:

Code

```
my_list = [1,2,3]
print(reduce(lambda x,y: x+y, my_list))
```

Examples of declarative programming languages are SQL, regular expressions, Prolog, OWL, SPARQL, and XSLT.

Procedural Programming

The first programming paradigm that a new developer usually learns is procedural programming. Procedural code essentially contains the logical steps that explicitly instruct a computer on how to complete a task. A hierarchical top-down approach is used by this model and considers data and procedures as two distinct entities. Procedural programming breaks the program into procedures (also known as functions or routines), which comprise a sequence of steps to be executed. Simply put, procedural programming means writing down a list of instructions to tell the machine what it should do to complete the task step by step. The main features of procedural programming are briefly outlined below (Bhatia, 2020):

Subroutine
This is a sequence of program instructions that performs a specific task, packaged as a unit.

- Predefined function. This is an instruction or **subroutine** which is identified by a name. Predefined functions are included in the programming language. Usually, a predefined function belongs to a library. Two examples of predefined functions in Python are `float()` (which returns a floating point number), and `format()` (which formats a specified value).
- Local variable. This is a variable that is declared in a method's main structure and limited to the local scope assigned to it. The local variable may only be used in the system in which it is specified, and the code will stop functioning if it is used outside of the defined method. In the following Python example, "x" is a local variable.

Code

```
def func1():
   x = "I am a local variable"
   print(x)
func1()
```

- Global variable. This is a variable that is declared above every other code-defined function. For this reason, unlike local variables, global variables can be used in all functions. In the following Python example, "y" is a global variable.

Code

```
y = "I am a global variable"
def func2():
   print(y)
func2()
```

- Passing parameters. This is a procedure that is used to pass parameters to functions, subroutines, and processes. The passing of parameters can be achieved by "pass by value," "pass by reference," "pass by result," "pass by value-result," and "pass by name." Here is an example of parameter passing in Python.

Code

```
def sum(m,n):
   sum = m + n
   return sum
print(sum(5, 10))
```

Examples of procedural programming languages are C, C++, Lisp, PHP, and Python.

Object-Oriented Programming

When creating software applications, it is important to put the different building blocks of the software into related groups. In procedural programming, we can achieve this by creating functions. Object-oriented programming (OOP) allows us to do this by a single definition: **class**. Everything we need to execute the code correctly is defined in the classes (oodesign, n.d.). An object is an instance of a class. When a program is executed, the object is created based on its class definition, and it behaves as defined by the class. The properties of an object represent its state, while its methods represent all the actions it can perform. The main principles of object-oriented programming are briefly outlined below (Rouse, 2020):

Class
In object-oriented programming, this is a blueprint or prototype that defines the variables and the functions common to all objects of a certain kind.

- Encapsulation. This means that the execution and state of each entity are privately kept inside a given boundary (i.e., a class). Other objects do not have access to the content of the class, but can only call up a list of public functions or methods. This data-hiding characteristic provides greater program security and prevents the accidental corruption of data.
- Abstraction. According to this principle, objects disclose only internal mechanisms that are important to the use of other objects, covering any unnecessary code for implementation. This idea allows developers to make improvements and enhancements more easily over time.
- Inheritance. It is possible to allocate relationships and subclasses between objects, enabling developers to reuse a common logic while still retaining a specific hierarchy. This property forces a more rigorous analysis of data, reduces development time, and ensures a higher accuracy.
- Polymorphism. Depending on the context, objects may assume more than one type. For any execution of that entity, the program can decide which purpose or use is required, thereby minimizing the need to repeat code.

The following example shows a simple class and object definition in Python.

Code

```
class MyClass:
 x = "I am a class property, my name is X"
 y = "I am a class property, my name is Y"
myObject = MyClass()
print(myObject.x)
print(myObject.y)
```

Examples of object-oriented programming languages are Java, JavaScript, Python, C++, C#, Ruby, Scala, and PHP.

Functional Programming

In imperative programming, a program consists of a sequence of instructions that usually change variables (Sebesta, 1996). In functional programming, programs consist exclusively of **functions**. At first, there are often many strange technical terms in the context of functional programming, but its basics are actually easy to understand.

Function
This is a block of organized, reusable code that is used to perform a single, related action.

In general, functional procedures can be reduced to a few core principles. Some programming languages (e.g., Haskell) implement these principles directly in the language, others (e.g., JavaScript) require the programmer to take care of their compliance. The two most important principles for functional programming are as follows:

- Functions have inputs and outputs, but no side effects; they do not process any data that has not been explicitly passed to them (they are pure functions).
- Data ("variables") are never really changed, but only serve as a static starting point for generating other static data (known as **immutability**) (Rollins, 2018).

Immutable
Also called an unchangeable object, this is an object whose state cannot be modified after it is created.

Examples of functional programming languages are Lisp, Haskell, Scala, Erlang, and Clojure, but other languages, such as Python, R, and JavaScript, also allow you to write parts of programs in a functional style. Even in Java, functional programming has found its place with Lambda expressions and the stream API, which were introduced in Java8. The following example shows the usage of "list" and "map" in functional programming. It converts the elements of an array to uppercase and prints them.

Code

```
cities = ['berlin', 'london', 'paris']
upperedCities = list(map(str.upper, cities))
print(upperedCities)
```

Aspect-Oriented Programming

Aspect-oriented programming (AOP) is a programming paradigm for object-oriented programming, which allows you to use generic functionalities over several classes. These generic functionalities are called cross-cutting concerns (Spring, n.d.).

The first question that we are addressing is: What is aspect-oriented programming? To answer this question, consider the places in the programming of an application where you need recurring functionalities that are not part of the actual program logic. A classic example of this is **logging**, e.g., logging errors and other data. This is integrated in many places and intersperses the code with "unnecessary" code that does not directly belong to the functionality, leading to a worse readability of the actual application. Such requirements are called cross-cutting concerns because they cut through the core functionality. In contrast, the modules of the core functionality are called core concerns. Further examples of such requirements, which do not affect the actual core functionality, are security, authentication, tracing, and profiling. The modular implementation of such cross-cutting con-

Log
This is a detailed list of application information, system performance, or user activities.

cerns is the core idea of aspect-oriented programming. These cross-cutting modules are called aspects and are added at runtime. There are a number of AOP implementations, but some of the more common ones are AspectJ, Spring AOP, AspectC++, and Aspect#.

Rule-Based Programming

Rule-based programming is a programming paradigm based on mathematical logic. In contrast to imperative programming, a rule-based program does not consist of a sequence of statements, but of a set of axioms, which are to be understood here as a pure collection of facts or assumptions. If the user of a rule-based program asks a question, the machine tries to calculate the solution statement from the axioms alone.

For this purpose, a set of rules and statements constructed according to the syntax are inserted into the program code together with the information needed for the solution. Rule-based programming languages belong to the declarative programming languages and have their origins in the research field of artificial intelligence.

An imperative program describes exactly how and in which order a problem is to be solved. In contrast to this, a rule-based program ideally describes only what is valid. The "how" is already given by the solution method, which is derived from the existing rules. In other words, we model the program as a set of constraints that the system should satisfy and the inference machine tries to create a process satisfying those constraints. Perhaps the most well-known rule-based programming language is Prolog.

Symbolic Programming

Symbolic programming is a model of programming in which the program may manipulate its own formulas and components as if they were plain data. Complex processes that construct other more complicated processes by combining smaller units of logic or functionality can be built by symbolic programming. Thus, such programs can alter themselves efficiently and tend to learn, making them more suitable for different kinds of applications, such as artificial intelligence, expert systems, natural language processing, and computer games.

Generally, complex data structures are easy to build in symbolic programming languages. You can generate them just by writing them and then expand on them by performing simple operations. If you have ever created a linked list in C, you know it takes a lot of function calls to do so. In fact, the appeal of symbolic programming for AI is mainly the ease of constructing and manipulating complex data structures in Lisp or Prolog, just to name a few examples (Covington, 2010).

Event-Driven Programming

Event-driven programming is a software paradigm in which the interaction of components is controlled by events. Events can be triggered both externally (e.g., by user input or sensor values) and by the system itself (e.g., change notifications). An event can be a trigger for event handling, with which the system responds. An event-driven approach has little control over when data is processed. Event-driven programming relies on an event loop

that is constantly listening to the new events arriving. The operation of event-driven software is based on events. After entering an event loop, the events decide whether to execute and in what order. A simple example is the graphical user interface, where the user determines when and which data are processed by performing actions that trigger events.

The use of this paradigm requires that all systems involved in the processing of the event are able to communicate with each other during planning and development. Typically, event-driven programming requires a definition of what is considered an event. For this, computer systems or sensors monitor the status of objects and can trigger an event if necessary. This is followed by the processing of the event according to defined rules and the consequence of the event itself. If it is not possible to immediately achieve the defined sequence of the event during event processing, the event is temporarily stored in the achieved status and only continued when the sequence can be achieved.

Event-driven programming can be the most beneficial if it is already taken into account in the planning phase. In fact, rewriting an already existing software to an event-driven equivalent can lead to unreasonable effort due to the lack of the required interfaces.

4.2 Program Design

Software design paradigms are a critical element of program design. A paradigm is basically a way of thinking, and, generally, a software design paradigm tries to introduce solutions for the design, programming, and maintenance of software systems. It is possible to use design paradigms either to define a design solution or as an approach to problem solving (Stephens, 2015). Using a design paradigm, problem solving happens through abstraction of the problem. This approach is similar to the use of language metaphors; metaphors are used to help clarify new or foreign concepts and to bridge between a problem we understand and a problem we don't.

A software design paradigm in software engineering is a general, reusable approach to a frequently occurring problem within a given context in software design. It is not a completed specification that can be translated into source or computer code directly. Rather, in several different situations, it is a summary or blueprint for solving a problem. In this sense, design patterns can formalize the best practices that can be used for designing an application or system. An overview of the major design and development paradigms in software engineering is given in this section.

Domain-Driven Design

Domain-driven design (DDD) describes procedures that are intended to make complex software projects more transparent for all those involved. At the same time, it defines a series of techniques and elements with which an optimized domain model is to be achieved. The term itself expresses what is special about domain-driven design: The design should be based on the **domain**. In other words, the architecture and implementation are

Domain
This is the targeted subject area of a computer program.

consistently oriented to the domain. Since most software is implemented specifically to support domain-oriented processes, alignment with the domain is an obvious way to support these processes even further (Evans, 2010).

Usually, people from a wide variety of backgrounds will come together while working on a project. To keep communication as free from misunderstandings as possible, a uniform standard for describing all components of the project is crucial—a ubiquitous language. This language consists of all the terms that domain experts use when they talk about the domain. In fact, experience shows that projects tend to develop their very own language. The terms from the language should then also be used in the code and in the database to name fields, classes, columns, or tables. This makes it easier to implement the technical terms in the software because the domain-oriented terms do not have to be translated into other terms that are used in the technical implementation. In domain-driven design, each domain can be built from the following components (Evans, 2010):

- Modules. Domain-specific components of the domain
- Entities. Objects with mutable or ambiguous properties defined by their unique identity (e.g., people)
- Value objects. Objects uniquely defined by their properties and typically immutable
- Associations. Relationships between objects in the model
- Aggregates. Unit of objects and their relationships
- Service objects. Domain-relevant functionalities important for several objects of the domain
- Domain events. Domain-relevant events registered by special objects and made visible to other parts of the domain (e.g., sending events in one aggregate to other aggregates in the domain)
- Factories. Different generation patterns (mostly factory or builder patterns) used for complex scenarios
- Repositories. Clean separation of domain and data layer for abstraction of the system

Data-Oriented Design

Data-oriented design is a design paradigm for optimizing programs by carefully considering the memory layout of data structures for better performance. In other words, it means designing architectures with a focus on data representation, specifically with an emphasis on efficient memory layout and access (Fabian, 2018). The principles of data-oriented design are listed here (Sharvit, 2020):

- Separate code from data.
- Model entities with generic **data structures**.
- Data are immutable.
- Data are comparable by value (i.e., data collections are considered to be equal if they represent the same collection of values).
- Data have a literal representation (i.e., a precise definition).

Data structure
This is a data organization, management, and storage format that enables efficient access and modification.

Data-Driven Design

User experience (UX) This refers to a person's emotions and attitudes about using a particular product, system, or service.

