Software Engineering Cheatsheet

Edoardo Riggio

June 5, 2022

Software Engineering - S.P. 2021
Computer Science
Università della Svizzera Italiana, Lugano

Contents

1 Software Development Life Cycle

1.1 Waterfall Model

This is a family of models. They identify phases and activities, forcing a linear progression from a phase to the next. No return is allowed (i.e. you cannot go from a phase to one before it).

1.2 Royce's Waterfall

1.2.1 Feasibility Study

This is a cost/benefits analysis.

It determines whether the project should be started, explore possible alternatives and needed resources.

Output → **Feasibility Study Document** - This document contains a preliminary problem description, scenarios with alternatives, and costs for these alternatives.

1.2.2 Requirement Analysis and Specification

This is a domain analysis.

It is needed in order to identify requirements and derive specifications for the software. It requires an interaction with the user, and an understanding of the properties of the domain.

Output → **RASD** (Requirements Analysis and Specification Document)

1.2.3 Design

Defines the software architecture.

It determines all the aspects of the software, such as: components/modules, relations among components and interactions among components. It also enables concurrent development and separates responsibilities.

Output → **Design Document**

1.2.4 Coding and Unit Testing

It is used to produce and test code.

Each module is implemented using the chosen programming language. Each module is then tested in isolation by the module's developer.

Output → **Software** - Composed of classes and modules.

1.2.5 Integration and System Test

All of the modules are integrated.

Modules are integrated into (sub)systems and all of the latter are tested. A complete system test is needed in order to verify the overall properties of the software. Sometimes **alpha tests** and **beta tests** are used.

Output → **Software** - Integrated modules.

1.2.6 Deployment

The goal is to distribute the application and manage the different installations and configurations at the clients' sites.

1.2.7 Maintenance

These are all the changes that are done after the delivery. It often is more than 50% of the total cost.

Maintenance can be of three types:

- **Corrective Maintenance**

  It is made up of diagnosing and fixing errors, which are possibly found by users.

- **Adaptive Maintenance**

  Modify the system to cope with changes in the software environment.

- **Perfective Maintenance**

  Implement new or changed user requirements.

1.3 Spiral Model

In a spiral model, **prototypes** are produced at each new spiral. Prototypes are useful because they are an approximation of the model of the application. There exist two types of prototypes:

- **Throw-Away**

  There is little to no reuse of the components of the prototype.

- **Evolutionary**

  The components of the prototype are reused even in the final product.

1.4 Boehm's Spiral Model

1.4.1 Determine Goals, Alternatives and Constraints

This enables to identify specific objectives, alternatives and constraints for that specific phase.

1.4.2 Evaluate Alternatives and Risks

Access each alternative in order to reduce the potential risks.

1.4.3 Develop and Test

Artifacts are produced for the next phase. This also includes code.

1.4.4 Plan

Review the project and plan the next phase of the spiral.

1.5 Synchronize and Stabilize

What people are doing as individuals and as members of parallel teams is continually **synchronized**, and the product increments are periodically **stabilized** as the project proceeds.

1.5.1 Planning Phase

This phase is itself divided into three more phases:

- **Vision Statement**

  The product and program management create a vision statement defining the goals for the new product. Product features based on costumer input are identified and prioritized.

- **Specification Document**

  This is written by the program management and the developer teams. It defines the functional features, architectural issues, and interdependent components. It does not cover all the details of each feature, and does not lock the feature set of the project.

- **Schedule and Team Formation**

  The program management defines the schedule and arranges the teams. The teams are small and have a 1:1 ratio between developers and testers. The project is divided into sequential sub-projects containing priority-ordered features. The schedule is divided into milestone cycles.

1.5.2 Development Phase

All teams have to go through a complete cycle of development, which includes: development, integration, testing and fixing problems.

Teams and individuals work in parallel, and revise the feature set as they learn.

Whenever developers commit code, regression testing is automatically performed. Debugging and synchronization between teams happen on a daily basis.

1.5.3 Stabilization Phase

In this phase program managers coordinate and monitor the customer's feedback. Developers perform final debugging and code stabilization.

The **golden master disks and documentation** are prepared for final release.

