SOFTWARE ENGINEERING PRINCIPLES

BY PANKAJ KAMTHAN

1. INTRODUCTION

The more science becomes divided into specialized disciplines, the more important it becomes to find **unifying principles**.
— Hermann Haken

The role played by software in society is indispensable. As evident from the current literature [Forward, Lethbridge, 2008], software is increasingly being produced for a variety of domains. The question then is if there is any **'thread'** that binds the management, development, and improvement of these software systems. In pursuit of answering this question, this document explores a collection of software engineering principles (SEP)[1].

CONVENTION/NOTATION

In the rest of the document, the terms **'software'** and **'software system'** are considered synonymous, and therefore interchangeable.

2. THE NEED FOR TIME-INVARIANT, ESSENTIAL KNOWLEDGE

There is no shortage of knowledge today that is a **moving target**. The proposals that are perceived, and sometimes marketed, as **'latest and greatest'** today may face **obsolescence** tomorrow. For example, **(information) technologies, tools, and trends** are such.

[1] The acronym denotes the use in both singular and plural forms.

There is need for knowledge that is relatively **stable**. If the knowledge does not stand the **test of time**, the boundary of the subject and therefore the scope of learning are unclear. In general, the existence of such stable knowledge is a sign of **maturity** of a subject. For example, in mathematics, such knowledge appears in the form of definitions, theorems, and proofs.

The SEP are **time-invariant means** of guidance. It could be posited that SEP form **essential rather than ephemeral (or accidental) knowledge** of the discipline of software engineering [Brooks, 1987].

The scope of SEP is broad. For example, one or more of the SEP apply to project, process, people, and product **throughout** software development and beyond. They are relevant **both** to an individual software engineer and to a team of software engineers.

3. USES OF SOFTWARE ENGINEERING PRINCIPLES

An ounce of prevention really is worth a pound of cure.
— Brian W. Kernighan and Rob Pike

There are various applications of the SEP.

- **SEP: Basis for Concepts.** The SEP themselves form the **basis for other concepts** in software engineering. For example, other forms of knowledge such as design principles [Lethbridge, Laganière, 2005], techniques/methods/methodologies, best practices[2]/guidelines/heuristics, patterns/anti-patterns, or metrics, are each, in some way, based on the SEP. SEP are important for domain understanding [Millett, Tune, 2015]. In recent years, it has been established that **agile principles** are aligned with SEP [Séguin, Tremblay, Bagane, 2012].

- **SEP: Basis for Tools.** The SEP also form the basis of computer-aided software engineering (CASE) tools. For example, the idea of an **add-on** or **plug-in** to a user agent (such as Mozilla Firefox), an application software (such as Adobe Air), or to an integrated development environment (IDE) (such as IBM Eclipse) is based on the principles of modularity and incrementality.

[2] URL: http://simpleprogrammer.com/2013/02/17/principles-are-timeless-best-practices-are-fads/ .

- **SEP: Means of Prevention.** The SEP are **preventative means** for quality assurance and thereby aim to improve the quality of software systems. The SEP are important for **successful** software development; however, they are **not guarantees** for success. The SEP can help **avoid practices** that have led to **software project failures** [Boehm, 1983] and **software failures**.

4. HISTORY OF SOFTWARE ENGINEERING PRINCIPLES

It should be noted that a formulation of principles is **not arbitrary**. There can be several **factors** including the underlying philosophy, ethics, and experience in a subject that forms the basis of principles.

The history of SEP is long and interesting, and goes back to the origins of software engineering, namely in **programming** [Dijkstra, 1982].

5. META-PRINCIPLES

There are two main **meta-principles** that serve as a guide to any (engineering) principle [Meyer, 2014, Section 4.1]:

- **Abstractness:** A principle should be abstract enough so that it can be broadly applicable.

- **Falsifiability:** A principle should be a **contingency**. (A principle should **not be a platitude or a tautology**.) In other words, it should be possible to disagree with a principle and find counter-examples support such a disagreement.

6. THE PANORAMA OF SOFTWARE ENGINEERING PRINCIPLES

There are **different collections** of SEP [Boehm, 1983; Davis, 1994; Davis, 1995; Bourque, Dupuis, Abran, Moore, Tripp, Wolff, 2002; Ghezzi, Jazayeri, Mandrioli, 2003]. There are efforts to **unify** past SEP [Wang, 2008; Meridji, Abran, 2010].

The scope of SEP varies. For example, most SEP in [Boehm, 1983] apply to project, process, and people, rather than to product.

7. AN OVERVIEW OF SOFTWARE ENGINEERING PRINCIPLES

In this document, an exposition of one collection [Ghezzi, Jazayeri, Mandrioli, 2003] is considered in detail. The principles outlined in this collection are:

- **Principle 1. Separation of Concerns**
- **Principle 2. Modularity**
- **Principle 3. Incrementality**
- **Principle 4. Abstraction**
- **Principle 5. Generality**
- **Principle 6. Anticipation of Change**
- **Principle 7. Rigor**
- **Principle 8. Formality**

REMARKS

The collection of SEP included in this document is shown to be **relevant to software architecture** [Vogel, Arnold, Chughtai, Kehrer, 2011, Chapter 6].

7.1. RELATIONSHIP AMONG SOFTWARE ENGINEERING PRINCIPLES

The aforementioned SEP, **Principles 1-8**, are **not necessarily mutually exclusive**. Indeed, they can support each other, as evident from later discussion.

7.2. SOFTWARE ENGINEERING PRINCIPLES IN PERSPECTIVE

If we cannot illustrate a principle with a real product, there may well be something wrong with the principle.
— David L. Parnas

The principles are there to help us eliminate bad smells. They are not a perfume to be liberally scattered all over the system.
— Robert C. Martin

The best designers sometimes disregard the principles of design. When they do so, however, there is usually some compensating merit attained at the cost of the violation. Unless you are certain of doing as well, it is best to abide by the principles.
— William Lidwell, Kritina Holden, Jill Butler

SCOPE

- **SEP: Contextual.** The aforementioned SEP are **relative** (rather than absolute), **subjective** (rather than objective), and **situational** (rather than universal). For example, not all SEP are equally important in all software systems, in all situations. The choice of suitable SEP depends, among other factors, on the **goals of software quality**.

- **SEP: Guide, Not Mandate.** The principles are **not legal entities** (unless otherwise stated) in the sense that they are not ‘laws’ to be mandated and followed. However, they do guide the evolution of a subject, and therefore it is difficult to ignore them altogether.

In the following sections, perceived **limitations** of each SEP have been pointed out.

8. PRINCIPLE 1. SEPARATION OF CONCERNS

A wealth of information creates a poverty of attention.
— Herbert Simon

DEFINITIONS

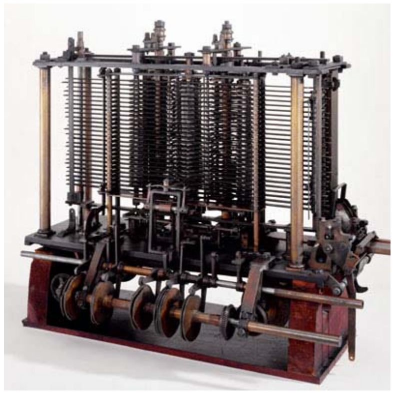
Definition [Concern]. A concern is a topic of interest or focus.

<u>Definition</u> [Concern] [ISO/IEC/IEEE, 2011]. [A kind of] interest in a system relevant to one or more of its stakeholders.

HISTORY

The historical context of separating concerns is in **politics and sociology**, specifically in imperialistic conquests, as suggested by the phrase **divide and conquer** (derived from the Latin **divide et impera**).

This still seems to be alive and well in the modern times of politics, sociology, and economics. The goals of the principle of separation of concerns in a software engineering context are rather modest.



Analytical Engine. (Source: Google Images.)

According to the **Computer History Museum**[3]:

> The Analytical Engine [...] had a 'Store' where numbers and intermediate results could be held, and a **separate** 'Mill' where the arithmetic processing was performed.

The principle of separation of concerns, as it applies to software engineering, has its **origins in computer programming** [Dijkstra, 1982][4].

The principle of separation of concerns suggests that **concerns be separated, and that each concern be looked at separately**. This allows one to deal with different aspects of a problem, and to concentrate on each individually.

MOTIVATION

There are several **reasons** for separating concerns. A premise fundamental to software engineering is that of **scale**. In general, non-trivial problems tend to be **large** and (**cognitively** or **structurally**) **complex**.