Data-driven design is often used to improve **user experience**. In the context of user experience, data-driven design can be described as a data-supported design for a better understanding of the target audience. It shows that your work is on the right track, finds the sore points, and can help to improve the design by adding objectivity (Chapman, n.d.).

Without trying to do any user research, many designers believe they know what users want. Roughly speaking, designers in the vast majority of cases are not consumers and usually do not have enough evidence to support the choices made on how to build the best user experience. Data provide designers with knowledge so they can produce the best possible designs for the individuals who use their products. Such knowledge can come in different ways, from primary and secondary sources. For designers, the crucial thing is finding out which data are worth using and which can be ignored.

Data-driven design allows designers to overcome their biases and encourages them to step beyond best practices. Designers can use insights from their particular audience to customize user experience. In addition, data play an important role in Agile development, especially in some key areas, such as product management. One of the major benefits of using a data-driven approach is the ability to address the underlying challenges surrounding delivery that improve the agility of development. Data as a measurement basis can simplify the decision-making process by

- showing where problems lie in the UX, so that these can then be specifically resolved;
- drawing a clearer and more objective picture of user needs, since they are generated by the users themselves and are not subject to personal tastes or preferences; and
- developing hypotheses on the basis of such data, recording clear, unambiguous, and precise formulations, measurability, a time frame, and expectations (Chapman, n.d.).

Behavior-Driven Development

Behavior-driven development (BDD), also known as specification-driven development (SDD), is a technique of Agile software development that strengthens the collaboration between quality management and business analysis in software development projects.

In behavior-driven development, during the requirements analysis, the tasks, goals, and results of the software are recorded in a specific text form, which can later be executed as automated tests and thus the software can be tested for its correct implementation. The software requirements are, therefore, mostly written in "if-then" sentences. This is intended to facilitate the transition between the language of the definition of the functional requirements and the programming language by means of which the requirements are implemented. Behavior-driven development consists of the following elements (North, n.d.):

- Strong **stakeholder** involvement in the process is encouraged through **outside-in** software development focused on meeting the requirements of the clients, end users, operations, and insiders.
- A textual description of the behavior of the software and software parts is provided through case studies, the use of standardized keywords to mark preconditions, external behavior, and desired behavior of the software.
- The automation of these case studies uses mock objects to simulate software parts that have not yet been implemented.
- There is a successive implementation of software parts and replacement of the mock objects.

Stakeholder
This is a person or group who has a legitimate interest in the course or result of a process or project.

Outside–in
This method optimizes the process of software development by focusing on satisfying the needs of stakeholders.

4.3 Programming Styles

The search for an optimal approach to software development often inspires new process models, which are mostly described in the form of Agile frameworks. Such frameworks aim to increase transparency and the speed of change and are intended to lead to faster deployment of the developed system in order to minimize risks and undesirable results in the development process.

Extreme Programming

Extreme programming (XP) essentially describes the way software is programmed. It focuses on Agile processes, short development cycles, and fast response times to new or changing requirements (AgileAlliance, n.d.-a). The main characteristic of extreme programming is the cyclic approach at all levels: from programming through daily coordination in the development team to joint requirements management with the customer.

In principle, extreme programming is targeted to the requirements of the customer. This sounds obvious at first, but classic software development can only respond to customer wishes to a limited extent, and it becomes especially difficult when these wishes change regularly. Extreme programming also tries to promote the creativity of developers and accepts errors as a natural factor in the work. At the same time, extreme programming, like other Agile methods, is based on iterative processes. Completing a big project from start to finish and investing several months only to find at the end that the result does not fit simply breaks XP. Instead, there is a specific focus on constant testing, discussion, and publication in short cycles. In this way, errors can be quickly identified and eliminated.

To meet the requirements, a rather clear framework has been developed. It comprises various values, principles, and techniques. In addition, concrete roles are assigned so that tasks can be clearly allocated.

Values

The extreme programming values are communication, simplicity, feedback, courage, and respect. These values are outlined below (AgileAlliance, n.d.-a):

- Communication. The software development process relies on communication to transfer information between team members. Extreme programming highlights the significance of an effective form of contact like face-to-face conversation with the aid of a white board or other drawing mechanism.
- Simplicity. This means asking the question of what the easiest way that is going to work is. Here, the goal is to prevent waste and to do only absolutely necessary tasks in order to keep the design of the system as simple as possible so that it is easier to manage, maintain, and revise. Simplicity also means discussing only the requirements that you know about and not trying to predict the future.
- Feedback. Teams can find areas for improvement and are able to revise their practices through continuous feedback on previous activities. The team builds something, collects input on the concept and execution, and then adjusts the product accordingly.
- Courage. This is needed to solve organizational problems that reduce the productivity of the team. We need to be bold enough to stop doing something that does not fit and try something new. We need the confidence to embrace and act on input, even when it is hard to accept.
- Respect. Team members need to value each other in order to be able to communicate effectively, as well as to provide and accept feedback. Respectful communication improves collaboration and allows the team to recognize clear designs and solutions.

Rules

Don Wells defines the rules of extreme programming as follows (Wells, 1999):

- Planning
  - writing user stories
  - scheduling releases in release planning
  - making frequent, small releases
  - dividing the project into iterations
  - doing iteration planning at the beginning of each iteration
- Management
  - giving the team a dedicated workspace
  - setting a pace that can be maintained
  - holding a stand-up meeting at the beginning of each day
  - measuring the pace of the project
  - moving team members around
  - fixing XP when it breaks
- Software design
  - valuing simplicity
  - choosing a system metaphor
  - using CRC (**class-responsibilities-collaboration**) cards
  - not adding any functionality early
  - refactoring whenever possible
- Program
  - ensuring the continued availability of the customer
  - writing code according to agreed standards
  - coding the unit test first
  - creating all code in production in pair programming

Class-responsibility-collaboration
This is a brainstorming tool used in the design of object-oriented software.

- ensuring that only one pair of developers **integrates**
- integrating often
- using a dedicated integration machine to increase the speed and quality of work
- using joint ownership of the code

- Test
  - ensuring that all code undergoes unit tests
  - ensuring that all code passes all unit tests before it is released
  - creating tests when a bug is found
  - running acceptance tests often and publishing the result

Integration
This is the process of combining subroutines, software modules, or complete programs with other components of the software in order to create an application or to enhance the functionality of an existing application.

Practices

Extreme programming also suggests the use of 12 practices while implementing applications (Altexsoft, 2018). They are as follows:

1. Test driven development. Developers first write the test for a functionality and then the actual production code. The incremental approach of small tests taking only a few minutes' time and the implementation of the code ensures that testing and development are closely intertwined.
2. The planning game. This is a meeting to decide which user stories to schedule into the next iteration or release. The participants come from the project's team, IT, and the business stakeholders group.
3. On-site customer. This refers to a customer or user representative who will be present and working with the implementation team. The end user should be completely involved in the production process. The customer should be present at all times to answer team questions, set goals, and settle conflicts, if necessary.
4. Pair programming. This describes a scenario in which two programmers share a workstation and work together on the development of a feature or task.
5. Continuous integration. This is a software development method in which new code is continuously integrated into the existing code base. Developers will always keep the application completely integrated. Extreme programming teams take iterative development to a new level since they commit code several times a day.
6. Code refactoring. This means improving, clarifying, and optimizing the internal structure of existing code without affecting its external behavior. Refactoring does not involve rewriting code or fixing bugs. The term "refactoring" refers to specific, finite methods for refactoring code, such as the extract method used to clarify the meaning and purpose of a piece of code.
7. Small releases. A release is the final delivery of a software package after the completion of multiple iterations or sprints. A release can be either the initial development of an application or the addition of one or more complementary features to an existing application. A release should take less than a year to complete, and in some cases it may take as little as a few months.
8. Simple design. The easiest one that works is the best software design. If any ambiguity is detected, it should be eliminated. The right design should pass all checks, have no redundant code, and include as few methods and classes as possible. It should also clearly represent the purpose of the programmer.

9. Collective code ownership. This practice declares the responsibility of a whole team for a system's design. Codes may be checked and updated by each team member. Developers with access to code are not going to get into a situation where they do not know the correct place to implement a new feature. The practice discourages duplication of code. The introduction of collective code ownership helps the team to collaborate together and feel free to suggest new ideas.
10. System metaphor. This means a basic design with a collection of certain characteristics. New team members need to be able to understand the design and its structure. Without spending too much time reviewing requirements, they should be able to start work on it. In addition to these, there should be coherent naming of groups and methods. Developers can try to name an entity as though it already exists, rendering the overall system design comprehensible.
11. Coding standards. The team must have similar sets of coding standards, using the same formats and styles for writing code. The implementation of a standard helps all team members to read, exchange, and refactor code easily; track who worked on what pieces of code; and make learning faster for other programmers. The code written in compliance with the same laws promotes collective ownership.
12. 40-hour week. Extreme programming projects require developers to work efficiently, to be productive, and to maintain the quality of the product. They should feel well and rested in order to comply with these criteria. Maintaining a work-life balance avoids the burn-out of professionals. In extreme programming, the maximum number of hours should not exceed 45 hours per week.

Pair Programming

Pair programming describes a scenario in which two programmers share a workstation and work together on the development of a feature or task (Altexsoft, 2018). One of the two programmers writes the code. The other one reviews and provides strategic direction. As they work on this task, the two programmers regularly switch roles. One or both programmers continually comment on the development process (Tuple, n.d.). For pair programming to be effective, the workspace must also be designed for two people—the desk should at least have enough room for two chairs. The background noise in the room should be kept low and not be much louder than a quiet conversation between one or more such pairs.

Pair programming is often considered to be a part of extreme programming. In fact, pair programming belongs to the 12 traditional practices of extreme programming. Since extreme programming is considered an Agile method, pair programming is often described as an Agile approach. This is correct to a certain extent; however, pair programming can also be used in classic developments.

In pair programming, there are two main roles that are assigned at the beginning of a project (Böckeler, 2020). One programmer is the pilot or driver. This programmer writes the code and ideally explains their approach and train of thought out loud so that their partner can understand it. The second acts as the navigator or observer. During the input, this second programmer checks the code for errors, possible problems, and simpler solutions while keeping the overall problem in mind. This process ensures that the code being created is as good and simple as possible.

In pair programming, the regular exchange of roles is an essential step. In addition to the distribution of roles, the composition of the pairs should also always be varied. Not only the exchange of roles, but also regular breaks must be observed. Since the programming method requires high concentration over long phases, the process can be exhausting. Breaks are essential to keep looking at the code with fresh eyes and to clear the head in between.

Another important aspect of implementation is to understand pair programming not as a form of control or supervision, but as collaborative development aimed at finding the optimal solution. Accordingly, both partners are on an equal footing and any problems that the navigator notices are discussed and solved in this way. However, this only succeeds if a uniform programming style is mastered throughout the team.

Pairs

The pairs should be regularly regrouped. In terms of employee experience, three basic constellations are possible, each offering different advantages and disadvantages (Böckeler, 2020). The possible constellations are

- expert-expert,
- expert-newcomer, and
- newcomer-newcomer.

If two experts work together as a pair, this initially promises fast results with a high-quality output. However, it can happen that both are “stuck” in the existing structures and do not question them, meaning that other possible solutions or approaches are not even considered.

The combination of an expert with a newcomer offers the opportunity to carry out the familiarization and onboarding during the day-to-day business. At the same time, new ideas can be created, since the novice may question current procedures and methods and the employee who has already been with the company for a longer period of time may have to reflect on these to find the reason behind the existence of these procedures. However, it may also be that a new employee does not directly dare to express critique initially. Because of this, they may fall into a passive role and merely watch the expert or implement what the expert says when coding. In addition, the expert may tend to ignore comments coming from the newcomer. Therefore, an appropriate framework of trust should be created and care should be taken to ensure that equality is maintained in this constellation as well.

The pairing of two newcomers involves a certain risk, since both programmers have little or no knowledge of the company’s internal procedures and code base. At the same time, the teamwork here reduces the potential for errors compared to the individual work of each new employee who has little experience to date, since the partner also provides a control authority that can notice any errors (AgileAlliance, n.d.-b).

Advantages

There are a number of advantages that are often mentioned in the context of pair programming (Green, 2020). They are as follows:

- Knowledge transfer. Knowledge between both participants is shared and increased. Other perspectives broaden individual horizons.
- Enjoyment of the work. Often, the joy of exchange and interaction increases, at least for a certain amount of time.
- Improved collaboration. This can be observed in the tandem and also in the entire development team thanks to pair rotation.