1.6 Extreme Programming

1.6.1 Release Planning

This is made up of four different phases:

- **User Stories**

  User stories are written by customers as things that the system is expected to accomplish. They are short, three sentences format of written text without technicalities.

- **Architectural Spike**

  Find the simplest system metaphor which allows to explain the system to new people without resorting to huge documents. The design must be kept as simple as possible. Naming of classes and methods must be kept simple and consistent.

- **Project Velocity**

  This is a measure of how much work is getting don on the project. It is also used as an indicator of how many user stories can be done before a deadline.

- **Release Plan**

  The development team estimates the ideal developing time needed for each user story. The customer must indicate which of the user stories must have higher priority.

1.6.2 Iteration

This is made up of several phases:

- **Iteration Planning**

  During iteration planning meetings, customers select the user stories from the release plan. User stories and failed tests are broken down into programming tasks. These tasks are done by developers.

- **Stand-up Meetings**

  These meetings are scheduled at the beginning of every day.

- **CRC Cards**

  Class, Responsibilities and Collaboration should be used to design systems. **Responsibilities** represent anything an object knows or does, while **collaborators** represent the classes to collaborate with in order to fulfill the responsibilities. Individual cards are used to represent objects.

- **Unit Test First**

  Unit tests are created before any coding activity. This is known as **Test Driven Development** (TDD).

- **Pair Programming**

  All code is created by two people working together at a single computer. One of the couple is the **navigator**, which reviews the code, and one is the **driver**, which writes the code.

- **Continuous Integration**

  Developers should be integrating code every few hours. All unit tests must pass in order to detect any compatibility problem early.

1.6.3 Acceptance Tests

Here we have two phases:

- **Creation**

  Acceptance tests are created from user stories. The customer specifies some scenarios in which to test for a user story. These are used as regression tests prior to production release.

- **Customer Approval**

  Customers are responsible for verifying the correctness of the acceptance tests. Customers decide which failed tests are of highest priority. A user story is not considered completed until it passes all acceptance tests.

1.7 Scrum

Scrum is a framework that has **three roles** – product owner, scrum master and scrum team, **three ceremonies** – sprint planning, daily scrum meetings and sprint review, and **three artifacts** – product backlog, string backlog and burndown chart.

1.7.1 Roles

There are three different roles in Scrum:

- **Product Owner**

  This could be a customer representative or a customer proxy. He devises the vision, business plan and releases. He is also responsible for defining the features of the product prioritized on the return of investment. He also builds the product backlog and leads the scrum planning.

- **Scrum Master**

  Does whatever it's necessary to help the team be successful, by supporting the practices of Scrum. He manages conflicts within the team, identifies and prioritizes impediments, and implements remediation plans. The Scrum master is not a project manager, he does not assigning tasks to people.

- **Scrum Team**

  A team is typically composed by 5 to 10 people. These teams are cross functional and include all the expertise needed. Teams are also self-organizing. They set their sprint goals, and do everything within the boundaries of a project to achieve them.

1.7.2 Ceremonies

There are three ceremonies in Scrum:

- **Sprint Planning**

  This takes place at the beginning of each sprint, and sets the goals of the sprint. The project owner and the team review the product backlog, and discuss goals. The items selected are then broken down into tasks and added to the sprint backlog.

- **Daily Scrum Meetings**

  These are short, 15 minutes meetings, where tams report what they have done and what they are going to do. After the meeting, the Scrum master updates the burndown chart with the data of the completed tasks.

- **Sprint Review**

  The team demos the work they have done to everyone in the three roles. Presentations are not allowed.

1.8 Agile

The 12 rules of the Agile manifesto are the following:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to a shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity – the art of maximizing the amount of work not done – is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

1.9 Hacker Way

In the hacker way, command is decentralized. This means that subordinate leaders must use their initiative in order to accomplish tasks supporting their senior's intent. The goals of a layer are set by the immediate upper layer, and this is a continuous feedback loop. Three levels of command are:

- **Senior Leadership**

  It sets the strategic long-term direction, and ensures that the feedback engine runs smoothly. It is also responsible for the overall people management, meaning acquiring new capabilities, expelling obsolete capabilities and react to external changes.

- **Management**

  It provides operational context (goals) by linking strategy with tactics. They also devise the constraints that are aligned with the strategy of the company under which the developers operate. There is trust towards developers, and it provides them with all the software and hardware needed.