A property is **essential** if it cannot be eliminated (although it is possible to control and manage it). The **complexity** of software is an **essential property**, not an accidental property [Brooks, 1995]. There are different kinds of complexity, including **cognitive complexity**. The human intellect has constraints: it can handle only limited amounts of information and even this ability is time-variant.

There are various possible **incarnations** of the principle of separation of concerns: separation of concerns by space, separation of concerns by time, separation of concerns by views, separation of concerns by quality, and so on.

[3] URL: http://www.computerhistory.org/babbage/engines/ .

[4] The following excerpt from the publication suggests the origin: "Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation [...] all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. [...] But nothing is gained [...] by tackling these various aspects simultaneously. It is what I sometimes have called "the **separation of concerns**", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by "focusing one's attention upon some aspect": it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously."

REMARKS

- The principle of separation of concerns is critical to the pursuit of **prefactoring** (that is, development of software systems based on experiential knowledge, such as **guidelines**) [Pugh, 2005]. For example, "separate concepts into different classes based on behavior, not on data" is an object-oriented design guideline. Its relationship to its namesake refactoring is that, by applying the experiential knowledge, might decrease the amount of **refactoring** (that is, changing the internal structure of existing source code without changing its external behavior) required.

- There are a number of concerns in software development and, according to the **CONSIDER ALL FACTORS** pattern [Van Hilst, Fernandez, 2010], they **all** need to be considered.

CASE: SEPARATION OF CONCERNS BY SPACE

There are a number of examples of separation of concerns by space[5].

EXAMPLE 1

In **distributed computing**, the client is separated from the server. In **hypermedia engineering**, the source is separated from its target (destination). This allows the resource at each end to **evolve independently**.

EXAMPLE 2

The principle of separation of concerns is the basis for presenting **primary and secondary actors** in visually distinct places in a **UML Use Case Diagram**.

EXAMPLE 3

For the sake of **readability**, a UML Activity Diagram that is lengthy due to the presence of several sequential and parallel activities can, using connectors, be split into parts, as shown in Figure 1.

[5] This is more commonly known as **separation of concerns by parts**.

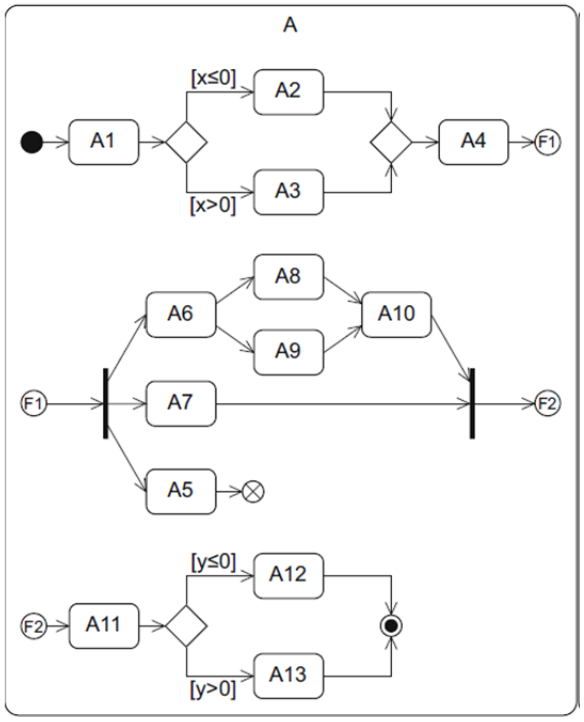


Figure 1. A UML Activity Diagram is separated into three, semantically interrelated, parts. (Source: [Seidl, Scholz, Huemer, Kappel, 2015].)

EXAMPLE 4

The principle of separation of concerns is the inspiration for the separation of (1) structure of information, (2) processing of information, and (3) presentation of information.

This is advocated by the **MODEL-VIEW-CONTROLLER (MVC)** pattern [Buschmann, Meunier, Rohnert, Sommerlad, Stal, 1996].

EXAMPLE 5

The principle of separation of concerns is the inspiration for the notion that **structure, content, and layout** of a document can be separated from each other, as shown in Figure 2, and each of these can be **authored and processed separately**.

This is advocated by the **Extensible Markup Language (XML)** [Salminen, Tompa, 2011, Section 4.1.1].

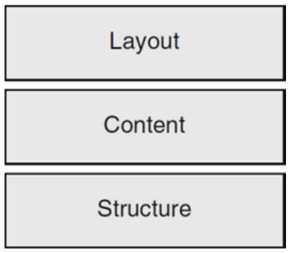


Figure 2. The three facets of a document.

EXAMPLE 6

The principle of separation of concerns is the inspiration for the **modular layout design** of a typical Home Page [Curtis, 2010].

EXAMPLE 7

The principle of separation of concerns is the basis for **path branching** and **path merging** in **(software) configuration management** [Aiello, Sachs, 2011], as shown in Figure 3.

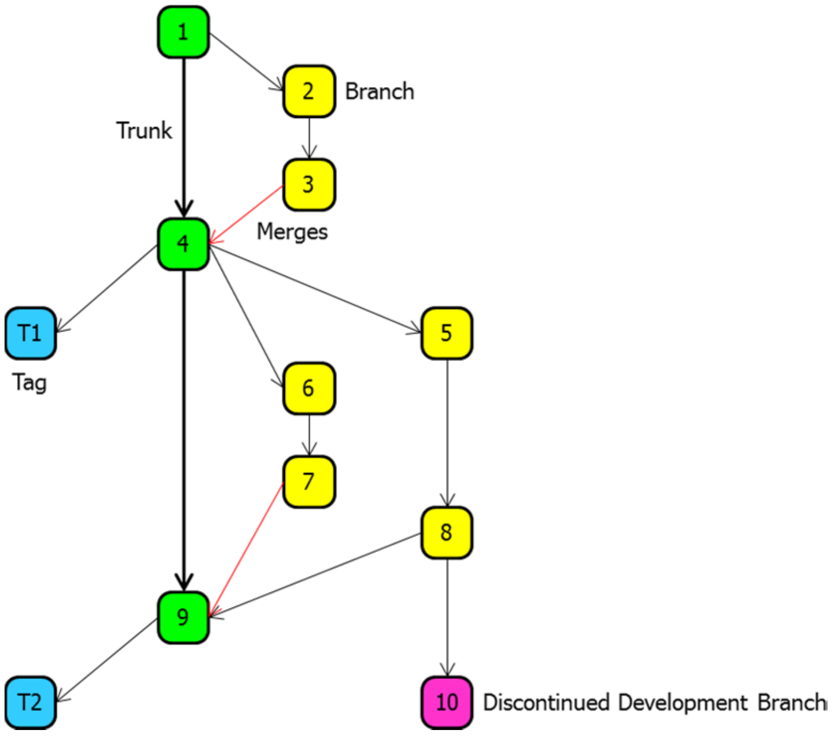


Figure 3. The principle of separation of concerns is applied to “branching and merging” of a typical version control system.

EXAMPLE 8

The principle of separation of concerns is the basis for a variety of programming paradigms including **object-oriented programming (OOP)** and **aspect-oriented programming (AOP)**. OOP advocates the separation of concerns into **objects**. AOP goes a step further: it is acknowledges that certain concerns are inherently 'cross-cutting' and advocates the separation of concerns into **aspects** and **objects**.

Figure 4 illustrates a case of cross-cutting concerns and their amelioration.

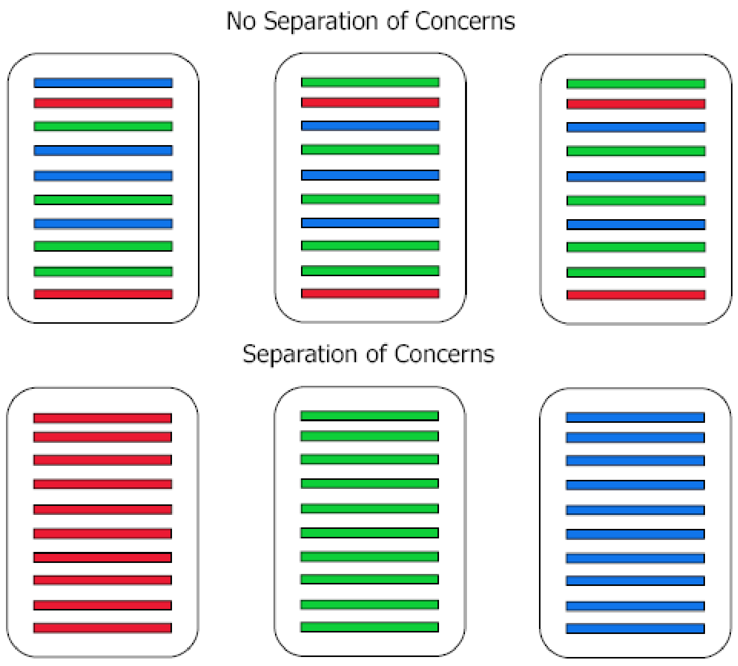


Figure 4. The principle of separation of concerns is applied to fragments of 'scattered' and 'tangled' source code.