In addition to the aforementioned advantages, additional benefits include better code, fewer errors, lower risk, improved discipline, and higher efficiency (Green, 2020).

These benefits may or may not apply to every situation. Even though two developers implement a common idea to solve a task, another idea might still be better. The implementation of the idea may be efficient, but if the wrong idea is implemented, effectiveness suffers. So, it is important not only to do things right, but also to do the right things.

People also like to refer to an "integrated" code review. Since four eyes see more than two, it can be assumed that more errors can be found during implementation than during a self-test by a single developer. However, is this really more effective or efficient than if done by a separate developer? In any case, it is clear that this advantage should also be examined in each individual case.

Best practices

There are some aspects and recommendations that organizations and/or the pairs can use to make their lives a little easier. Some best practices are briefly outlined below (Green, 2020):

- Clarify the general scope of pair programming.
- Clarify the specific collaboration, e.g., when you start in the morning, when you stop, when there are scheduled breaks, or at which workstation you work.
- Work on one task at a time—one task, one goal, one approach.
- Ideally, there are coding conventions and coding styles in the organization. The navigator should make sure that they are adhered to or inform the driver of any violations.
- Discussions are part of pair programming, but collaboration often takes place in open-plan offices, so the volume should be considered reasonable.
- Use line numbers to make it easier to identify specific lines of code.
- Play "ping pong." For example, in the course of test-driven development, developer A writes a test (ping) while developer B writes the implementation to pass the test (pong). Then, developer A extends the test (ping) and developer B extends the implementation (pong), going back and forth until the task is complete.
- It can be useful to use a timer to switch roles at fixed times, for example, every 20 minutes. The more well-rehearsed a tandem is, the less important the use of a timer becomes.

- Tandem programming is also a matter of attitude. Instead of “I have an idea, give me the keyboard,” it would be more desirable to have an “I have an idea, you take the keyboard sometime” approach.
- Last but not least, arrange lessons learned or retrospectives to learn from and with each other.

Mob Programming

Mob programming is a relatively new software development method that relies on a special kind of teamwork. Five to ten developers work on a task at the same time and in the same room. They also use a single terminal whose user interface can be projected onto a large area of the wall with the help of a projector. As a rule, team members take turns every 30 minutes or so, but they communicate constantly, do preliminary conceptual work, and contribute new ideas to collaborative development. Mob programming is modeled on Agile methods, specifically pair programming, which is designed to improve the quality of software through the principle of dual control. This principle is extended by mob programming to an even more comprehensive control of functionality and quality of the application.

Although the term has been around for longer, developer Woody Zuill (2014) has been considered the founder of this development approach since his “mob programming” presentation at JavaOne 2014 in San Francisco (Zuill, 2014). According to Zuill, mob programming is a process in which programmers come together in one place and collaborate on the same project, sharing even the same computer. Mob programming can increase productivity on certain projects by having multiple developers work collectively on a predefined problem (Zuill, 2014).

According to Zuill (2014), mob programming is designed to prevent blockages that are unavoidable with other development approaches. Communication barriers, personnel changes, and decision-making problems would get in the way of productivity on many projects. Zuill experimented with different ways of working a few years ago and found it best to let team members decide how they want to work or figure it out as they go. With a team of six developers, he tried different coding techniques.

Advantages

Mob programming (also called mobbing) has some advantages over other methods. Similar to pair programming, mob programming can reduce the error rate. Two pairs of eyes see more than one and four even more than two. Also, given the continuous discussion, the quality of the code is bound to increase because ideas are only implemented when everyone involved agrees; workarounds and emergency solutions that don’t really satisfy anyone rarely make it into the code.

In addition, team members from all disciplines sit together in mob programming. Waiting times because the expert for one area is busy with another task are a thing of the past. The work in progress is automatically limited to one item and everyone contributes their knowledge directly to each work step. Even internal team meetings to exchange informa-

tion about the project status become largely unnecessary; everyone knows where the project stands at any time, tasks do not have to be distributed, and telephone calls can be handled jointly. The flow of information within the team could not be better.

SUMMARY

In this unit, we learned about software development paradigms. We saw several kinds of programming paradigms and the difference between imperative and declarative programming paradigms, which are considered the two main ways of writing source code. In addition, we learned about software design paradigms and discussed domain-driven design, data-driven design, data-oriented design, and behavior-driven development.

We have also briefly explored three software programming frameworks, namely extreme programming, pair programming, and mob programming. We considered the values, rules, and practices of extreme programming. We discussed the different roles available in pair programming, as well as some advantages and best practices. Finally, there was a short introduction to the concept of mob programming.

UNIT 5

EXPERIMENTATION AND PRODUCTION

STUDY GOALS

On completion of this unit, you will have learned …

- the process of developing a machine learning model and how it differs from the standard process of software development.
- the life cycle of a model once it reaches production.
- the function of continuous integration and continuous development.
- how to build a scalable environment based on virtualization, containers, and platforms aimed at machine learning pipelines.

5. EXPERIMENTATION AND PRODUCTION

Introduction

In this unit, we will discover the development process of a machine learning solution, the steps needed to develop machine learning models that generate predictions, and how it differs from the model of traditional software development. We will get to know the explorative and iterative nature of the development process, starting from understanding the problem, to gathering and exploring data, generating candidate solutions, testing them, and integrating it all into a system. We will then explore the specific challenges involved in bringing a machine learning solution to production, due to its nondeterministic nature, the kind of teams that develop it, and its dependency on the data used to train it. We will learn about the requirements and challenges of version control, testing, and monitoring for a machine learning solution in production.

We will then delve into the challenges presented by adopting an Agile and iterative development model, its effect on the structure of an organization, and its internal processes. We will explore the DevOps approach with all its intricacies, and how it structures a pipeline from development to production.

When we want to run a large machine learning model, new issues appear. We will learn how to build replicable development environments, how to encapsulate processes in containers, and how to manage them at scale. We will discover well-known tools such as Docker and Kubernetes, and get acquainted with KubeFlow for machine learning (ML) deployment pipelines.

5.1 Experimentation and Production

Machine learning model
This is an algorithm that can be trained on data. A trained model can be used to get predictions on previously unseen data, e.g., to recognize certain types of patterns.

In this context, a **machine learning model** can be considered as a black box that, given a set of inputs, returns some outputs. The model is trained on known data, using a range of possible algorithms; the process is intrinsically experimental, and the result is intrinsically nondeterministic. When developing traditional software, we are presented with a limited set of conditions to react to, while in the domains where we use machine learning the conditions can be very complex. As an example, we could train a computer to differentiate between images of dogs and cats. We could try to define manually all the defining characteristic of “a dog” and of “a cat” in an image, but it will be time-consuming, and brittle in case of changes. What if we also want to differentiate between cats, dogs, and sheep? When using machine learning, we give a model input images and labels telling the model whether the image is a cat or a dog, and it then learns an association between inputs and labels, so that it can hopefully predict the right answer (cat or dog) for previously unseen images.

The development of a model has to bring value to the organization developing it, and we can conceptually divide the process into the following seven steps:

1. understanding the problem,
2. finding good assumptions and hypotheses,
3. collecting the available data,
4. exploring it to see if the assumptions and hypothesis seem realistic,
5. experimenting with possible models and data processing steps to solve our problem,
6. training a model, and
7. deploying it on a development pipeline to make sure it behaves as expected.

As we delve deeper into this topic, please keep in mind that all steps are connected to each other and they do not really happen in isolation, they inform each other, and we often go back and forth between one step and another (Educba, n.d.).

Understanding the Problem and Generating Hypotheses

We first need to understand the problem, and generate hypotheses about how to solve it. The scope tends to start somewhat vague, like "improving turnover," "reducing churn," or "identifying customers that won't pay," and it is necessary to find out what is actually needed for the project to succeed in a process called requirements engineering. To achieve this, we generally apply project management techniques, interviews with matter experts and stakeholders, and design thinking, among other processes. We often go back and forth checking and correcting assumptions: data science is still a relatively new field, and many organizations have not adopted a data-driven approach.

At the end of this process, we will have established clear goals with clear performance indicators. Rather than "identifying problematic customers," we now have a more concrete goal, such as "reducing the number of customers not paying after two weeks by at least 20 percent." Once the problem is defined, the next step is to start generating hypotheses about what could solve it, and what kind of data we would need. The data science process is very much about data: We need the right kind, and enough data, to properly train a model.

Data Collection, Cleaning, and Wrangling

Armed with an understanding of the problem and some hypotheses on how to tackle it, we now need data to work with. In some cases, we can get it from a database or a spreadsheet; in most cases, we need to gather and extract it thought much more complex process of **ETL** (extract, transform, load). In cases where we lack the data we need, we must either access it from other sources, or find ways to generate it ourselves. Finally, we could discover that the necessary data is just not available, and we would need to change our approach or find a way around the lack of data.

ETL
This is an acronym that stands for "extract, transform, load." It refers to the general procedure of copying data from one or more sources into a destination system.

The collected data will generally need cleaning and reformatting for the intended purposes. A process called data wrangling or data munging involves transforming and mapping data from one format to another to make it more appropriate for our uses, adapting ranges, removing some fields, applying standardization, and cleansing invalid values (Wikipedia, 2020a).

Exploratory Data Analysis

Once the data has been collected, cleaned, and stored, we can start working with it. We want to see what it can tell us, what sort of information we can extract, and what could be the appropriate next step. This step is called **exploratory data analysis (EDA)**, a process of investigating the data to discover patterns, test hypotheses, and check assumptions using summary statistics and visual methods.

Exploratory data analysis (EDA)
This is an approach to analyzing data sets to summarize their main characteristics, often visually.

It is a practical and exploratory process, strongly based on the experience and skillset of a data scientist, involving new ideas to explore, many dead ends, and very little reproducibility. A data scientist will explore different possibilities and delve deeper into something when it looks interesting. It can involve using summaries about the type, number, and average values of the data, and often involves visualizations, since humans are very visual beings, and we can see, at a glance, patterns that would escape us in a tabular format. At the end of the EDA process, we have clearer, more refined hypotheses, as well as some algorithms in mind to use in training the models.

Experimentation, Feature, and Model Selection

With the data accessible and useable, and an idea of what the data can tell us, we start experimenting with models. We generally test several different models, with different levels of complexity, starting from the simplest ones (e.g., linear regression, which is a linear approach to modeling the relationship between a scalar response and one or more variables), scaling up if needed to state-of-the-art models like the neural networks (computing systems vaguely inspired by the biological neural networks) that are getting a lot of attention in the last few years. If a simpler solution is sufficient, it is often better to stop there, because they tend to be lighter and easier to train and deploy.

It is important to understand that a machine learning model needs to be trained; we feed it some data and check the result against other data for which we have known results. The process is nondeterministic, and often resource and time-consuming. During the experimentation phase, we often use only a subset of the available data to speed up training and iterations, and we include the full dataset only during the final stages of training.

When training a model, we have a choice of what features (input variables of the given model, e.g., variables and predictors) of the dataset to use. This process is called **feature selection**. Some features can contain noise that would confuse the model and make it less precise and too many features can incur in the curse of dimensionality, making it hard (or impossible) for the model to reach an optimal point during training. One way to avoid the curse of dimensionality is to use methods for **dimensionality reduction** with the goal of maintaining as much relevant information as possible while reducing the number of features. Once this is done, we can consider possible models. The main kind of machine learning problems are: supervised, unsupervised, and reinforcement learning. They are defined as follows:

Feature selection
This is the process of selecting a subset of relevant features for use in model construction.

Dimensionality reduction
The transformation of data from a high-dimensional space into a low-dimensional space while retaining some meaningful properties is called dimensionality reduction.

- Supervised learning maps input data to known output data (e.g., categorizing images as cats and dogs or predicting the stock market).
- Unsupervised learning explores patterns in your data (e.g., clustering documents by topics by looking at similarities in those documents without specifying the topics beforehand).
- Reinforcement learning studies how agents interact with their environment by rewarding favorable situations and punishing bad ones (e.g., learning to play a videogame or a board game).

Once we have selected the model that we want to try, we divide our data into train, test, and sometimes validation data sets. The train data are used to train the model, and the test data are used to evaluate the performance of the model on the metrics we are interested in. The two sets generally cannot overlap. Similar to university tests, it is not useful to ask the exact same questions during a test that have been asked while studying (training). We often set aside a validation data set to compare the performance of the different models we are considering.