- **Developers**

  They are responsible for the tactical level of operations. The teams are composed of about 10 developers.

The decision is data-driven, and every layer must have a data scientist to help analyze data.

The philosophy of this method is to deploy, and if it fails, fix it. Some companies release versions of the product only to a subset of the users in order to test it. If the version is broken, they simply rollback to the previous working version.

2 Requirement Engineering

Requirement engineering analyses the context in which the problem to be solved lies, and finds out what needs to be developed, purchased or installed in order to fix the problem.

It is divided into two overlapping sets. The first is the **problem world's phenomena** set, and the second is the **machine solutions' phenomena** set. The intersection of these two sets is called **shared phenomena**.

There is also the distinction to be made between **system-as-is**, which is the system as it exists before the machine is built into it, and the **system-to-be**, which is the system as it should be when the machine will be built and operated in it.

- **System-As-Is**

  Contains the problems, opportunities and domain knowledge.

- **System-To-Be**

  Contains the who, what and why.

2.1 Categories of Requirements

2.1.1 Environmental Phenomena

This set is subdivided into:

- **Domain Properties**

  These are descriptive statements about the problem world that is expected to hold regardless of how the system will behave.

- **Assumptions**

  Statements that need to be satisfied by the environment and formulated in terms of environmental phenomena.

- **System Requirements**

  Prescriptive statements to be enforced by the **software-to-be**, possibly in cooperation with other system components, and formulated in terms of environmental phenomena. They are divided into:

  - **Functional Requirements**

    They define the functional effects that the **software-to-be** is required to have in its environment. They may also refer to environmental conditions under which such operations should be applied. They are also known as **features**.

  - **Non-Functional Requirements**

    They define constraints on the way the **software-to-be** should satisfy functional requirements or on the way it should be developed. There could be several such requirements, such as:

    * **Quality Requirements**
    * **Compliance Requirements**
    * **Architectural Requirements**
    * **Development Requirements**

2.1.2 Shared Phenomena

These are:

- **Software Requirements**

  Prescriptive statements to be enforced solely by the **software-to-be** and formulated only in terms of shared phenomena.

2.2 Requirements Engineering Process

2.2.1 Requirements Elicitation

This is the discovery of candidates requirements and assumptions that will shape the **system-to-be** based on the weaknesses of the **system-as-is**. It requires a preliminary phase of knowledge acquisition and discovery about the application domain, the organization and the **system-as-is**. This phase can be of two types:

- **Artifact Driven**

  Collect, read and synthesize relevant documentation about the **system-as-is**. Submit a list of specific questions to selected stakeholders with a list of possible answers and a brief context. Illustrate a typical sequence of interactions among system components that meet an implicit objective with concrete examples. Show positive/negative and normal/abnormal scenarios. Show mock-ups and prototypes of the final product to help understand and clarify its features.

- **Stakeholder Driven**

  Organize interviews with specific stakeholders to target the information we want to acquire. Interviews can be **structured**, which follow a sequence of pre-defined questions, or **unstructured**, where there is only a free informal discussion with the stakeholder about the **system-as-is**.

2.2.2 Requirements Evaluation

This step is for the evaluation of the elicited requirements, in order to obtain a set of low-risk, conflict-free requirements that stakeholders agree on. This step too is divided into several parts:

- **Conflict Handling**

  Negotiation may resolve conflicts there may be between stakeholders.

- **Risk Analysis**

  What is known as **functional risk** concerns the inability of the product to deliver the required services. **Non-functional risk**, on the other hand, concerns the inability of the product to deliver the required quality.

- **Prioritization**

  Some requirements often need to be prioritized. Unexpected circumstances might force re-planning of the development. Requirements with lower priority should be weakened or even dropped if necessary.

2.2.3 Requirements Specification

This is to detail, structure and document the agreed characteristics of the **system-to-be** as they emerge from the evaluation activity. The output docu-

ment of this phase is the **requirements document**, which contains objectives, domain properties, assumptions... This document should be easy to understand. These requirements can be of three different types:

- **Informal**

  Make use of **natural language** to document all of the agreed requirements. Ambiguity and noise are inherent to natural language, and may lead to different interpretations.