CASE: SEPARATION OF CONCERNS BY TIME

There are a number of examples of separation of concerns by time[6].

[6] For the sake of argument, a trivial example of the principle of separation of concerns by time is to study during the week and pursue personal interests on the weekend (unless, of course, one is a student in computer science and software engineering in which case this is a non-example, as a temporal separation is not possible or recommended).

EXAMPLE 1

The principle of separation of concerns by time suggests that the lifecycle of software could be separated into, say, **pre-production, production, and post-production**.

A process model is concerned with production, which can be subdivided further. For example, a process model can be separated into phases (or stages).

EXAMPLE 2

In some cases, the **separation of concerns by time is unavoidable**. For example, there must be a requirement before the design and, in turn, a design before implementation. (If that is not the case, then requirement and/or design knowledge is implicit.)

EXAMPLE 3

It is known that, usually, **software requirements are not all equally significant**. Thus, requirements are often **prioritized**. Then, highest priority requirements (including those that are essentially about 'bells and whistles') should be addressed (designed, implemented, and tested) before the others. The same idea applies to use cases.

CASE: SEPARATION OF CONCERNS BY VIEWS

A view is about **perception** of a certain reality.

EXAMPLE 1

A thing can have **multiple views** of interest, as shown in Figure 5.

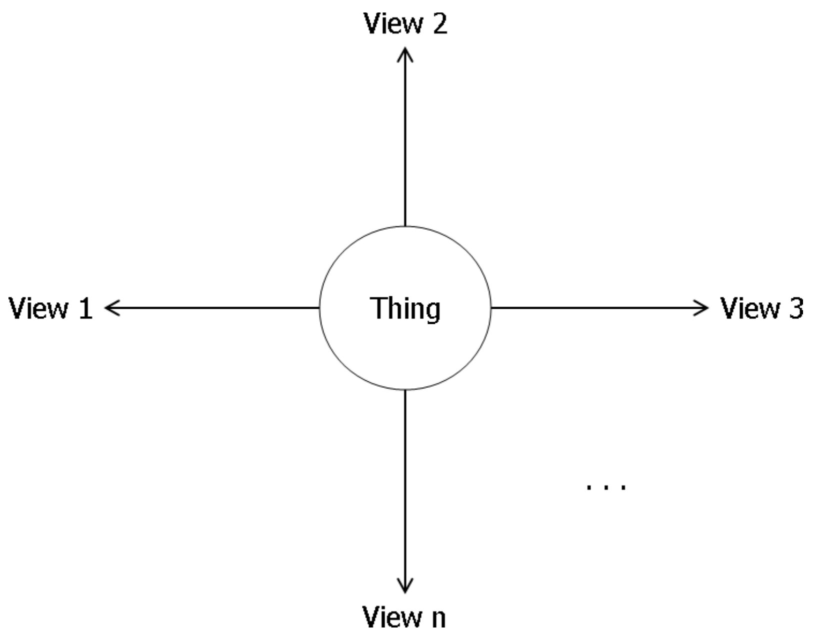


Figure 5. A thing with n significant views.

EXAMPLE 2

There can be different kinds of things, with different views of interest. For example, there are **four significant 'views'** of software engineering, given by the **IEEE Software and Systems Engineering Standards Committee**, as shown in Figure 6.

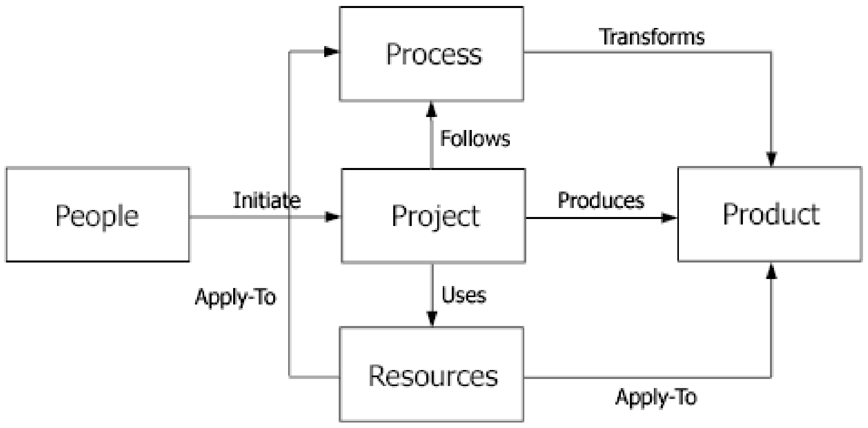


Figure 6. The basic elements of software engineering. (Source: IEEE.)

EXAMPLE 3

In the **HOT Framework** for understanding the **agile software engineering**, there are three significant views [Hazzan, Dubinsky, 2014]:

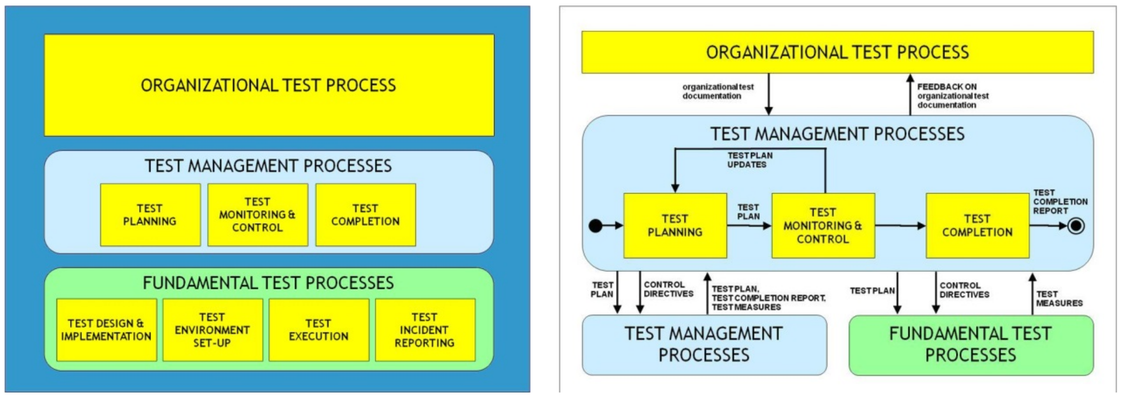
1. **The Human Perspective.** This includes cognitive and social aspects and refers to learning and interpersonal (teammates, customers, management) processes.

2. **The Organizational Perspective.** This includes managerial and cultural aspects and refers to the workspace and issues that spread beyond the team.

3. **The Technological Perspective.** This includes practical and technical aspects and refers to how-to and source code-related issues.

EXAMPLE 4

The view(s) taken towards software engineering can change the way software is developed. For example, a **test process** can have **static and dynamic** views, as illustrated, respectively, in Figure 7.


ORGANIZATIONAL TEST PROCESS
TEST MANAGEMENT PROCESSES
TEST PLANNING
TEST MONITORING & CONTROL
TEST COMPLETION
FUNDAMENTAL TEST PROCESSES
TEST DESIGN & IMPLEMENTATION
TEST ENVIRONMENT SET-UP
TEST EXECUTION
TEST INCIDENT REPORTING
ORGANIZATIONAL TEST PROCESS
organizational test documentation
FEEDBACK ON organizational test documentation
TEST MANAGEMENT PROCESSES
TEST PLAN UPDATES
TEST PLANNING
TEST PLAN
TEST MONITORING & CONTROL
TEST COMPLETION
TEST COMPLETION REPORT
TEST PLAN
CONTROL DIRECTIVES
TEST PLAN, TEST COMPLETION REPORT, TEST MEASURES
TEST PLAN
CONTROL DIRECTIVES
TEST MEASURES
TEST MANAGEMENT PROCESSES
FUNDAMENTAL TEST PROCESSES

(a) Static View (b) Dynamic View

Figure 7. The two different views of the ISO/IEC 29119 Standard Test Process Model[7].