We have noted that this exploratory step is often performed with only a subset of the data in order to select the most promising models. These models can then be retrained with more data and choosing different characteristics of the models (called **hyperparameters**) to improve performance. This process is called hyperparameters tuning, and it involves training several models with different hyperparameters and selecting the models with the better results.

Hyperparameters
These are parameters whose values are used to control the learning process.

Training

Once we have selected some models and tuned their hyperparameters, we can start training them to compare their performance. We usually train several models with different choices of features. In case our dataset is very large, and in the case of models requiring a lot of time and resources to train, it can be useful to use only a subset of the dataset. Once we have selected the desired model, or very few candidates, we can proceed with the whole dataset.

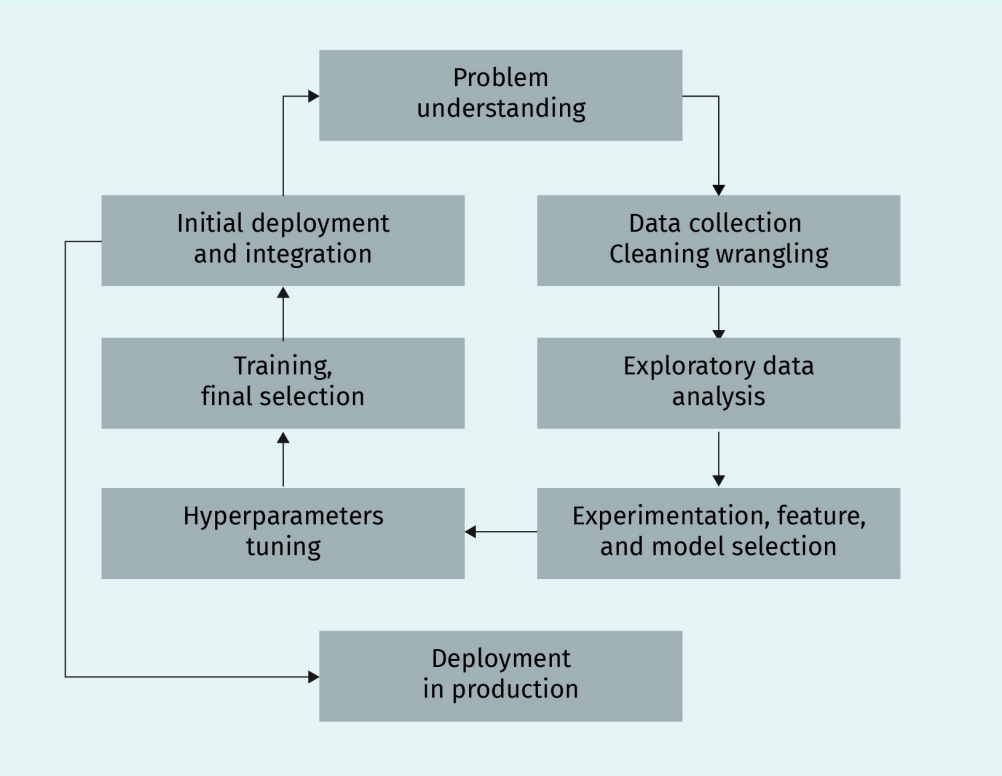
Depending on the dataset size, this can require a lot of time. One of the biggest existing models, GPT-3 by OpenAI, has 175 billion parameters and would require 355 years and $4,600,000 to train on current hardware (Brown et al., 2020). However, models often take between a few hours to several days to train, so it is important to do it only when we are quite confident the result will be, if not useful, at least informative.

At the end of the process we have our trained models and can check their performance using the test dataset.

Initial Deployment and Integration

The model alone is not very useful, so it needs to be made available to the one or more applications in the organization. A common way of doing this is to encapsulate the model in an application that provides the predictions as services to the other applications. Once this is done, we can do some basic tests of usability, connect it to other services, and set it up in a development pipeline to make sure it behaves as expected.

Figure 25: Model Development and Production Lifecycle (1)



Source: Created on behalf of IU (2022).

Differences to Software Engineering

The machine learning development life cycle has some similarities and some differences, compared to the traditional software engineering life cycle. In both cases, it is necessary to understand the problem we are solving for the organization; however, in the traditional process, the problem tends to be much better defined and the development tends to be more straightforward, even if subject to cycles of implementation and testing assumptions. The machine learning life cycle, on the contrary, is all about experimentation and iterations, testing hypotheses, and making sure the performance of the metrics we care about is within a desired range. The result of our process depends not only on the algorithms we use, but also on the data we use for training. While the result of a traditional software development is supposed to be deterministic, the result of a machine learning development process is a statistical, nondeterministic model satisfying some constraints

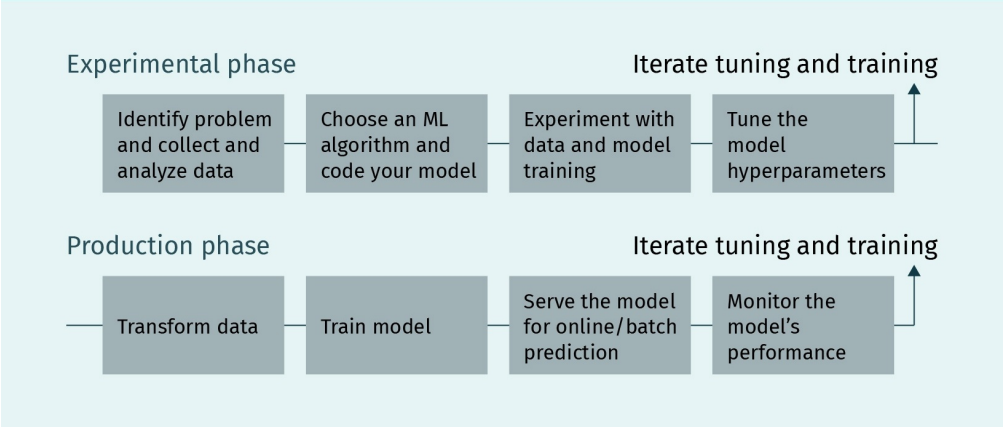
that will hopefully perform as expected on new, real data (Hegde, 2020). For this reason, this process has sometimes been called Software 2.0, using machine learning models to solve problems (Karpathy, 2017).

Bringing a Model to Production

The requirements to bring models to production have many similarities to the production life cycle of traditional software, but also present some key differences. For example, the process needs to be integrated in a continuous development and deployment pipeline, allowing for quick turnarounds. However, the complete ML system contains the code to create the model, the trained model itself, other possible artifacts (outputs created by the training process), and the data used to create the model; all of this must be integrated in a deployment pipeline.

Bringing and keeping models in production requires versioning the code and everything needed to generate it; storing it; deploying it; making sure the interfaces that other applications used to access the models do not break; serving the predictions; logging the results; and monitoring health, performance, and what happens when new unseen data is used (Google, n.d.).

Figure 26: Model Development and Production Lifecycle (II)



Source: Pedori (2020).

Models in Production

Machine learning systems are software systems, so when they are in production, they face all challenges of traditional systems, and some additional specific ones connected to the nondeterminism and the effect of new data and a changing environment. We can think about the following systems we use practically every day to identify the challenges they can present.

- Online search engines are machine learning systems that try to match a search query to the best possible web page corresponding to it. The result should be accurate and fast, and the system needs to be constantly retrained on fresh web content. They sometime adapt to the usage of the user, too.
- Movie streaming websites often include recommendation systems, which propose new content based on previously watched films. They need to learn constantly, be available, and serve huge amounts of data.
- Financial services monitoring transactions to identify fraud are machine learning systems that need to react very quickly. They must be up-to-date on patterns of usage with a very thin margin of error, since rejecting a valid transaction annoys the users, and accepting fraudulent transactions can be very expensive.

If we consider these examples, we can imagine the huge challenges presented by these systems: they must be fast, constantly available and updated, and be able to revert to a working state in case something goes wrong. To achieve this, we need to keep track of everything used to create and serve the models (with version control), deploy the models, test them, monitor their performance and health, and be able to react when things go wrong. Let us investigate these steps in more detail (Fowler, 2019).

Version Control

Version control is the management of changes to documents, computer programs, and other collections of information. We use it to track changes, be able to revert to previous working solutions, and coordinate updates. We usually identify the changes via numbers or letters, called revisions or revision numbers. In machine learning, an initial version of a set of files or code could be "revision 1," increasing the number after any change, moving to "revision 2," etc. With a version control system, we can keep track of changes (when and by whom they were made), compare different versions, and synchronize changes between different parts of an application that need to communicate with each other.

While in in traditional software development we only have to keep track of code (both the application and code used to support and integrate it), in a machine learning system, in addition to tracking the code used to create the model and to deploy it, we need to keep track of the trained model itself, as well as of the data used to create the model.

Versioning code

In this case, there are two kinds of code written in a programming language: the implementation code (used to connect and serve components) and the modeling code (used for model development). They can often be written in different programming languages, and can be developed by different teams. We version the code to keep track of releases and dependencies that could need to be updated, and to make sure it interacts with other components as we expect. This is the same in traditional software development, and we will cover the concept of infrastructure as code in a later section. A machine learning system also needs to version data and models.

Versioning data

The data used to train a model can change both its content and the way it is structured. We use metadata to refer to the way the data are structured, their format, names, order, and number of columns. New data can have different statistical properties and distributions, or they can just contain cases not seen during the development of a model. If we want be able to generate the same model over and over again (a requirement for reliable deployment), we need to make sure we are using the same data that we used when we shipped. We achieve this by versioning both the data and their metadata.

One issue in versioning the data used to generate the model is its size. Traditional version control systems expect the code to be in kilobytes or megabytes, while training data can consist of hundreds of gigabytes, or more. This has given rise to systems specializing in versioning data.

Versioning models

We also need to keep track of and version the models and other artifacts generated during the model development, matching them with the code used to generate and support them and the data used to train them. The storage requirements for the models are simpler than the ones for the data, but still bigger than most code, so our version control systems often need to be expanded to allow for it. Versioning data and code allows us, in principle, to be able to re-generate a specific model, but since this takes time and resources, we want to store a specific version of a model together with the rest.

Deployment

Once developed and trained, we need to deploy the model so that it can be useful. There are two main scenarios: We can embed it as an artifact (the model file) in an application, building it together with the application using it; or we can deploy it as a service, wrapping it in an interface. The latter allows us to decouple model and applications, but it can increase latency and complexity.

Testing and Quality

In traditional software engineering, we use tests to make sure our system behaves correctly, and to be warned early if something breaks, either due to an update or an external change. In a machine learning environment, the scope is bigger and we want to test

- integration with other components, similar to traditional systems.
- data, validating it against the expected schema describing the metadata, and making sure it conforms to the data the model expects.
- model quality, since they are nondeterministic and we need to make sure their performance metrics fall in the range we are ready to accept.
- bias and fairness, a huge topic that we are not able to cover entirely in such a limited space. In short, we need to make sure the model behaves in a way that is aligned with the goals of the organization.

Monitoring and Observability

Once the model is live, we need to understand how it performs in production, closing the feedback loop to the development process and gathering data on how the model performs on live data. Monitoring is a standard practice for production software, covering: **KPI** (key performance indicator), software reliability, performance, debugging information in the case of faults, and other indicators that something unexpected is happening. We can use and adapt the same tools and practices for machine learning, capturing a number of different results.

Key performance indicator (KPI)
This evaluates the success of an organization or of a particular activity in which it engages.

- Model inputs. These are the data that are being sent to the models.
- Model outputs. Here, we compare the results to the predictions or recommendations, based on certain inputs, to understand how the model is behaving on real data.
- User action and rewards. This is the next step after the users receive the outputs. To make sure we are actually solving the stated problem, we need to keep track of the effect of the predictions. Do users buy the suggested items, watch the movie, click on the webpage, and pay on time?
- Model fairness. We want to make sure the model continues to behave in a way that is aligned with our values.
- Model computational performance. The reaction time can be critical in some cases, and we want to keep track of the central processing unit (CPU) and memory load in case we need to scale.

When Things Go Wrong

So far, we have considered the steps necessary to bring a model to production and the challenges involved. Most of what we have covered has the goal of keeping the model in production, minimizing the problems, or allowing us to react properly when things go wrong. If a system is in production long enough, things will go wrong, and we need to recover.The following list details things that can go wrong and how we can deal with them:

Model drift
The degradation of a model's prediction power due to changes in the environment is called model drift.