- **Semi-Formal**

  Make use of diagrammatic notations to substitute or complement the natural language specifications. All items and relationships and items are declared formally. **UML** (Unified Modeling Language) is the industry standard for documenting object oriented systems.

- **Formal**

  Make use of formal notations that have a defined syntax, semantic and proof theory, in order to describe and derive requirements.

2.2.4 Requirements Consolidation

In this steps we detect defects, report them, analyze their causes, and undertake the appropriate actions in order to fix them and ensure quality of the requirements. There are four steps to this process:

- **Inspection Planning**

  Determine the size and members of the inspection team, schedule and formats of the reports.

- **Individual Reviewing**

  Each inspector reads the requirements document or parts of it to look for defects.

- **Defect Evaluation at Review Meetings**

  The defects found by each inspector are collected and discussed to recommend appropriate actions.

- **Requirements Document Consolidation**

  The requirements document is revised to address all of the concerns expressed in the inspection report.

3 Testing

There are several types of failure that can affect code, such as:

- **Transient**

  It occurs only with certain inputs.

- **Permanent**

  It occurs with all inputs.

- **Recoverable**

  The system can recover automatically.

- **Unrecoverable**

  The operator needs to intervene in order to recover from failure.

- **Non-Corrupting**

  The failure does not corrupt data.

- **Corrupting**

  The failure corrupts data.

There are also several testing categories, such as:

- **Unit Testing**

  It tests individual components (methods, classes...).

- **Module/Integration Testing**

  A collection of related components tested as a group.

- **Sub-System/System Testing**

  This is a higher-level integration testing, from sets of modules to the whole system, including functional and non-functional requirements.

- **System vs Components**

  Testing a system's functionality might be more important than testing its components.

- **Old vs New** Testing old capabilities is more important than testing new capabilities. This is also known as **regression testing**.

- **Typical vs Boundaries**

  If resources are limited, focus on testing the typical scenarios.

- **Deterministic and Repeatable**

  Devise tests that are deterministic and repeatable, for example by using frameworks like jUnit.

- **Cost/Benefit Trade Off**

  Test is extra work, but it pays off in debugging and maintenance.

- **Top-Down Testing**

  Start testing sub-systems. Similarly test modules.

- **Bottom-Up Testing**

  Start by testing unit and modules.

- **Black Box Testing**

  Write tests according to the specifications.

- **White Box Testing**

  Write tests according to the structure of the implementation.

- **Gray Box Testing**

  Drive tests by both Back and White Box aspects.

3.1 Black Box Testing

In order to test a unit, it needs a description of what it should do. This is done in order to derive meaningful inputs and check correct outputs.

A single unit test is divided in the following steps:

1. **Arrange/Setup**

   It defines the software preconditions (such as needed objects, parameters and resources).

2. **Act/Exercise**

   Invoke the method under test.

3. **Assert/Verify**

   Check that the results/effects of the method match the expected ones.

4. **Teardown**

   Release resources, dispose of unneeded objects.

3.1.1 Input Partitioning

Input partitioning is done in order to try and find all of the reasonable equivalence classes, check all of the combinations and document all tests.

Boundary conditions are checked by paying attention to primitive types.

Parameterized testing is used in case tests have the same setup/exercise/assertion and you're just testing for different values. Write down a static provider for the arguments, and then specify a Method Source for the actual test.

Inputs have to be partitioned into classes of behavior which are similar, and pick a representative – which represents the behavior of the whole class.

3.2 Mocking

The purpose of mocking is to isolate the unit being tested and not the behavior or state of external dependencies.

Normally, mocking frameworks enables the developer to inject mock objects instead of real dependencies. Mocking objects imitate real dependencies, but only on a limited set of scenarios.

The main **issue** with mocking is how to express testing scenarios without the need of writing too much code. This is solved by using fluent APIs.

3.3 White Box Testing

Here testing units are treated as white boxes. This means that the code can be examined in order to generate test cases. Test cases need to maximize the coverage of that structure.

3.4 Coverage

The idea is to use the structure to drive the test case generation. There are several ways to cover code:

- **Statement Coverage**

  Devise the minimal amount of test cases that will cover every statement in the code at least once.