[7] URL: http://softwaretestingstandard.org/ .

EXAMPLE 5

A (computer) **program** can have multiple views, each by possibly a different stakeholder of that program, including those that show **decision (control) flow** and **data flow** through the program.

EXAMPLE 6

A **document** can have multiple views, each serving a different purpose, and made explicit by a **reader**, as illustrated in Figure 8.

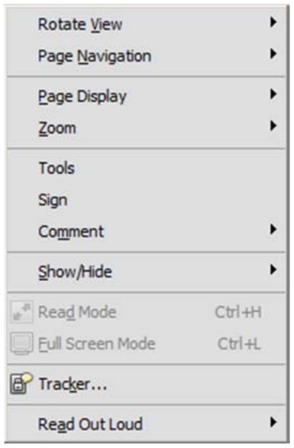


Figure 8. The Adobe Reader provides multiple views of the same document.

EXAMPLE 7

Figure 9 shows a view of a process.

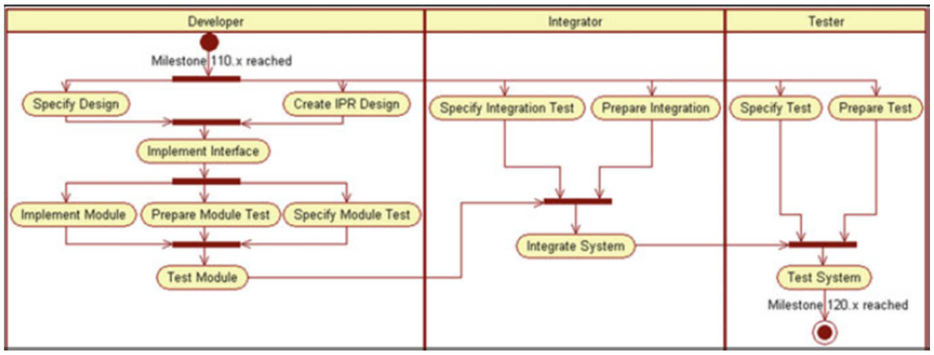


Figure 9. A **process-oriented view** of subprocess **"Develop Subsystem (SW)"** as UML Activity Diagram. (Source: [Dietrich, Killisperger, Stückl, Weber, Hartmann, Kern, 2013].)

EXAMPLE 8

Figure 10 shows a view of a product.

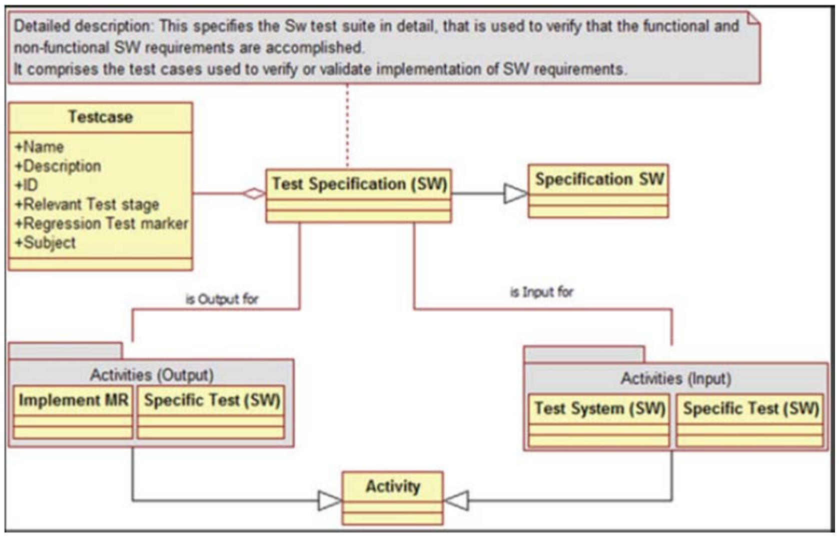

Detailed description: This specifies the Sw test suite in detail, that is used to verify that the functional and non-functional SW requirements are accomplished.
It comprises the test cases used to verify or validate implementation of SW requirements.

Figure 10. A **product-oriented view** of work product **"Test Specification (SW)"** as UML Class Diagram. (Source: [Dietrich, Killisperger, Stückl, Weber, Hartmann, Kern, 2013].)

EXAMPLE 9

Figure 11 shows a view of a person.

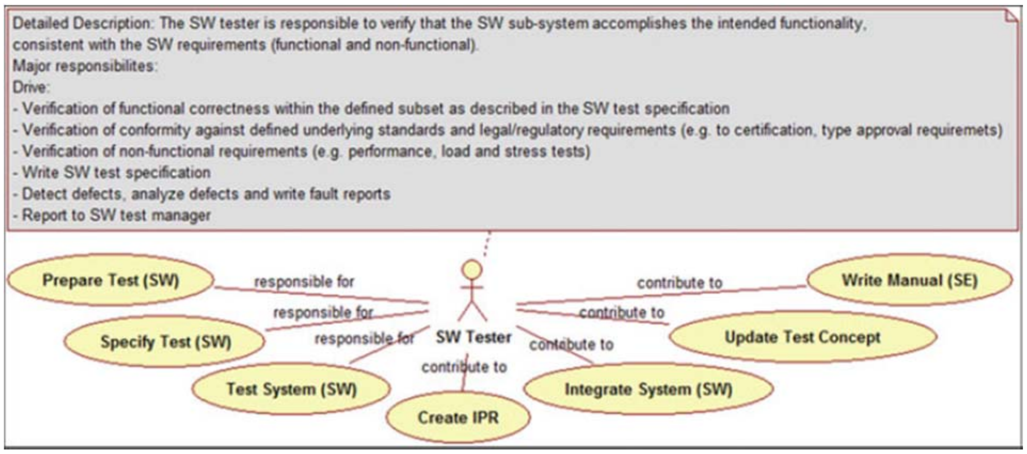

Detailed Description: The SW tester is responsible to verify that the SW sub-system accomplishes the intended functionality, consistent with the SW requirements (functional and non-functional).
Major responsibilites:
Drive:
- Verification of functional correctness within the defined subset as described in the SW test specification
- Verification of conformity against defined underlying standards and legal/regulatory requirements (e.g. to certification, type approval requiremets)
- Verification of non-functional requirements (e.g. performance, load and stress tests)
- Write SW test specification
- Detect defects, analyze defects and write fault reports
- Report to SW test manager

Figure 11. A **role-oriented view** of role **"SW Tester"** as UML Use Case Diagram. (Source: [Dietrich, Killisperger, Stückl, Weber, Hartmann, Kern, 2013].)

CASE: SEPARATION OF CONCERNS BY QUALITY

There are a number of quality attributes. However, for a given software system, these quality attributes are **not all equally significant**. This motivates the need for **prioritization**, which in turn depends on separation of concerns by quality attributes.

EXAMPLE

In software quality engineering, one recommended approach for dealing with software quality is the following: **syntactic correctness first, semantic correctness next, and efficiency thereafter**.

CASE: SEPARATION OF CONCERNS BY RESPONSIBILITIES

This is about one aspect of **object-oriented design (OOD)**, namely **class design**. To **effectively manage change during development or maintenance**, it is important to **allocate responsibilities** properly to each class.

EXAMPLE

This is regarding class design shown in Figure 12 and described in [Seidl, Scholz, Huemer, Kappel, 2015]. In this case, **`Course`**, **`Employee`**, **`Student`**, and **`Printer`** are different concerns. Furthermore, **design and implementation** are different concerns.

(An **interface** signifies a **contract**. An **interface** does not have any direct instances and its operations do not have an implementation. An interface is denoted like a class but with the additional keyword **`<<interface>>`** before the name. A dashed inheritance arrow with a hollow, triangular arrowhead from a class to an interface signifies that this class implements the interface. A dashed arrow with an open head with the keyword **`<<use>>`** signifies that a class uses an interface.)

The classes **`Course`** and **`Person`** implement the interface **`Printable`**. The classes that implement **`Printable`** must provide an operation **`print()`**. The **`print()`** operation is different for each class. For example, for a course, the **name and the number of hours** are printed, and for a **`Person`**, the **name and address** are printed. For **`Employee`**, the behavior of the operation **`print()`** in **`Person`** is deemed sufficient. However, in the class **`Student`**, the behavior of the operation **`print()`** is **inherited** from **`Person`** and **extended**. (In fact, the operation **`print()`** is overwritten as the matriculation number is also printed.)