- If the model has been in production long enough, the data used to create it, and the data it is exposed to, will drift apart. This phenomenon is called **model drift** and will require us to retrain the model. Versioning keeps track of the new data, and testing ensures the new model works as expected.
- Part of the infrastructure can fail, and we need to be able to re-deploy or retrain a model when needed.
- Monitoring the health of the system lets us know when the aforementioned steps are needed.
- If we deploy an updated or newly trained model, we need to monitor it to ensure it behaves as expected; versioning allows us to roll back if it does not.
- If our model is constantly updated and retrained, we need to monitor its performance to make sure it does not degrade, and to be able to roll back to a known working state if, or when, it does.

5.2 Continuous Integration and Delivery

Software applications are increasingly developed in an Agile manner, and it is important to be able to **ship early, and ship often** (AOE, n.d.). To achieve this, it is necessary to automate the IT processes and infrastructures supporting the deployment to production. This often requires cultural changes in existing organizations in order to bring development and operations closer together. This gave rise to the concept of DevOps, a set of practices combining software development and IT operations, aimed at shortening the development life cycle and providing continuous delivery with high software quality. It also gave rise to the related concept of infrastructure as code (IaC), the process software runtime environment, and networking settings and parameters so that they can be stored and modified on request in simple textual format in the code repository (Null, 2020). The goal here is to be able to release a new version of the application quickly, allowing for rapid iteration cycles of development and testing, checking assumptions, correcting bugs and other errors, and adding necessary features. Currently, the most common way to ensure this is to adopt the concept of DevOps.

Ship early, ship often
This exemplifies a software development philosophy that emphasizes the importance of early and frequent releases in creating a tight feedback loop between developers and testers or users.

DevOps

DevOps is mostly about culture and procedures, and requires the adoption of an Agile mindset. The main concept for us here is the DevOps pipeline, outlining the transition from programming to operations and bringing the two closer together.

Figure 27: DevOps Pipeline

Source: Kobdani (2020).

Plan

It is important to plan the whole workflow before the developers start coding, involving product and project managers to generate a production plan for the whole team. In this stage, we work on segmenting the project in smaller parts that can be tackled in development sprints of one to several weeks in which individual team members work on their assigned tasks.

Develop

This is the actual programming stage, often involving shared coding guidelines. At the end of this stage, developers push the result to a shared **coding repository**, requiring pull requests and **code reviews** from other team members before having it in the master branch, the one used for building software for delivery.

Coding repository
This is a data structure that stores metadata for a set of files or directory structure in a revision control system.

Code reviews
This is a software quality assurance activity in which one or several people check the source code of a program.

Build

In a typical case, the pull request initiates an automated process that compiles the code into a build, a deployable package, or an executable. Some programming languages need to be compiled while others (like Python) do not. However, they all need to be checked for code problems to identify errors before they go through the pipeline and cause a bigger issue. In case of code problems, the build fails, the developer is notified of the issue, and the original pull request generally fails.

Test

Once the build succeeds, we move to testing. Tests can be manually run by the developers or automated. In the DevOps approach, there is no separate team for testing and quality assurance, instead, the developers design the tests themselves. Testing can also involve security checks, run load tests, and performance tests.

Release

If the tests pass, that gives us some assurance that there will not be any unexpected problems and we can move on to the release stage, which is sometimes considered a milestone. In some environments, the release happens automatically once the built test phases are passed, as we will see when covering continuous deployment. In many other organizations, it can involve a manual process of making sure that everything works and ensuring that the new release is up to the expected standards before approving it.

Deploy

When a release is ready, the application can be moved to production and deployed. When only minor changes are being implemented, the deployment can be an automatic process. In case of big changes, the build might first be deployed to a production-like setting to monitor how it works, often called staging.

In the case of critical updates, it is common to use a blue-green deployment strategy. This entails having two similar development environments where the latest application is hosted by one environment while the modified version is hosted by the other, usually called a staging environment. This lets the developers send the live requests to the staging environment, making sure it behaves as expected, and it allows the program to quickly revert to a known working environment in case there are issues not caught during the testing process.

Operate

Once the new version is live in the production environment and exposed to users, the production team makes sure that everything runs smoothly. This includes scaling resources when needed and collecting feedback from the users to make sure that the new version is not only technically sound, but solves the right problems.

Monitor

The live running system is monitored for performance, and for the result of user interaction, including the user feedback mentioned above. We collect data, logs, and analytics, and check for possible bottlenecks in the pipeline. Using the data collected during monitoring, the loop starts again planning new releases, improvements, and fixes.

Best Practices

A DevOps approach includes several best practices in an attempt to implement the lessons learned by several organizations and years of trial and error in different teams. We will cover two main aspects: the CI/CD pipeline (continuous integration and continuous delivery or development), and the concept of infrastructure as code (IaC).

Continuous integration (CI)

In any bigger modern development team, different developers work on different features of the same application. The way things were traditionally done was having a specific day to merge all the different features, sometimes called a “merge day.” This could involve a huge amount of work, since the developers would have to solve conflicts created by the different source code branches.

In the framework of continuous integration, developers can merge the branches they are working on at any time, once a day or even more often. The contribution is automatically validated and tested for integration with the rest of the environment, ensuring that the functionality is not compromised (Patel, 2020).

Continuous delivery (CD)

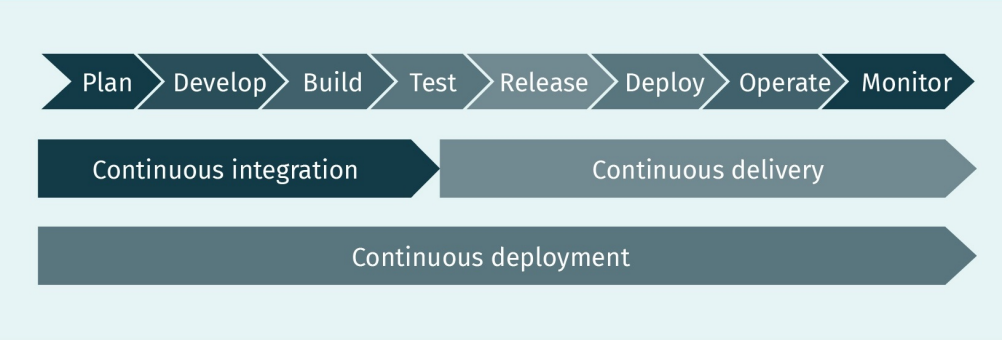
After automating builds and unit and integration testing, continuous delivery can automatically release the validated code to a repository. This requires that CI be integrated in the pipeline. The goal of continuous delivery is to have the code base ready for the production environment at any time; if it is merged to the master branch, we can ship it. Every change is expected to be able to deliver production ready builds that the operation team can quickly deploy into production (Patel, 2020).

Continuous deployment (CD)

Continuous deployment can be considered a natural extension of continuous delivery. When a production-ready build is released, it is automatically deployed into production, without a manual step or approval. This means that the whole pipeline must be very well thought out, making sure that issues are caught before the last stage, since the changes implemented by the developers end up live right away, potentially within minutes or less. This allows us to get feedback very quickly, but at the cost of introducing unexpected breaking changes. The whole process is designed to maximize the advantages while minimizing the risks.

The DevOps CI/CD diagram shown below presents a simple comparison of continuous integration, delivery, and deployment.

Figure 28: DevOps CI/CD



Source: Kobdani (2020).

Infrastructure as code

Infrastructure as code refers to the process of defining all software runtime environment and networking settings and parameters in files that can be stored and modified on request in simple textual format in the code repository as opposed to physical hardware configuration or interactive configuration tools (Null, 2020).

Provisioning
The process of setting up infrastructure is called provisioning.

The text files are generally called manifests, and can be **provisioned** and configured automatically to build servers, define testing, staging, development, and production environments. There, operations can be versioned and tracked, and we can make sure every stage of the pipeline uses the same environment, avoiding the famous issue of "works fine on my computer," where the code that worked in development or testing does not work in production.

Automatically provisioning the environment reduces the chance of human error, allows for faster scaling, and speeds up the development pipeline (Ziolkowski, 2020).

Impact on Team and Development Structure

The DevOps approach of getting development and operations to work together has the direct goal of developing, testing, and deploying applications faster. This requires changes in the team and development structure. First of all, it is necessary to eliminate, or at least minimize, the compartmentalization of teams and responsibilities, often called silos (DevOps, 2015). Doing so requires reconsidering and changing the team structure, also affecting team size and leadership.

Team Size

Teams should be small enough to easily work together, share information, and have very short coordination meeting, including the idea of a “morning stand-up meeting,” where everybody is standing, providing more encouragement to make it short. Amazon even introduced the idea of the two-pizza team, i.e., teams should be small enough that two pizzas would be enough to feed everyone (Gitlab, n.d.). This usually means five to eight people, including project managers and non-developers.

For some organizations, changing team size can be challenging, requiring big structural revisions, moving to functional or role-aligned teams, more communication, and a more Agile structure. This process, however hard, is part of the benefits of adopting a DevOps mindset, resulting in a number of smaller, company-aligned teams headed by team managers.

Leadership

An Agile DevOps-inspired mindset can be challenging for some leadership styles. It implies less control, iterations quickly testing assumptions, and smaller independent teams that cannot be directly managed by the company management. This change often requires further organizational adjustments that can be hard to accept, embrace, or implement, and that cannot be merely superficial. The organization needs to change, and to overcome the issues of change resistance, low change readiness, and weak employee commitment. Tackling these challenges gave rise to the concept of transformational leadership, a leadership style in which leaders promote, empower, and motivate workers to make changes that help the organization grow and shape its potential success (Wikipedia, 2020b).

Right mix

To enable an organization to move towards smaller, DevOps aligned teams and to embrace a fast development pipeline, it is necessary to identify skill gaps and bottlenecks. This also means determining which combination of roles and skills a company still needs to acquire in order to achieve the goals of the teams, not only on the technical level, but at the level of interpersonal and soft skills. This can involve upskilling current employees or recruiting new ones who bring the desired skillsets and personalities. One result of this is seeing a more diverse workforce as a strength.

5.3 Building a Scalable Environment

Scalability is one of the key objectives in a modern infrastructure because an organization that can scale consistently is one with a great potential for growth. This skill allows us to configure our systems to grow during high demand and to scale down when demand falls. The approaches we delineated describing an Agile development pipeline both enable and require scalability. This also applies to machine learning systems, which are, at their core, software systems.

A way to allow scaling is to concentrate on the elements of the CI/CD pipeline. As we pointed out, one important element of DevOps is infrastructure as code, which allows us to create replicable environments. This can happen in two main ways: virtualization or containerization.

Virtualization

Virtualization is the process of running a virtual instance of a computer system, called virtual machine (VM), in a layer abstracted from the actual hardware, generally running several operating systems. The applications running on top of a virtualized machine appear to have a dedicated machine, with the operating system and supporting libraries in the guest virtualized system not directly connected to the host operating system below.

A virtual machine is the emulated equivalent of a computer system that runs on top of another system. Virtual machines may have access to any number of resources, for example, computing power through hardware-assisted but limited access to the host machine's CPU and memory; one or more physical or virtual disk device for storage; a virtual or real network interface; and any devices, such as video cards, USB devices, or other hardware, that are shared with the virtual machine. The virtual machine is created, managed, and run by a program called hypervisor. Many modern operative systems have hypervisors built in, as in the case of Linux, with the kernel-based virtual machine (KVM) and Windows, with Microsoft Hyper-V (Opensource, n.d.; Wikipedia, 2020c).

Containers

Another concept related to VM is that of containers, which are conceptually similar to virtual machines, but more limited in scope. Both containers and virtual machines allow for running applications in an isolated environment, but containers are not fully isolated and independent machines. Containers run as isolated processes, sharing the same host operating system and libraries. This means they all share the same operative system, but are allowed to access specific libraries and data. It also means they are much lighter, faster to launch, and appropriate for running even a single program, or a separate application (Opensource, n.d.).

Containers are a very commonly used to provide the services of a machine learning model to other applications in an organization. The model is wrapped in an application providing an interface and a protocol to access the predictions of the model as a service, and it is deployed as a **microservice** in a container.