- **Decision Coverage**

  Also known as **branch coverage**. Devise the minimal amount of test cases that will cover every decision in the control flow of the program at least once.

- **Condition Coverage**

  Devise the minimal amount of test cases that will cover every compound condition – in boolean sub-expressions – at least once.

4 Software Metrics and Quality

A **software metric** is any type of measurement which relates to a software system, process or related documentation. There are two types of metrics:

- **Direct Metric**

  It is measured directly on an observed process or artifact.

- **Indirect Metric**

  It is calculated from other metrics – direct or indirect.

A **metric** is a function mapping an attribute of an entity onto a symbol in the mathematical set. A **metric value** is the value of such function assigned to a particular attribute. By manipulating the symbols in the range we can draw conclusions about the attributes in the domain.

A good metric is both **valid** – it measures what it is intended to measure, and **reliable** – it yields consistent results. Other desirable properties of a metric are:

- **Objective**
- **Precise**
- **Intuitive**
- **Robust**
- **Computable**
- **Economically Practical**

4.1 Metrics for Cost Estimation and Quality Evaluation

Cost estimation and planning are closely related activities. The goals of such activities are:

- To establish a budget for a software project.
- To provide a means of controlling project costs.
- To monitor progress against budget.
- To establish a cost database for future estimation.

The cost is estimated as a mathematical function of product, project and process attributes whose values are estimated by project managers. The function is derived from a study of historical costing data. A measured-based estimation is composed of the following steps:

- **Measure**

  Develop a system model and measure its size.

- **Estimate**

  Determine the effort with respect to an empirical database of measurements from similar projects.

- **Interpret**

  Adapt the effort with respect to a specific Development Project Plan.

The use of **LOC** (Lines Of Code) as an attribute for cost estimation is not the best. All measurements that are based on volume over time are flawed because they do not take in account quality.

In order to compute a quality evaluation, the following metrics are used.

- **Class Size Metrics**

  NOM (Number Of Methods), NIA (Number of Instance Attributes) and NCA (Number of Class Attributes).

- **Method Size Metrics**

  LOC (Lines Of Code), NOS (Number Of Statements), NOA (Number Of Arguments), NOI (Number Of Invocations).

- **Inheritance Metrics**

  HNL (Hierarchy Nesting Level), NOC (Number Of immediate Children), NMI (Number of unmodified Methods Inherited) and NMO (Number of Overridden Methods).

- **Coupling Between Objects**

  It is the number of classes to which a given class is coupled. It is to be interpreted as the number of other classes a class requires in order to compile.

- **Lack of Cohesion Methods**

  The **cohesion** is the degree to which the elements inside a module belong together. The **LCOM** is in function of the number of disjoint sets of local methods. This measurement is **low** if all methods access all attributes, and is **equal to 1** if each method uses one given attribute.

- **Defect Density**

  The number and size of known defects – it's a time-based count.

4.2 Object-Oriented Metrics

Object-oriented metrics can be divided into three main categories:

- **Inheritance**
  - **ANDC**

    The Average Number of Derived Classes metric describes the average of derived classes.
  - **AHH**

    The Average Hierarchy Height metric is an average depth of the inheritance hierarchy.
- **Size**
  - **NOP**
  - **NOC**
  - **NOM**
  - **LOC**
  - **CYCLO**
- **Communication**
  - **NOM**
  - **CALLS**
  - **FANOUT**

    It counts the types referenced classes and interfaces. It only counts those types that are not part of the same Inheritance Branch.

4.2.1 God Class

The characteristics of a God Class are that it tends to centralize the intelligence of the system, to do everything and to use data from small data-classes. This type of class is large and has a lot of complex behavior.

In order to detect a God Class, use can use the following detection strategies:

- The class uses directly more than a few attributes of other classes.
- The functional complexity of this class is very high.
- The class cohesion is low.

4.2.2 Envious Method

An Envious Method is more interested in the data given to it from a handful of other classes.

In order to detect an envious method, we use the following detection strategies:

- The method uses directly more than a few attributes of other classes.
- The method uses far more attributes of other classes than its own.
- The used "foreign" attributes belong to very few other classes.

4.2.3 Data Classes

These Data Classes are "dumb" data holders.