The class **`Printer`** can now process each class that implements the interface **`Printable`**. Thus, a specific operation **`print()`** can be realized for **each class** and the class **`Printer`** remains **unchanged**.

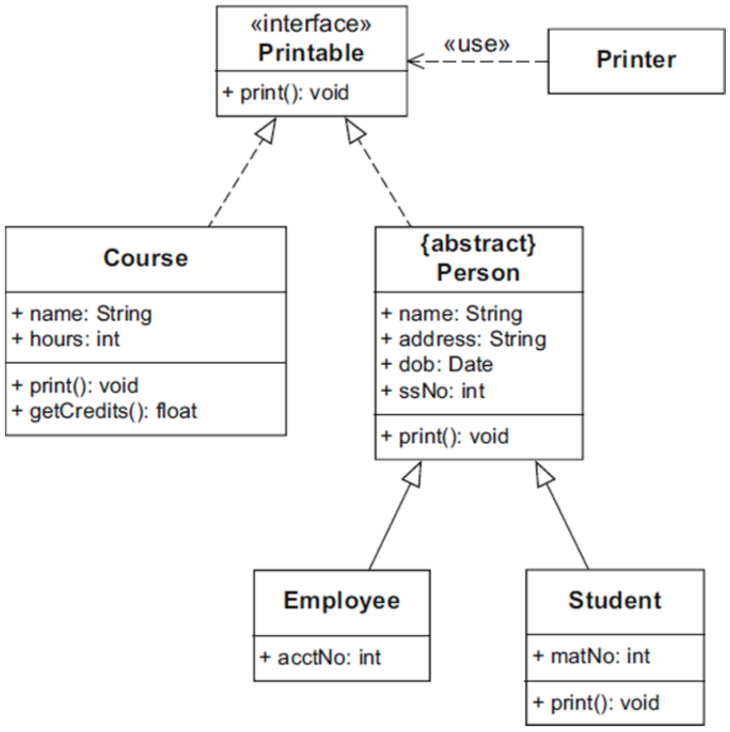


Figure 12. A UML Class Diagram at different levels of abstraction. (Source: [Seidl, Scholz, Huemer, Kappel, 2015].)

ON THE RELATIONSHIP AMONG DIFFERENT CASES OF SEPARATION OF CONCERNS

There are both points of convergence and divergence among the different cases of principles of separation of concerns.

The principles of **separation of concerns by space and by time** are, obviously, **different**. For example, **parallelization is separation of concerns by space**, and **concurrency is separation of concerns by time**.

However, the different cases of principles of separation of concerns can **co-exist**.

For example, they can be applied to the team projects in the following manner: the identification and separation of responsibilities (including those related to quality assurance/evaluation) takes place based on different views of the project, which is followed by distribution/allocation of work (space) and scheduling (time).

In the **Unified Process (UP)** [Jacobson, Booch, Rumbaugh, 1999], the **workflows** (space) and the **phases** (time) are interrelated.

SEPARATION OF CONCERNS BY VIEWS AND BY QUALITY

EXAMPLE

The following source code of a Java program can have multiple views, each relevant to some (possibly different) stakeholder.

```
public class DoNotAttempt {
  public static void main(String() args) {
    int x = 1;
    while (x > 0) {
      System.out.println("Finite Loop!");
    }
  }
}
```

Here are some of the relevant semiotic views [Baecker, Marcus, 1990, Section 3.2]:

1. **Lexical Quality View.** This is relevant to programming language designer. This is because a programming language designer needs to be concerned about the characters (from some character set), or perhaps **tokens**, adopted by the programming language.

2. **Syntactic Quality View.** This is relevant to programmer. This is because if the source code of a program is not syntactically correct, then it is not even a program. In the previous example, a syntactical error is shown in orange.

3. **Semantic Quality View.** This is relevant to programmer and reviewer. This is because if the source code of a program is not semantically correct, then it violates the knowledge of the domain it addresses. In the previous example, a semantical error is shown in dark red.

4. **Pragmatic Quality View.** This is relevant to reviewer. This is because the source codes of programs are read (and understood) by people **other** than their programmers. It is **not automatic** that the source code of some program is readable (or understandable).

From a software engineering perspective, these views are equally important, unless otherwise highlighted.

RELATED SOFTWARE ENGINEERING PRINCIPLE(S)

The **principle of abstraction** and the **principle of modularity** enable a realization of the principle of separation of concerns by space.

LIMITATIONS

There are certain indirections to separation of concerns.

The primary practical challenge to separation of concerns in a product is to be able to find appropriate 'boundaries' in order to separate the concerns. For example, the quality attributes, such as security and usability, are often **not independent** of each other. It is yet to be satisfactorily seen if **aspect-oriented software development (AOSD)** is a viable approach to this issue.

A separation of X into two concerns X_1 and X_2 means that X_1 and X_2 must **both** be **managed**. The responsibility of **management increases** as the number of (separate) **concerns increases**. For example, separation of concerns raises the **issue of traceability among the concerns involved**.

9. PRINCIPLE 2. MODULARITY

The principle of modularity is based on the following:

1. **Decomposition (Top-Down Division).** A large and/or complex system is divided into simpler parts called **module**s, where each module performs a set of (semantically-related) functionality.

2. **Composition (Bottom-Up Aggregation).** The modules interface and work together to perform a task.

MOTIVATION

The principle of modularity allows one to **manage size and complexity**. It contributes to **maintainability** in a number of ways:

- **Modifiability.** It aids **modifiability** as modules can be **modified independently** of each other. However, this is contingent on the degree of coupling between modules.

- **Reusability.** It aids **reusability** as higher-level modules can **reuse** lower-level modules. Therefore, a change in a single module requires changes **only** in dependent modules.

- **Understandability.** It aids **understandability** as it allows the restriction of focus on parts (modules) rather than the whole.

THE MEANING OF 'MODULE'

The meaning of the term **'module'** varies considerably in literature, and across paradigms and programming languages. For example, in **Java**, a **class** (or a **package**) is module.

EXAMPLE 1

The underlying philosophies of **object-orientation** (and, therefore, object-oriented programming) and **modularization** (and, therefore, modular programming) are based on modularity.

EXAMPLE 2

The ability to organize information is crucial to **Information Design (ID)** [Garrett, 2011]. The information could be organized as **top-down division** (categories imply segregation of content) or as **bottom-up aggregation** (content inspires the creation of certain categories), as shown in Figure 13.

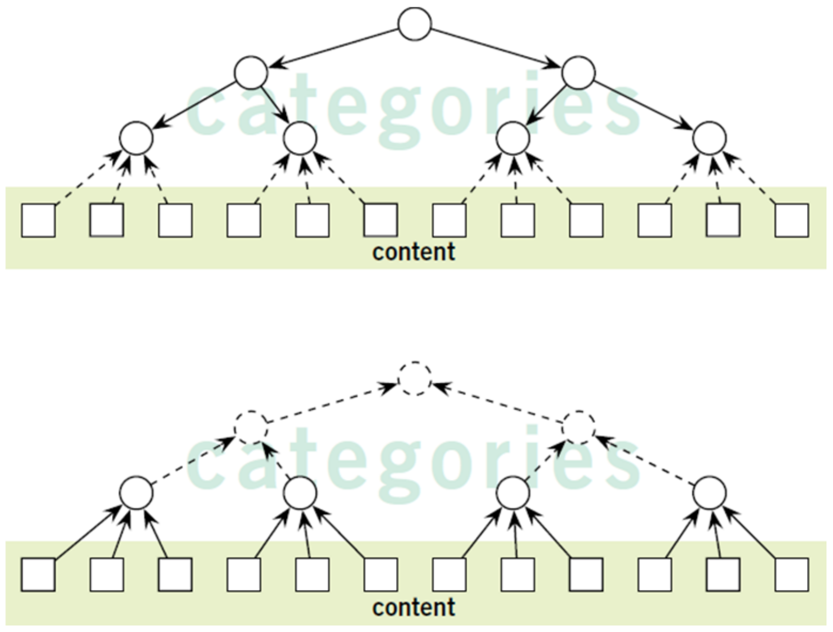


Figure 13. Two ways of organizing information. (Source: [Garrett, 2011].)

COHESION AND COUPLING

There are two basic properties of modules: **cohesion** and **coupling**[8].