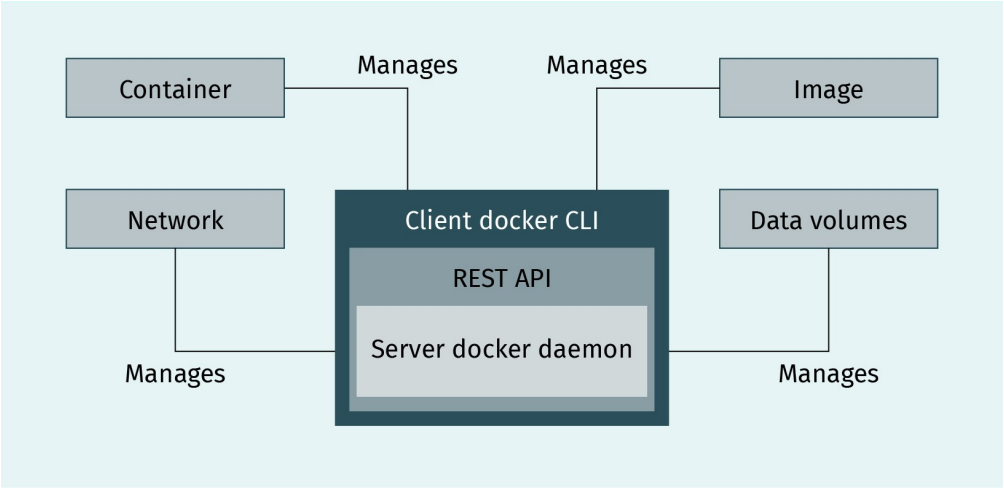
Microservice
A microservice is a kind of architecture that arranges an application as a collection of loosely coupled fine-grained services connected via lightweight protocols.

Docker

One of the most commonly used container technologies is Docker, an open-source solution that allows us to package our application and the associated dependencies into an image and run it on any machine. Docker extends existing Linux container functionalities, providing versioning of images and containers (Tozzi, 2017). The following terms are some basic Docker-specific terms you should know as taken from the Docker glossary (n.d.-a).

- Dockerfile. A text file that contains commands to create an image.
- Image. A Docker image contains elements, such as code, config files, environment variables, libraries, and run time, that are required to run an application as a container.
- Container. A standardized unit that can be easily built to deploy a particular application or environment.
- Docker registry. A service that contains Docker images and repositories.
- Docker daemon. A server which is a type of long-running program called daemon process. A daemon (also known as background process) is a Linux or UNIX program running in the background.
- Docker engine. A client-server application comprised of the Docker daemon, the API, and client services, forming the interface between the resources of the host and the running containers.
- Docker network. Docker includes support for networking containers through the use of network drivers.
- Docker volume. The preferred mechanism for persisting data generated by and used by Docker containers.

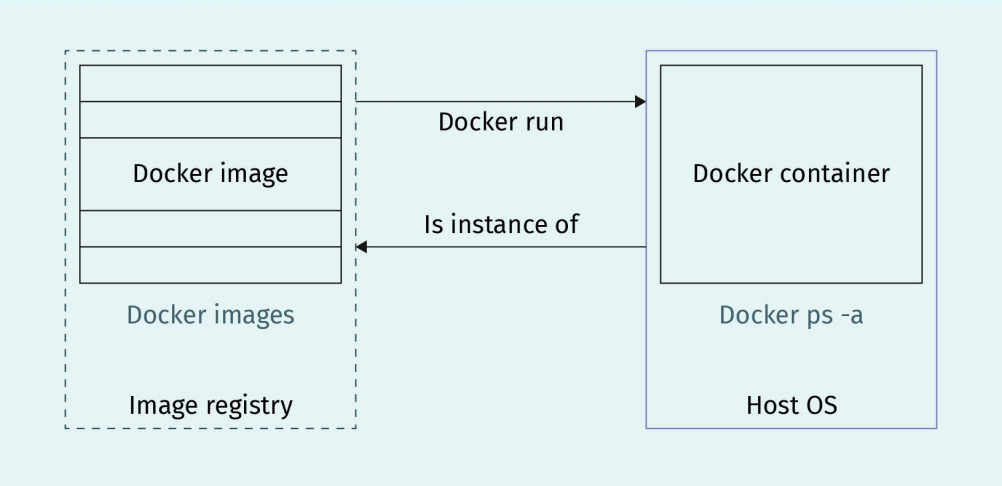
Figure 29: Docker Components



Source: Kobdani (2020).

An image is an executable package that includes all you need to run an application, namely code, runtime, libraries, environment variables, and configuration files. A container is launched by running an image. The following diagram illustrates the concept of Docker images and containers and how they are related to each other (Docker, n.d.).

Figure 30: Docker Image vs. Container



Source: Kobdani (2020).

Orchestration

When operating on a large scale with several containers interacting with each other, it is necessary to coordinate them. The process of deciding which ones start first, which ones depend on others, and how to behave in case of failure and shutdowns, is called orchestration. Container orchestration automates the provisioning, management, scaling, and networking of containers, and it is particularly necessary in an organization managing hundreds of containers (Redhat, n.d.). This is particularly important when paired with a microservice architecture. With microservices in containers, it is possible to orchestrate their services more easily. Container orchestration allows you to automate and manage the following tasks:

- provisioning and deployment
- configuration and planning
- resource allocation
- container availability
- scaling or removing containers to evenly distribute workloads across your infrastructure
- load balancing and traffic routing
- monitoring of the container status
- configuring applications based on the container in which they will run
- securing interactions between containers

Container orchestration tools provide a framework for managing containers and microservice architectures on a large scale. There are many container orchestration tools that can be used for container life cycle management, such as Kubernetes.

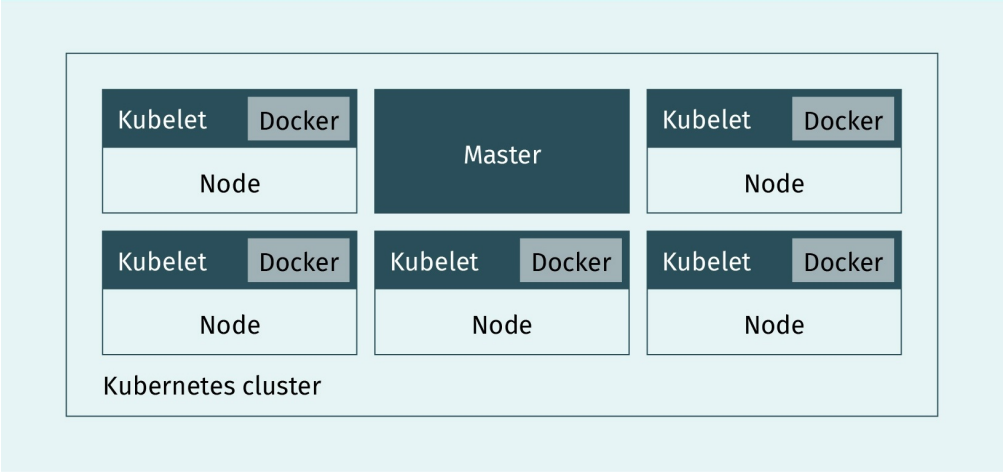
Kubernetes

Kubernetes is an open-source tool for the orchestration of containers, originally developed and designed by Google. It allows the development of application services that span multiple containers, planning and scaling of containers across clusters, as well as monitoring their health over time (Kubernetes Documentation, 2020). Kubernetes eliminates many of the manual processes associated with the deployment and scaling of containerized applications. You can have cluster groups of hosts (either physical or virtual machines) running containers because Kubernetes provides the right platform to manage these clusters easily and efficiently. More generally, with Kubernetes you can implement an infrastructure in your production environments that is completely container-based and reliable. The main concepts needed to understand Kubernetes are clusters and pods.

A Kubernetes cluster is a set of node machines for running containerized applications. Running Kubernetes means we are running at least a cluster containing a control plane and one or more compute machines, or nodes, at a minimum. A Kubernetes cluster consists of two types of resources: the master, responsible for the administration and coordination of the cluster, and nodes, the workers that run applications. The master coordinates all activities in the cluster, such as scheduling applications, managing the desired status of applications, scaling applications, and rolling out new updates.

A node is a virtual machine or a physical computer that serves as a working machine in a Kubernetes cluster. Each node has a "kubelet," an agent to manage the node and communicate with the Kubernetes master. The following diagram illustrates a Kubernetes cluster.

Figure 31: Kubernetes Cluster



Source: Kobdani (2020).

When we run applications on Kubernetes we have one master and one or more nodes. We instruct the master to start the application containers as well as the nodes that communicate with the master via the Kubernetes API, which end users can also use to interact with the cluster directly.

Pods are the smallest computing units one can build and manage in Kubernetes. They are a group of one or more containers, with shared storage and network resources, and a specification on how to operate the containers (Kubernetes Documentation, 2020). A pod models an application-specific "logic host," which includes one or more application containers that are relatively tightly coupled.

A pod always runs on a node, and a node may have several pods, all managed by the Kubernetes master.

MLOps

A machine learning system is a software system, meaning we can adapt the DevOps techniques used in the development and operation of large-scale software systems using the following criteria:

- Team skills. Data scientists are part of the team and are not always experienced engineers, so their code often needs some work to reach production quality.
- Development. Machine learning is experimental and nondeterministic. It is challenging to keep track of what did or did not work, maintaining reproducibility and code reusability.
- Testing. Testing an ML system includes elements not present in other software systems. In addition to typical unit and integration tests, you need data validation, trained model quality evaluation, and model validation.
- Deployment. Machine learning systems can require deployment in a multi-step pipeline to automatically retrain and deploy models, adding complexity and the necessity to automate steps that are manually performed by data scientists before deployment.
- Production. The performance of machine learning systems can be affected by the quality of coding, but also by the changes in data profiles. Models can decay in more ways than other software systems, and it is vital to keep track of this using statistics and monitoring, sending notifications, or automatically rolling back to known working versions.

As such, machine learning and other software systems are similar in the requirements for continuous integration of source control, unit testing, integration testing, and continuous delivery of the software module or the package. However, they differ in a few areas. CI is no longer only about testing and validating code and components, but also testing and validating data, data schemas, and models. At the same time, CD is no longer about a single software package or a service, but rather a system (an ML training pipeline) that should automatically deploy another service (model prediction service). We can also add the new property of continuous training (CT), unique to ML systems, which is concerned with automatically retraining and serving the models (Google, n.d.). These different and added requirements led to the creation of the concept of MLOps.

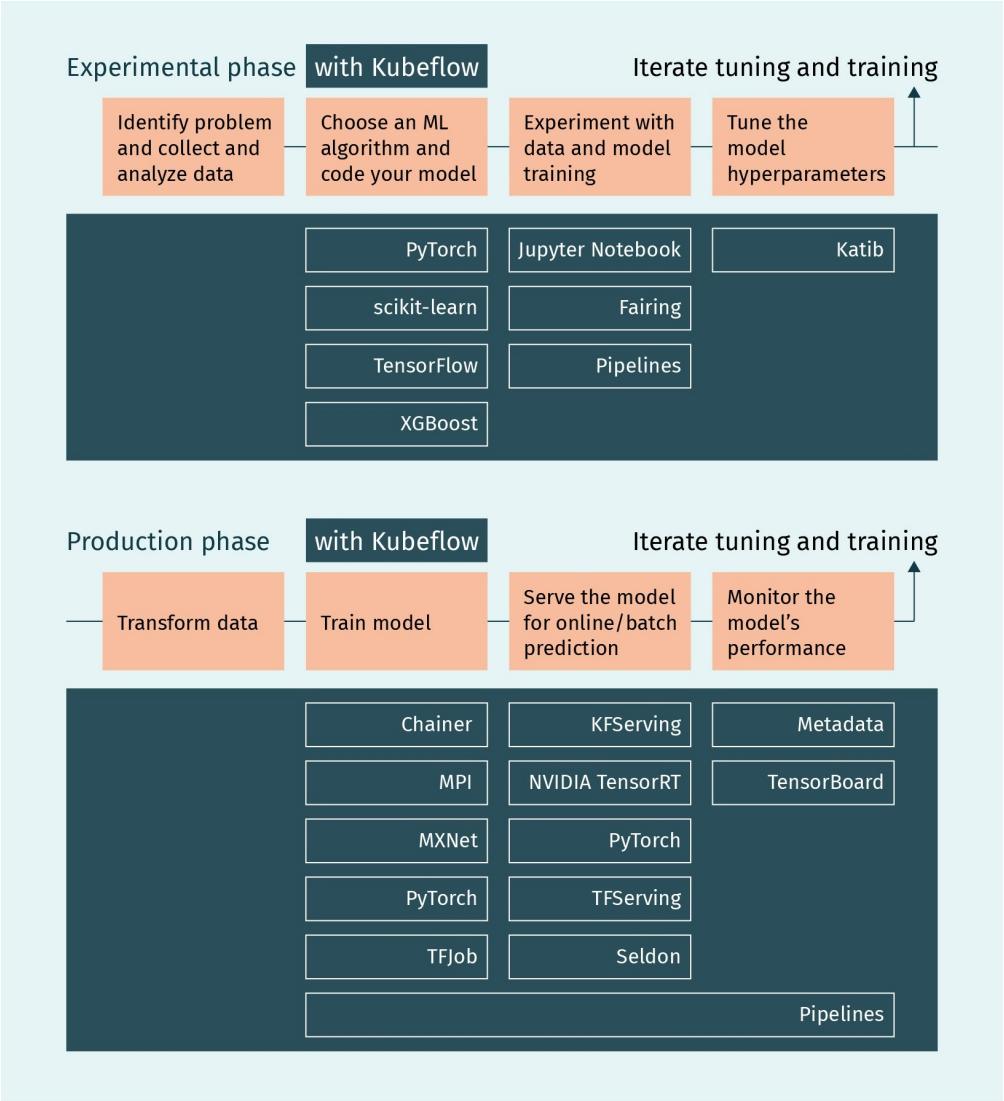
MLOps (a compound of "machine learning" and "operations") is a practice for collaboration and communication between data scientists and operations professionals to help manage the production machine learning life cycle described in the previous section (Google, n.d.).