In order to detect an envious method, we use the following detection strategies:

- The interface of the class reveals data rather than offering services.
- The class reveals many attributes and is not complex. This measures by either
  - There are more than a few public data elements.
  - The complexity of the class is not high

  or
  - The class has many public data elements.
  - The complexity of the class is not very high.

4.3 Code Smells and Technical Debt

A **code smell** is a surface indication that usually corresponds to a deeper problem in the system. The characteristics of code smells are the following:

- They are easy to spot.
- Code smells are not problems per se, they are often indicators of problems.
- Some code style violations may be considered as code smells.
- Design flaws are also considered code smells.

Every time we ignore a code smell, we accumulate **technical debt**. It indicates the extra effort needed in order to add a new feature, fix an existing one...

Technical debt has also **interest**. This is measured by computing the difference in time it takes to do something with technical debt and without.

Technical debt has a **human cost**. This means that if the technical debt is too high, it affects the team's morale, frustrates developers, lowers productivity and generates fights.

4.3.1 Code Review

Code review is a systematic examination of computer source code.It is intended to find mistakes overlooked in software development, improving the overall quality of software.

Software inspection follows a strict sequential process, and is applied to any activity – from requirement specification to programming, These are the steps of code review:

- **Planning**

  Materials to be inspected must meet the inspection entry criteria.

- **Overview**

  Assignment of inspection roles, and education of participants on the materials under review. There are several roles, such as: Author, Reader, Testers and Moderator.

- **Preparation**

  The participants review the item to be inspected and the related material to prepare the meeting.

- **Inspection Meeting**

  All defects are reported by the participants.

- **Rework**

  Defects found during inspection are resolved by the author.

- **Follow-up**

  All defects found in the meeting should be corrected. 9:30

5 Docker

Docker supports the construction, distribution and execution of independently deployable units.

A **Docker Image** is an immutable, layered collection of files that bundle together all the essential components required to configure and run a fully operational environment.

A **Docker Container** is a runnable instance of an image. A container can be created out of an image, and can also be started and stopped. Docker containers are not virtual machines, they are virtual environments.

Containers are an abstraction at the application layer that packages code and dependencies together. Multiple containers can run on the same machine and share the same OS kernel. Each container runs as an isolated process in the user space.

5.1 Dockerfile

A docker image can be built by reading the instructions from a **Dockerfile**. A Dockerfile contains all of the commands needed in order to assemble an image. The format of Dockerfile instructions is:

```
# Comment
INSTRUCTION:arguments
```

5.1.1 FROM

Docker images might depend on other docker images. Whatever is declared in the environment of an image will be available in the Dockerfile.

```
FROM <image>[@<tag>] [AS <name>]

# For example
FROM ubuntu:16.04
```

5.1.2 ADD and COPY

The **ADD** and **COPY** instructions copy files from a source to a destination within the container. If an image is built on a Linux-based system, then ownership of files can be set too.

```
ADD | COPY [-chown=<user:group>] <src> <dest>

# For example
# The following two lines have identical behavior
ADD my_app.jar /myapp/app.jar
COPY my_app.jar /myapp/app.jar

# Copy or add all files that start with "hom" to /mydir
```

```
ADD hom* /mydir/
COPY hom* /dir/
```

ADD can also handle URLs and can automatically decompress archives if set as the source.

```
# Creates the file /foobar
ADD http://example.com/foobar /

# The contents of the archive are unpacked into /tmp/
ADD foo.tar.gz /tmp/
```

5.1.3 RUN and ENV

The **RUN** command executes a shell command during the image construction. The effects of the command will be available in the next step of Docker.

```
RUN <command>
RUN ["executable", "param1", "param2"]
```

By default the shell-form commands are executed via `/bin/sh` on Linux.

```
RUN echo $HOME
RUN ["/bin/sh", "-c", "echo $HOME"]
```

Environment variables can be defined and used all along the Dockerfile.

```
ENV <key> <value>
ENV <key>=<value>

# For example
ENV HOME /home/username
ENV HOME=/home/username
```

Here is an example that uses both **RUN** and **ENV**:

```
FROM openjdk:latest

ENV MAVEN_OPTS=-Xmx256M

RUN apt-get update > /dev/null
RUN apt-get install -y maven git

RUN git clone https://github.com/pdurbin/maven-hello-world.git

RUN cd maven-hello-world/my-app; mvn compile
```