Definition [Cohesion] [ISO/IEC/IEEE, 2010].
(1) The manner and degree to which the tasks performed by a single module are related to one another.
(2) In software design, a measure of the strength of association of the elements within a module.

[8] The concepts of **cohesion and coupling** were introduced in [Stevens, Myers, Constantine, 1974].

REMARKS

There are **different reasons for a module to be cohesive** (intradependent) leading to **different types of cohesion**: functional cohesion, communicational cohesion, temporal cohesion, and so on.

Definition [Coupling] [ISO/IEC/IEEE, 2010].
(1) The manner and degree of interdependence between modules.
(2) The strength of the relationships between modules
(3) In software design, a measure of the interdependence among modules in a computer program.

REMARKS

- There are **different reasons for modules to be coupled** (interdependent) leading to **different types of coupling**: data coupling, control coupling, content coupling, and so on.

- In **object-oriented design (OOD)**, classes can be coupled in different ways, including **inheritance**.

HIGH COHESION AND LOW COUPLING

Let the degree of dependency between two things be quantitatively measured on a spectrum, and let high and low be two **qualitative conclusions**.

The goal for a set of modules is **high cohesion and low coupling**[9]. In case of a module with **high cohesion**, all elements of the module are strongly related. This is possible when there is a strong rationale for **closeness or togetherness** among the elements. In case of a set of modules with **low coupling**, the communication between them is simple.

EXAMPLE

Figure 14 illustrates two configurations, one with **low cohesion and high coupling**, and the other with **high cohesion and low coupling**. The presence of **low cohesion or high coupling** means that the modules were not defined appropriately (and need to be **refactored**, if possible).

[9] The notion of **high cohesion and low coupling** is so important that it is sometimes labeled as one of the **design principles** [Vogel, Arnold, Chughtai, Kehrer, 2011, Chapter 6].

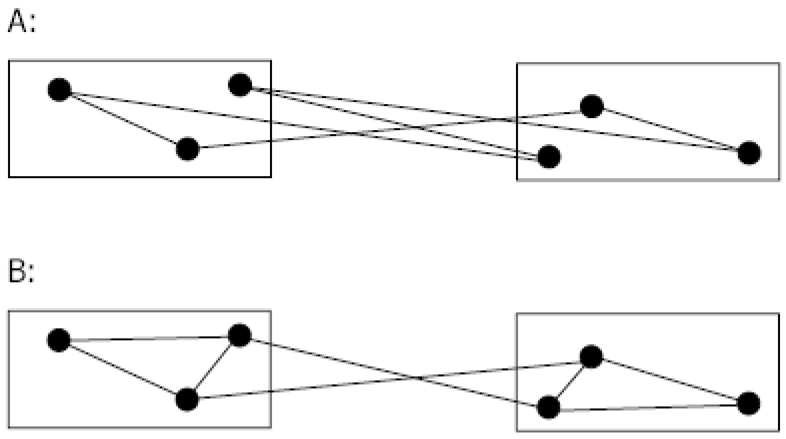


Figure 14. (a) A configuration with low cohesion and high coupling and (b) a configuration with high cohesion and low coupling.

COHESION AND OBJECT-ORIENTED DESIGN

Let the design of a class in an object-oriented system be **abstracted and modeled** as a **graph**. (For example, let a 'vertex' abstract and model an operation, and let an 'edge' abstract and model an operation call.) Then, the property of cohesion in a class belonging to such a system can be viewed from the perspective of **connectivity in that graph**.

The following is one metric for **connectivity in a graph, G**:

$$C(G) = \frac{2[e - (v - 1)]}{[(v - 1)(v - 2)]},$$

where v is the number of vertices and e is the number of edges in G [Streekmann, 2012].

It can be seen (omitting pathological cases) that the range of C(G) is the interval **[0, 1]**.

Then, **low C(G)** suggests **low cohesion** in a class, and **high C(G)** suggests **high cohesion** in a class.

EXAMPLE

For the graph (of the class) in Figure 15(a), C(G) = 0, and for the graph (of the class) in Figure 15(b), C(G) = 1.

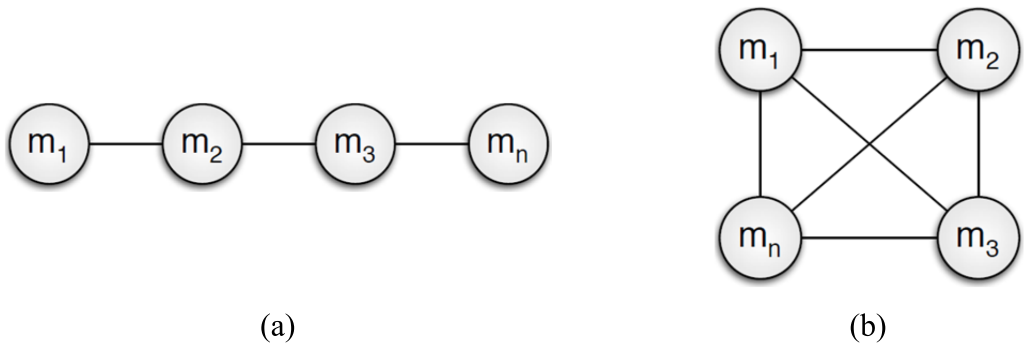


Figure 15. An illustration of C(G). (Source: [Streekmann, 2012].)

REMARK

C(G) is a **general metric**, and can be applied to any **software project artifact** that can be modeled as a **graph**.

COUPLING AND OBJECT-ORIENTED DESIGN

EXAMPLE

Let a module be a class, and a relationship (between modules) be an association.

In the construction shown in Figure 16(a), there are 5 classes, and 4 associations. In general, n classes lead to **at least** n – 1 associations.

In the construction shown in Figure 16(b), there are 5 classes, and 10 associations. In general, n classes lead to **at most** C(n, 2) = n(n – 1)/2 associations. (This can be proven using **mathematical induction**.)

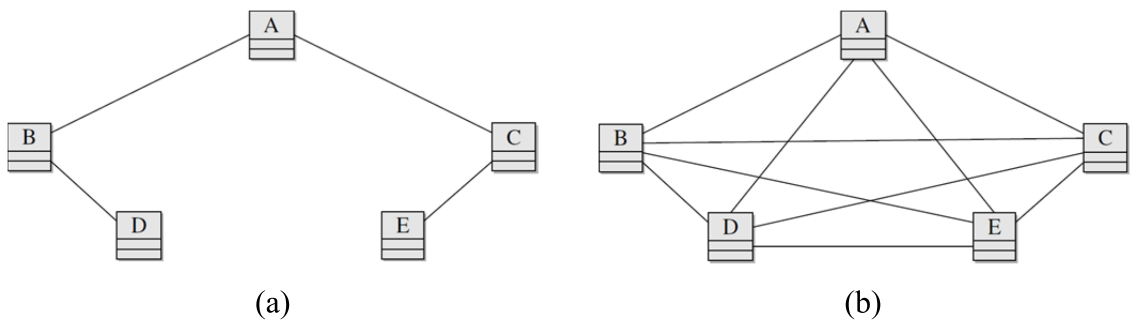


Figure 16. An illustration of (a) lower limit of coupling, and (b) upper limit of coupling. (Source: P. Grogono.)

COUPLING AND SERVICE-ORIENTED COMPUTING

The notion of 'low coupling' is important in **Service-Oriented Computing (SOC)** [Fehling, Leymann, Retter, Schupeck, Arbitter, 2014, Section 4.2.1].

COUPLING AND SOFTWARE PROCESSES

The notion of coupling applies to **software processes**.

EXAMPLE

The **Waterfall Model** [Royce, 1970] has **low coupling** among its phases, and **Extreme Programming (XP)** [Beck, Andres, 2005] has **high coupling** among its practices.

COUPLING AND SOCIAL NETWORKS

The notion of coupling can be applied to **relationships among people** as part of analyzing a **social network** [Hayes, 2003; Golbeck, 2013], as shown in Figure 17. There is also interest in investigating the **'strength'** of the 'coupling'.