MLOps and Pipeline Management with Kubeflow

Kubeflow, started by Google as an open-source platform for running **TensorFlow**, is a multi-cloud, multi-architecture framework machine learning platform that manages deployments of ML workflows on Kubernetes. It is aimed at data scientists building and experimenting with data pipelines, and for deploying machine learning systems to different environments, as well as for testing, development, and production (Kubeflow, n.d.-a). It offers tools to support most of the steps we covered when discussing model creation and the deployment pipeline.

TensorFlow
It is a free and open-source software library for machine learning.

Figure 32: Model Development and Production with Kubeflow



Source: Kubeflow (n.d.-b).

PyTorch
This is an open-source machine learning library based on the Torch library.

Scikit-learn
This is a machine learning library for the Python programming language.

Jupyter Notebooks
This is an application that allows users to create and share documents that contain live code, equations, visualizations, and narrative text.

As you can see from the diagram, Kubeflow integrates commonly used tools (**PyTorch**, **scikit-learn**, **Jupyter Notebooks**, etc.) and adds some components specific to KubeFlow. KubeFlow offers services for spawning, managing, and sharing Jupyter Notebooks; interactive data science; experimenting with ML workflows; using pods or containers in the cluster instead of locally; permitting the use of standardized images of development environments; and granting access to specific teams or individuals.

KubeFlow Pipelines offers a platform for building, deploying, and managing multi-step ML workflows based on Docker containers (Kubeflow, n.d.-b). It also offers support for building, deploying, and managing multi-step ML workflows based on Docker containers, using KubeFlow Pipelines. A pipeline is a description of an ML workflow written in Python code, showing all components and how they interact in a graph form, and including the definition of inputs and outputs of each component. Every component of a pipeline is a Docker image with self-containing code that performs one step of the pipeline.

Kubeflow offers several components that you can use to build your ML training, hyperparameter tuning, and serving workloads across multiple platforms, supporting the resolution of many of the issues we discussed in the previous sections (Kubeflow, n.d.-c).

SUMMARY

This unit introduced the characteristics of the development process of a machine learning model: exploratory, iterative, and nondeterministic. We explored some challenges related to data quality, exploration, development, testing, and deployment of machine learning models. We then approached the many challenges of bringing and keeping a machine learning system in production, some similar to the ones encountered to the traditional software development process, some slightly different, and some intrinsic to the nondeterministic and data driven nature of machine learning. We explored how versioning and testing is different in the field of machine learning, and how to adapt to potential issues.

We then covered the concepts related to deployment pipelines, including continuous integration, continuous deployment, and the DevOps approach. We introduced the concept of MLOps, the practice of collaboration between data scientists and other members of the production team. We covered virtualization, containers, and orchestration, with specific examples of Docker and Kubernetes. Finally, we reviewed how KubeFlow, built on top of Kubernetes, helps in the steps involved in supporting a machine learning system in production.

BACKMATTER

LIST OF REFERENCES

AgileAlliance. (n.d.-a). *Extreme programming*. https://www.agilealliance.org/glossary/xp/

AgileAlliance. (n.d.-b). *Pair programming.*https://www.agilealliance.org/glossary/pairing/

Ahmad, M. O., Markkula, J., & Oivo, M. (2013). Kanban in software development: A systematic literature review. *Proceedings of the 2013 39th Euromicro conference on software engineering and advanced applications* (SEAA) (pp. 9—16). IEEE. https://doi.org/10.1109/SEAA.2013.28

Altexsoft. (2018, February 23). *Extreme programming: Values, principles, and practices*. https://www.altexsoft.com/blog/business/extreme-programming-values-principles-and-practices/

AOE. (n.d.). *Agile methods & processes in companies*. https://www.aoe.com/en/agile.html

Arntz, M. (2020). *Iterative development vs. Agile*. Cprime. https://www.cprime.com/resources/blog/iterative-development-vs-agile

Aurum, A., & Wohlin, C. (2005). Requirements engineering: Setting the context. In A. Aurum & C. Wohlin (Eds.), *Engineering and managing software requirements* (pp. 1—15). Springer. https://wohlin.eu/rm_chapter05.pdf

Basu, A. (2015). *Software quality assurance, testing and metrics*. PHI Learning.

Baucherel, K. (2019). Scrum in rugby. *Pixabay*. https://pixabay.com/photos/rugbyscrum-heineken-cup-saracens-4498375/

Beck, K. (1999). Embracing change with extreme programming. *Computer, 32*(10), 70—77. https://doi.org/10.1109/2.796139

Beck, K. (2014). *Test-driven development: By example*. Addison-Wesley.

Bell, D. (2003, June 14). *UML basics: An introduction to the Unified Modeling Language*. IBM. https://developer.ibm.com/technologies/web-development/articles/an-introduction-to-uml/

Bhatia, S. (2020, August 24). *Procedural programming [definition].* hackr-io. https://hackr.io/blog/procedural-programming

Böckeler, B. (2020, January 15). *On pair programming*. Martin Fowler. https://martinfowler.com/articles/on-pair-programming.html

Boehm, B. W. (1988). A spiral model of software development and enhancement. *Computer*, *21*(5), 61—72. https://doi.org/10.1109/2.59

Boehm, B. (1996). Anchoring the software process. *IEEESoftware*, *13*(4), 73—82. https://ieeexplore.ieee.org/document/526834

Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *Unified Modeling Language user guide* (2nd ed.). Addison-Wesley.

Boschetti, M. A., Golfarelli, M., Rizzi, S., & Turricchia, E. (2014). A Lagrangian heuristic for sprint planning in Agile software development. *Computers & Operations Research*, *43*(1), 116—128. https://doi.org/10.1016/j.cor.2013.09.007

Breck, E., Polyzotis, N., Roy, S., Whang, S. E., & Zinkevich, M. (2019). Data validation for machine learning. *Proceedings of the 2nd SysML conference*. SysML. https://mlsys.org/Conferences/2019/doc/2019/167.pdf

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., ... Amodei, D. (2020). *Language models are few-shot learners*. arXiv. https://arxiv.org/abs/2005.14165v4

Carilli, J. F. (2013). *Transitioning to Agile: Ten success strategies* [Paper presentation]. Project Management Institute Global Congress.

Chapman, C. (n.d.). An overview of data-driven design. *Designers.* https://www.toptal.com/designers/ux/data-driven-design

Charette, R. N. (2005). Why software fails. *IEEE Spectrum*, *42*(9), 42—49. https://doi.org/10.1109/MSPEC.2005.1502528

Chen, L. (2015). Continuous delivery: Huge benefits, but challenges too. *IEEE Software, 32*(2), 50—54. https://doi.org/10.1109/MS.2015.27

Cohn, M. (2009). *Succeeding with Agile: Software development using Scrum.* Addison-Wesley.

Coram, M., & Bohner, S. (2005, April). The impact of Agile methods on software project management. *Proceedings of the 12th IEEE international conference and workshops on the engineering of computer-based systems (ECBS 2005)* (pp. 363—370). IEEE. https://doi.org/10.1177%2F875697281704800101

Cortellessa, V., Di Marco, A., & Inverardi, P. (2011). *Model-based software performance analysis.* Springer.

Coventry, T. (2015). Requirements management—planning for success! Techniques to get it right when planning requirements. *Proceedings of the PMI global congress 2015—EMEA*. Project Management Institute.

Covington, M. A. (2010, August 23). *First lecture on symbolic programming and LISP.* University of Georgia.

Darryl, T. (2003). *IBM acquires Rational*. eWeek. https://www.eweek.com/pc-hardware/ibm-acquires-rational

Department of Defense. (1988). *Military standard: Defense system software development*. (DOD-STD-2167A). U.S. Department of Defense.

DevOps. (2015). *Comparing DevOps to traditional IT: Eight key differences*.

Docker. (n.d.). *Docker homepage*. https://docs.docker.com/

Dutchguilder. (2007, October 16). Iterative development illustration. *Wikimedia Commons*. https://commons.wikimedia.org/wiki/File:Development-iterative.png

Dwork, C., Hardt, M., Pitassi, T., Reingold, O., & Zemel, R. (2012). *Fairness through awareness*. arXiv. https://arxiv.org/abs/1104.3913v2

Educba. (n.d.) *Introduction to Machine Learning (ML) lifecycle*. https://www.educba.com/machine-learning-life-cycle/

Evans, E. (2010). *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley.

Fabian, R. (2018). *Data-oriented design*. https://www.dataorienteddesign.com/dodbook/dodmain.html

Fowler, M. (2019). *Continuous delivery for machine learning.* Martin Fowler. https://martinfowler.com/articles/cd4ml.html

Fowler, M., & Highsmith, J. (2001). The Agile manifesto. *Software Development, 9*(8), 28—35. http://users.jyu.fi/~mieijala/kandimateriaali/Agile-Manifesto.pdf

Gade, K. (2019). *AI needs a new developer stack!* Fiddler. https://blog.fiddler.ai/2019/06/ai-needs-a-new-developer-stack/

Gannod, G. C., Eberle, W. F., Talbert, D. A., Cooke, R. A., Hagler, K., Opp, K., & Baniya, J. (2018). Establishing an Agile mindset and culture for workforce preparedness: A baseline study. *Proceedings of the 2018 IEEE frontiers in education conference (FIE)* (pp. 1—9). IEEE. http://dx.doi.org/10.15439/2019F198

Gherkin. (n.d.). *SpecFlow*. https://specflow.org/bdd/gherkin/

Git. (n.d.). *Git homepage*. https://git-scm.com/

Gitlab. (n.d.). *Create the ideal DevOps team structure*. https://about.gitlab.com/blog/2019/06/12/devops-team-structure/

Glinz, M., & Wieringa, R. J. (2007). Stakeholders in requirements engineering. *IEEE Software*, *24*(2), 18—20. https://doi.org/10.1109/MS.2007.42

Google. (n.d.). *MLOps: Continuous delivery and automation pipelines in machine learning*. https://cloud.google.com/solutions/machine-learning/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning

Gornik, D. (2017). *IBM rational unified process: Best practices for software development teams*. IBM. ftp://ftp.software.ibm.com/software/rational/web/whitepapers/2003/rup_bestpractices.pdf

Green, D. M. (2020, January 21). Pair programming: Benefits, tips & advice for making it work. *SitePoint*. https://www.sitepoint.com/pair-programming-guide/

Guigova, I. (2009, March 12). Approaches, styles, or philosophies in software development. *Codeproject.* https://www.codeproject.com/Articles/33992/Approaches-Styles-or-Philosophies-in-Software-Deve

Hammarberg, M., & Sunden, J. (2014). *Kanban in action*. Manning Publications.