5.1.4 EXPOSE

Docker containers are isolated and do not expose ports to the host machine. The **EXPOSE** instruction functions as documentation about which ports are intended to be published.

```
EXPOSE <port(s)>

# For example for https, http and ssh
EXPOSE 443 80 22
```

In order to actually expose the ports, the following command must be ran:

```
docker run -p <host_port>:<container_port>
```

5.1.5 ENTRYPOINT

An entry point allows to configure a container that will run as executable. This is invoked when `docker run` is executed.

```
ENTRYPOINT <command>

# For example
ENTRYPOINT ["executable", "param1", "param2"]
```

The following is an example Dockerfile for a maven executable:

```
FROM openjdk:latest

RUN apt-get update > /dev/null
RUN apt-get install -y maven git

RUN git clone https://github.com/pdurbin/maven-hello-world.git

RUN cd maven-hello-world/my-app; mvn compile

ENTRYPOINT ["java", "-cp", "maven-hello-world/my-app/target/classes
    ", "com.mycompany.app.App"]
```

5.1.6 VOLUME

Docker containers do not persist data by default. Whenever a container is removed, all data is lost. Docker, however, supports two ways of declaring volumes to persistent data:

- **Volume**

  It creates a volume in the docker area, and mounts it to the specific mount

point. It can be specified in the Dockerfile.

```
VOLUME <mount-point>
```

- **Bind Mount**

  It binds a mount point in the docker container to the filesystem of the host machine. It's specified when running a container.

```
docker run -v <host_machine_path>:<container_mountpoint>
```

5.2 Building a Dockerfile

In order to build a Dockerfile, we use:

```
docker build <path> -t <image_name>:<tag>
```

5.3 Docker Registry

A **Docker Registry** allows to store and retrieve newly created images. The main public directory is Docker Hub. A docker image can be either pushed to a public docker registry, or to a private registry (such as GitLab).

5.4 Docker Compose

Docker, through docker-compose, supports the construction and life cycle of multi-container Docker applications.

5.4.1 Structure

A docker-compose.yml file defines a set of services offering some functionalities.

```
version: "3.8"

services:
  backend:
    ...
  db:
    ...
  ...
...
```

5.4.2 Networks

The docker-compose.yml file may also define one or more named internal virtual networks to support communication between containers.

```
version: "3.8"

services:
  ...
networks:
  network:
    driver: bridge
```

5.4.3 Services

A service can be specified with a build section.

```
version: "3.8"

services:
  backend:
    build: .
    container_name: backend
    networks: [network]
    depends_on: [db]
    ...
...
```

Or with an image, such as:

```
version: "3.8"

services:
  db:
    image: postgres:12-alpine
    container_name: db
    networks: [network]
    networks:
    ...
  ...
...
```

5.4.4 Ports

By default, containers communicate only through the internal network. In order to expose public services, you need to expose specific ports.

```
version: "3.8"

services:
```

```
4   backend:
5     build: .
6     container_name: backend
7     networks: [network]
8     depends_on: [db]
9     ...
10    ports:
11      - 8080:8080
12 ...
```

5.4.5 Environment

The environment section is used to pass environment variables to containers.

```
1  version: "3.8"
2
3  services:
4    backend:
5      container_name: backend
6      ...
7      environment:
8        - DB_NAME=${DB_NAME}
9        - DB_USER=${DB_USER}
10       - DB_HOST=db
11       - DB_PASSWORD=${DB_PASSWORD}
12       ...
13   db:
14     ...
15     environment:
16       - TZ="Europe/Zurich"
17       - POSTGRES_DB=${DB_NAME}
18       - POSTGRES_USER=${DB_USER}
19       - POSTGRES_PASSWORD=${DB_PASSWORD}
20     ...
21   ...
22 ...
```

5.4.6 Volumes

This volume specification binds the `./data` folder in the host machine to the `/application/data` folder in the container.

```
1  version: "3.8"
2
3  services:
4    backend:
5      build: .
6      container_name: backend
7      ...
8      volume:
9        - ./data:/application/data
10 ...
```