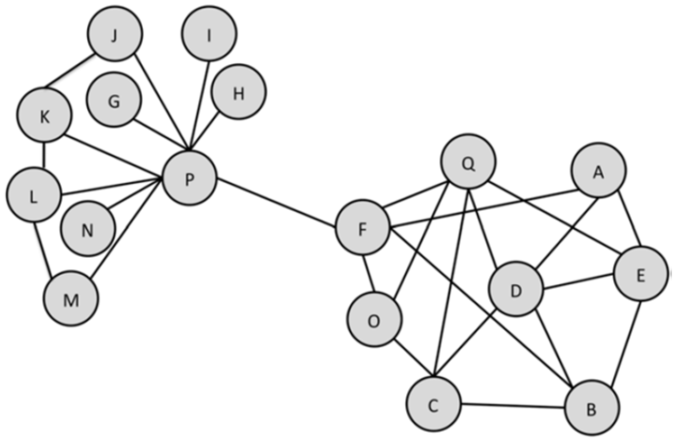


Figure 17. A **graph-theoretic model** of a social network. (Source: [Golbeck, 2013].)

RELATED SOFTWARE ENGINEERING PRINCIPLE(S)

The principle of modularity is an **extension** of the principle of separation of concerns.

LIMITATIONS

It is easier to discuss modularity in theory than to carry out in practice.

10. PRINCIPLE 3. INCREMENTALITY

The principle of incrementality characterizes an evolutionary process that proceeds in a stepwise fashion, in **approximations** of the end-product. The idea is that each **approximation** is an **increment** over the previous one.

The difference between two subsequent increments is a **step**. The challenge in practice, of course, is in determining how 'small' any step, including the first step, should be[10]. It is likely, but not necessary, that the steps are identical.

MOTIVATION

The principle of incrementality relies on the **history of computer science**. The term **"successive refinement"** was used in the early 1970s, and referred to a methodology that encouraged gradual elaboration of programs to deal with size and complexity.

The principle of incrementality is motivated by **practical experiences** in software engineering, including the fact that there are domains for which **not all requirements are known or understood**. In such a case, according to the principle of incrementality, **subsets** that provide **value** [Biffl, Aurum, Boehm, Erdogmus, Grünbacher, 2006] could be **developed first for an early feedback**. In short, the principle of incrementality is an **approach towards learning** (more about requirements, and indeed more about the product itself) through **communication among stakeholders** [Hazzan, Dubinsky, 2014, Chapter 10].

The principle of incrementality aims to **ease software development**. It also leads to a **small turnaround for tracing and debugging**. The idea is the following: if the modified system does not work as expected, the errors must have been introduced by the small step (provided that there are no hidden dependencies).

A MATHEMATICAL INTERPRETATION OF INCREMENTALITY

Let each increment be considered as a set, and let P be some **target** end-product. For example, P could be a conceptual model for some software system or the user interface design of some software system. Then the principle of incrementality could be expressed in a more compact manner to suggest the following.

[10] This is reminiscent of a similar issue that is faced when using certain **numerical methods** for approximating the value of a **definite integral**.

There exists a finite sequence $<P_n>$ of non-empty sets, n denotes time, such that

(1) $| P_n |$, $| \cdot |$ denotes size, is strictly monotonically increasing (that is, $P_i \subset P_{i+1}$, for $i \geq 1$),

and

(2) $<P_n> \longrightarrow P$, as n increases.

Let $s_i = P_{i+1} - P_i$ denote the steps. Then, $| s_i |$ can either be a **constant** or a **variable**.

INCREMENTALITY AND 'PIECEMEAL GROWTH'

The notion of **'piecemeal growth'**, introduced in the context of **patterns for urban architecture** [Alexander, Ishikawa, Silverstein, 1977], is based on the principle of incrementality.

INCREMENTALITY AND AGILE METHODOLOGIES

Business Days[11] **and Short Releases** are a hallmark of agile methodologies [Hazzan, Dubinsky, 2014, Chapter 10], and a manifestation of the principle of incrementality.

INCREMENTALITY AND PROTOTYPING

The notion of **(rapid) prototyping**[12] is based on the principle of incrementality. The **intermediate stages** of an incrementally developed product are called **prototypes**[13].

There are several **applications of prototyping**. For example, when building a library of numerical functions, it is preferable to add and test elementary functions to each of the categories (for example, matrix manipulation, numerical integration, and so on) first, rather than attempt to completely exhaust one category.

[11] A **Business Day** takes place between each pair of consecutive iterations of a release [Hazzan, Dubinsky, 2014]. The rest of the iteration days are working days. In the first part of the Business Day, the previous iteration is summarized. In the second part, after a reflective session takes place, the next iteration planning starts.

[12] It has been stated [Wang, 2008] that prototyping itself is a SEP; however, the rationale for doing so is unclear.

[13] There are distant similarities between prototype and conceptual model, and between prototype and proof-of-concept but, essentially, these are pairwise different.

The construction of a **user interface** is a prime example that can benefit from prototyping. For example, it is **preferable** to introduce a few constructs (for example, button, menu, window, and so on) in user interface design, and follow it by verification (say, user evaluation and feedback).

INCREMENTAL VERSUS ITERATIVE

There are **differences** between an incremental approach towards development and an iterative approach towards development.

For example, an **incremental** approach aims to **improve the process**; an **iterative** approach aims to **improve the product** [Cockburn, 2008].

LIMITATIONS

An incremental approach towards development may give an impression that the project is in a constant state of progress **(‘perpetual beta’**), and therefore that of **immaturity**. This perception may discourage some project managers to support it broadly.

There are certain **limitations to prototyping**:

- A software process model may or may not support prototyping. For example, the **Spiral Model** supports prototyping, while the **Waterfall Model**, essentially, does not.

- After a certain number of prototypes, the **cost** of creating prototypes may **outweigh** the **benefits**. This is particularly the case with a high frequency of **throwaway prototypes**.

11. PRINCIPLE 4. ABSTRACTION

What makes the desert beautiful is that somewhere it hides a well.
— Antoine de Saint Exupéry

Thinking like a computer scientist means more than being able to program a computer. It requires thinking at multiple levels of abstraction.
— Jeannette M. Wing

The principle of abstraction is an **extension** of the principle of separation of concerns.

SHOW OR HIDE

The principle of abstraction suggests that, in a given context, one should highlight the **important/relevant aspects** of the problem, and **suppress** the **unimportant/irrelevant aspects** of the problem.

ABSTRACTION AND ASTRONOMY

The earth can be seen at different levels of abstraction, as shown in Figures 18 and 19.

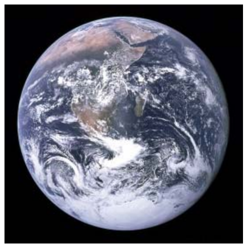


Figure 18. The **"Blue Marble"** photograph of Earth, taken during the Apollo 17 lunar mission in 1972. (Source: [Wikipedia].)

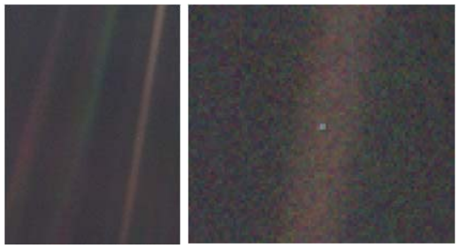
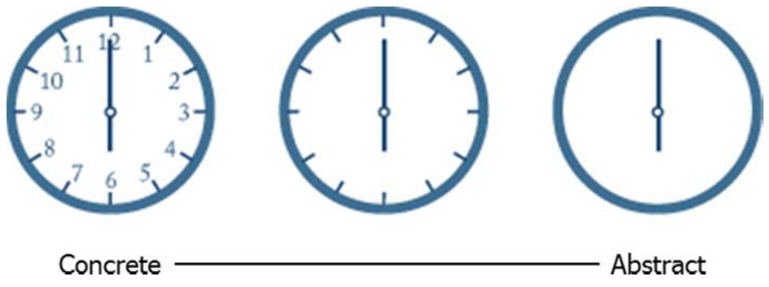


Figure 19. The **"Pale Blue Dot"** photograph of Earth, as seen from about 6 billion kilometers, in which the Earth appears as a tiny dot (the blueish-white speck approximately halfway down the brown band to the right). (Source: [Wikipedia].)

ABSTRACTION AND INDUSTRIAL ENGINEERING

A clock can be presented at different levels of abstraction.



(Source: Anonymous.)

ABSTRACTION AND TOY INDUSTRY

A toy car usually represents the external body of a car, but not the internal details of its engine. In other words, a **toy car abstracts (or suppresses)** the details.

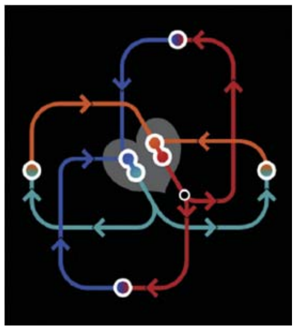
A Toy Car. (Source: Google Images.)