Heck, P. (2020). Testing machine learning applications. *Fontys*. https://fontysblogt.nl/testing-machine-learning-applications/

Hegde, R. (2020, April 22). *How machine learning lifecycle is different from a software development lifecycle?* Medium. https://medium.com/@rajeshhegde/how-machine-learning-lifecycle-is-different-from-a-software-development-lifecycle-cacdc1fd0077

Highsmith, J. (2009). *Agile project management: Creating innovative products*. Pearson.

Hiraeth, M. L. (2018). Technologie Blaupause Haus Zeichnung [Blueprint of a residential house.]. *Pixabay*. https://pixabay.com/de/illustrations/technologie-blaupause-haus-zeichnung-3216744/

Humble, J., & Farley, D. (2015). *Continuous delivery: Reliable software releases through build, test, and deployment automation*. Addison-Wesley.

H2O. (n.d.). *Machine learning interpretability.*https://www.h2o.ai/explainable-ai/

IBM. (2020). *AI fairness 360*. http://aif360.mybluemix.net/

IEEE Computer Society. (1990). *IEEE standard glossary of software engineering terminology*. IEEE.

IEEE Computer Society. (1998). *Industry implementation of international standard ISO/IEC 12207: 1995 (ISO/IEC 12207)*. IEEE.

International Organization for Standardization. (2005). *Information technology—Open distributed Processing—Unified modeling language (UML) version 1.4.2. (ISO/IEC Standard No. 19501:2005)*. ISO. https://www.iso.org/standard/32620.html

International Organization for Standardization. (2017). Systems and software engineering —Software life cycle processes (ISO/IEC/IEEE Standard No. 12207:2017). *ISO*. https://www.iso.org/standard/63712.html

Jacobson, I., & Bylund, S. (2000). *The road to the unified software development process.* Cambridge University Press.

James, M., & Walter, L. (2010). *Scrum reference card*. CollabNet Inc. https://www.collab.net/sites/default/files/uploads/CollabNet_scrumreferencecard.pdf

Jira. (n.d.). *Jira homepage*. Atlassian. https://www.atlassian.com/software/jira

Jongerius, P., Offermans, A., Vanhoucke, A., Sanwikarja, P., & van Geel, J. (2013). *Get Agile! Scrum for UX, design & development*. BIS Publishers.

Kahootz. (2015, June 3). *Uber and Airbnb: The importance of Agile working in 2015*. https://www.kahootz.com/uber-and-airbnb-the-importance-of-agile-working-in-2015/

kanbanboard. (n.d.). *Kanbanboard homepage*. Github. https://github.com/kanboard/kanboard

Karpathy, A. (2017, November 11). *Software 2.0*. Medium. https://medium.com/@karpathy/software-2-0-a64152b37c35

Katalon. (2020). *What is end-to-end (E2E) testing? All you need to know*. https://www.katalon.com/resources-center/blog/end-to-end-e2e-testing/

Kim, M., Zimmermann, T., DeLine, R., & Begel, A. (2018). Data scientists in software teams: State of the art and challenges. *IEEE Transactions on Software Engineering*, *44*(11), 1024—1038. https://doi.org/10.1109/TSE.2017.2754374

Kotonya, G., & Sommerville, I. (1998). *Requirements engineering: Processes and techniques*. Wiley.

Kruchten, P. (2003). *The rational unified process: An introduction*. Addison-Wesley.

Kubeflow. (n.d.-a). *Kubeflow: An introduction to Kubeflow*. https://www.kubeflow.org/docs/about/kubeflow/

Kubeflow. (n.d.-b). *Overview of Kubeflow pipelines*. https://www.kubeflow.org/docs/pipelines/overview/pipelines-overview/

Kubeflow. (n.d.-c). *Kubeflow overview.*https://www.kubeflow.org/docs/started/kubeflow-overview/

Kubernetes Documentation. (2020). *Concepts*. Kubernetes. https://kubernetes.io/docs/concepts/

Lage Junior, M., & Godinho Filho, M. (2010). Variations of the Kanban system: Literature review and classification. *International Journal of Production Economics*, *125*(1), 13—21. https://doi.org/10.1016/j.ijpe.2010.01.009

Leffingwell, D. (2007). *Scaling software agility*. Pearson.

Lithmee. (2020). *What is the difference between Agile and iterative*. Pediaa. https://pediaa.com/what-is-the-difference-between-agile-and-iterative/

Macaulay, L. A. (2012). *Requirements engineering.* Springer.

Masson, R., Iosif, L., MacKerron, G., & Fernie, J. (2007). Managing complexity in Agile global fashion industry supply chains. *The International Journal of Logistics Management*, *18*(2), 238—254. https://doi.org/10.1108/09574090710816959

Miller, G. G. (2001). The characteristics of Agile software processes. *Proceedings of the international conference on technology of object-oriented languages* (pp. 0385—0387). IEEE. http://faculty.salisbury.edu/~xswang/research/papers/serelated/agile/12510385.pdf

Miller, R. (2003). *Practical UML – A hands-on introduction for developers*. Borland Developer Network.

Mitchell, I. (2015). Scrum framework. *Wikimedia Commons*. https://commons.wikimedia.org/wiki/File:Kanban_principles.jpg

North, D. (2006, June 5). *Behavior modification*. Stickyminds. https://www.stickyminds.com/better-software-magazine/behavior-modification

North, D. (n.d.). *Introducing BDD*. Dan North & Associates. https://dannorth.net/introducing-bdd/#translations

Null, C. (2020). *Infrastructure as code: The engine at the heart of DevOps*. TechBeacon. https://techbeacon.com/enterprise-it/infrastructure-code-engine-heart-devops

oodesign. (n.d.). *Design principles*. https://www.oodesign.com/design-principles.html

OpenSource. (n.d.). *Virtualization*. https://opensource.com/resources/virtualization

Patel, C. (2020). *DevOps best practices*. DZone. https://dzone.com/articles/devops-best-practices

Petersen, K., Wohlin, C., & Baca, D. (2009). The waterfall model in large-scale development. *Proceedings of the international conference on product-focused software process improvement* (pp. 386—400). Springer.

Poskitt, C. (2018, June 5). *4 examples of businesses who have made Agile working work.* Turbine. https://www.turbinehq.com/blog/4-examples-agile-working

Pouloudi, A., & Whitley, E. A. (1997). Stakeholder identification in inter-organizational systems: Gaining insights for drug use management systems. *European Journal of Information Systems*, *6*(1), 1—14. https://doi.org/10.1057/palgrave.ejis.3000252

Project Management Institute. (2008). *A guide to the project management body of knowledge (PMBOK guide)* (4th ed.). PMI.

Redhat. (n.d.). *What is container orchestration?* https://www.redhat.com/en/topics/containers/what-is-container-orchestration

Rollins, S. (2018, January 20). Functional programming principles every imperative programmer should use. *DZone*. https://dzone.com/articles/functional-programming-principles-every-imperative

Rouse, M. (2020). *Object-oriented programming (OOP).* Techtarget. https://searchapparchitecture.techtarget.com/definition/object-oriented-programming-OOP

Royce, W. W. (1970). Managing the development of large software systems. *Proceedings of IEEE WESCON* (pp. 328—338). IEEE. http://www-scf.usc.edu/~csci201/lectures/Lecture11/royce1970.pdf

Sakolick, I. (2020). *What is CI/CD? Continuous integration and continuous delivery explained*. InfoWorld. https://www.infoworld.com/article/3271126

Schwaber, K., & Beedle, M. (2002). *Agile software development with Scrum*. Prentice Hall.

Sebesta, R. W. (1996). *Concepts of programming languages.* Addison-Wesley.

Shams-Ul-Arif, Q. K., Khan, Q., & Gahyyur, S. A. K. (2010). Requirements engineering processes, tools/technologies & methodologies. *International Journal of Reviews in Computing*, *2*(6), 41—56.

Sharvit, Y. (2020, September 29). *Principles of data oriented programming*. Klipse. https://blog.klipse.tech/databook/2020/09/29/do-principles.html

Shenhar, A. J., & Dvir, D. (2007). Project management research—The challenge and opportunity. *Project Management Journal*, *38*(2), 93—99. https://doi.org/10.1177%2F875697280703800210

Sherehiy, B., Karwowski, W., & Layer, J. K. (2007). A review of enterprise agility: Concepts, frameworks, and attributes. *International Journal of Industrial Ergonomics*, *37*(5), 445—460. https://doi.org/10.1016/j.ergon.2007.01.007

Sims, C., & Johnson, H. L. (2012). *Scrum: A breathtakingly brief and Agile introduction*. Dymaxicon.

Spring. (n.d.). *Aspect oriented programming with Spring*. https://docs.spring.io/spring-framework/docs/2.0.x/reference/aop.html

Srivastava, A., Bhardwaj, S., & Saraswat, S. (2017). SCRUM model for Agile methodology. *Proceedings of the2017 international conference on computing, communication and automation (ICCCA)* (pp. 864—869). IEEE. https://doi.org/10.1109/CCAA.2017.8229928

Stellman, A., & Greene, J. (2005). *Applied software project management*. O'Reilly.

Stephens, R. (2015). *Beginning software engineering*. Wiley.

Sugimori, Y., Kusunoki, K., Cho, F., & Uchikawa, S. (1977). Toyota production system and Kanban system materialization of just-in-time and respect-for-human system. *The International Journal of Production Research, 15*(6), 553—564. https://doi.org/10.1080/00207547708943149

Sutherland, J. (2004). Agile development: Lessons learned from the first Scrum. *CutterAgile Project Management Advisory Service: Executive Update*, *5*, 1—4.

Takacs, P. Z. (2018, February 25). *Scrum Events*. Scrumthing. https://scrumthing.org/2018/02/25/scrum-events/

Takeuchi, H., & Nonaka, I. (1986). The new product development game. *Harvard Business Review, 64*(1), 137—146. https://hbr.org/1986/01/the-new-new-product-development-game

Tozzi, C. (2017). *Understanding why Docker is so popular*. Container Journal. https://containerjournal.com/features/understanding-why-docker-popular/

Tuple. (n.d.). *Pair programming guide*. https://tuple.app/pair-programming-guide

Tutorialspoint. (n.d.). *Software engineering overview*. https://www.tutorialspoint.com/software_engineering/software_engineering_overview.htm

UCF. (n.d.). *Major programming paradigms*. University of Central Florida. http://www.eecs.ucf.edu/~leavens/ComS541Fall97/hw-pages/paradigms/major.html

Waldner, J. (1992). Kanban principles. *Wikipedia Commons.* https://commons.wikimedia.org/wiki/File:Kanban_principles.svg

Wells, D. (1999). The rules of extreme programming. *ExtremeProgramming*. http://www.extremeprogramming.org/rules.html

Wei, J., & Field, M., (2004). *A case study: Using IBM rational unified process as the methodology framework*. IBM. https://www.ibm.com/developerworks/rational/library/4474.html

Wikipedia. (2020a). *Data wrangling*. https://en.wikipedia.org/wiki/Data_wrangling

Wikipedia. (2020b). *Transformational leadership*. https://en.wikipedia.org/wiki/Transformational_leadership

Wikipedia. (2020c). *Hyper-V.* https://en.wikipedia.org/wiki/Hyper-V

Yau, A., & Murphy, C. (2013). *Is a rigorous Agile methodology the best development strategy for small scale tech startups?* (Technical report no. MS-CIS-13-01). University of Pennsylvania. https://repository.upenn.edu/cgi/viewcontent.cgi?article=2025&context=cis_reports&httpsredir=1&referer=

Zhang, J. M., Harman, M., Ma, L., & Liu, Y. (2019). *Machine learning testing: Survey, landscapes and horizons*. arXiv. https://arxiv.org/abs/1906.10742v2

Ziolkowski, D. (2020, November 12). *What is software provisioning? A detailed introduction*. Plutora. https://www.plutora.com/blog/software-provisioning-introduction

Zuill, W. (2014, September 1). Mob programming. *Life, Liberty, and the Pursuit of Agility.* http://zuill.us/WoodyZuill/category/mobprogramming/

LIST OF TABLES AND FIGURES

IU Internationale Hochschule GmbH
IU International University of Applied Sciences
Juri-Gagarin-Ring 152
D-99084 Erfurt

Mailing Address
Albert-Proeller-Straße 15-19
D-86675 Buchdorf

media@iu.org
www.iu.org

Help & Contacts (FAQ)
On myCampus you can always find answers to questions concerning your studies.