ABSTRACTION AND FINE ART

The notion of abstraction in fine art traces back to the **post-impressionism** era and early 20th century. In other words, the movement in fine art has been from concrete to abstract.

Abstract Art. (Source: “Counter-Composition” by Theo van Doesburg.)

ABSTRACTION AND HUMAN PHYSIOLOGY



A Model of Heart. (Source: "Functionality of Heart" [Katz, 2012].)

ABSTRACTION AND MARKETING

The Apple's "silhouette" advertisement campaign for the iPod began in 2003. The television advertisements featured dancing silhouettes holding white iPods and wearing white earphones. The advertisements **de-emphasized** the **design of the device** and **emphasized** the **user experience**.

Apple's iPod Advertisement. (Source: Google Images.)

ABSTRACTION AND THE USER INTERFACE DESIGN OF WEB APPLICATIONS

The principle of abstraction is the basis for the **modular layout design** of a typical Home Page [Curtis, 2010], as shown in Figure 20.

The **rectangles** corresponding to the components in the final layout after all the other interface details are removed. The header components (top of page) and footer components (bottom of page) are displayed as gray boxes, whereas remaining components in the body of the page are displayed as white boxes.

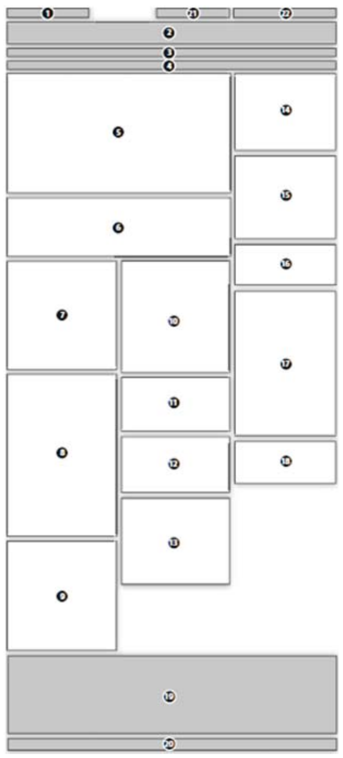
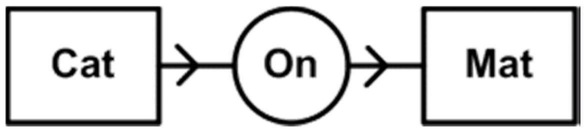


Figure 20. The layout of a typical Home Page. (Source: [Curtis, 2010].)

ABSTRACTION AND MATHEMATICS

The examples of abstractions in mathematics are ubiquitous. This is because **mathematics** is considered the **language of abstraction**.



A Conceptual Graph. (Source: Google Images.)

Indeed, **mathematical expressions** are **abstractions of reality**. For example, **numbers** (distance between earth and the moon), **sets** (collection of apples), **functions** (change in behavior), **relations** (car and doors), **graphs** (population growth versus time), and so on, are but a few examples of mathematical abstractions along with their real-world counterparts.

EXAMPLE 1

For the sake of argument, there is **no physical existence** of the **number “6”**. However, there are several representations of the number, as shown in Figure 21.

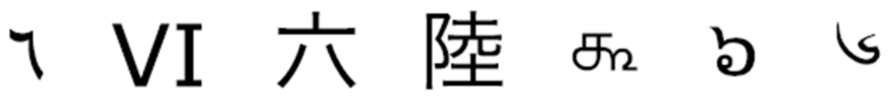


Figure 21. A collection of representations of the number “6”. (Source: [Machiavelo, 2012].)

EXAMPLE 2

The quadratic equation

$$ax^2 + bx + c = 0,\ a \neq 0,$$

is an abstraction of the equations

$$x^2 + 1 = 0,$$

$$x^2 - 2 = 0,$$

and so on.

EXAMPLE 3

The control flow structure of source code can be represented as a graph, as shown in Figure 22.

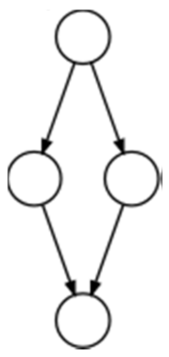


Figure 22. The `IF-THEN-ELSE` statement represented as a graph. (Source: Wikipedia.)

EXAMPLE 4

An Artificial Neural Network (ANN) is an interconnected group of vertices, analogous to the network of neurons in a brain, as shown in Figure 23. The mathematical modeling of neurons is an area of study in **Artificial Intelligence (AI)** and **Cognitive Science** [Arbib, 1987].

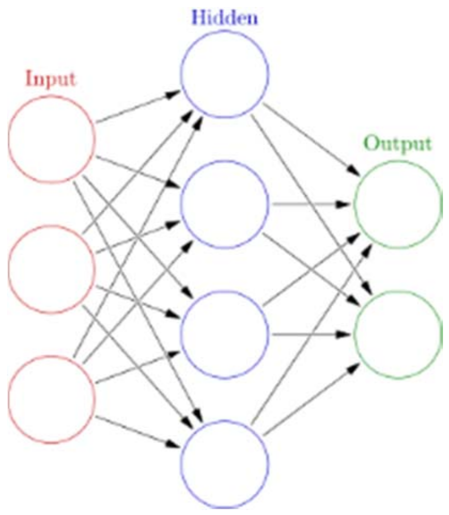


Figure 23. An artificial neural network modeled as a graph in which each circular vertex represents an artificial neuron and an edge represents a connection from the output of one neuron to the input of another. (Source: Wikipedia.)

ABSTRACTION AND PUBLISHING

The table of contents of a document (such as a book) can be presented at different levels of abstraction, as shown in Figure 24.

```
Preface
Chapter 1 Introduction
Chapter 2 Background on Software Quality
. . .
```

(a)

```
Preface
Chapter 1 Introduction
        1.1. Past
        1.2. Present
        . . .
Chapter 2 Background on Software Quality
. . .
```

(b)

Figure 24. The table of contents with a list of (a) chapters only, and (b) chapters and sections.

ABSTRACTION AND COMPUTER SCIENCE

The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.
— Edsger W. Dijkstra

There are numerous examples of abstractions in computer science [Kramer, 2007]. In general, **programs are abstractions** built on top of hardware.

EXAMPLE 1

An **Abstract Data Type (ADT)** is a mathematical model of a data type. An ADT, as opposed to a **data structure**, is independent of implementation. In (object-oriented) programming languages, the notion ADT is important.

EXAMPLE 2

In systems programming, **memory addresses abstract to variable names**, and the **effect of a procedure abstracts to the comments** in programs.

HIERARCHICAL ABSTRACTION AND “THE MAP THAT CHANGED THE WORLD”

The idea behind **hierarchical abstraction** is that one can break down the design of a computer into layers so that one can focus on one level at a time without having to worry about what is happening at the lower levels of the hierarchy.

There is an analogy with **geology** [Hey, Pápay, 2015]. Figure 25 shows the **geological map** of Britain devised by **William Smith**, the founder of **stratigraphy**, the branch of geology that studies rock layers and layering. In **1815**, William Smith published the first **large-scale geological map of Britain**[14]. He was first to formulate the **superposition principle** by which rocks are successively laid down on older layers.

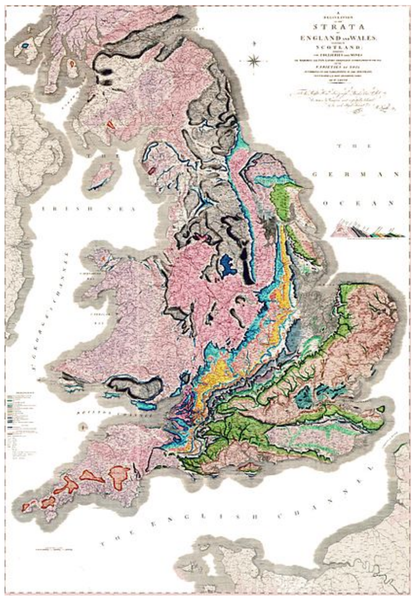


Figure 25. An early geological map of Britain showing different strata. (Source: Wikipedia.)

It is a **similar layer-by-layer approach** in computer science that allows one to design complex systems with hundreds of millions of components. In fact, layered approach is rather common in many **software architectures**, as shown in Figure 26.

[14] The map is also known as "the map that changed the world". While this was not the world's first geological map, William Smith's map was the first geologic map covering such a large area ever published.

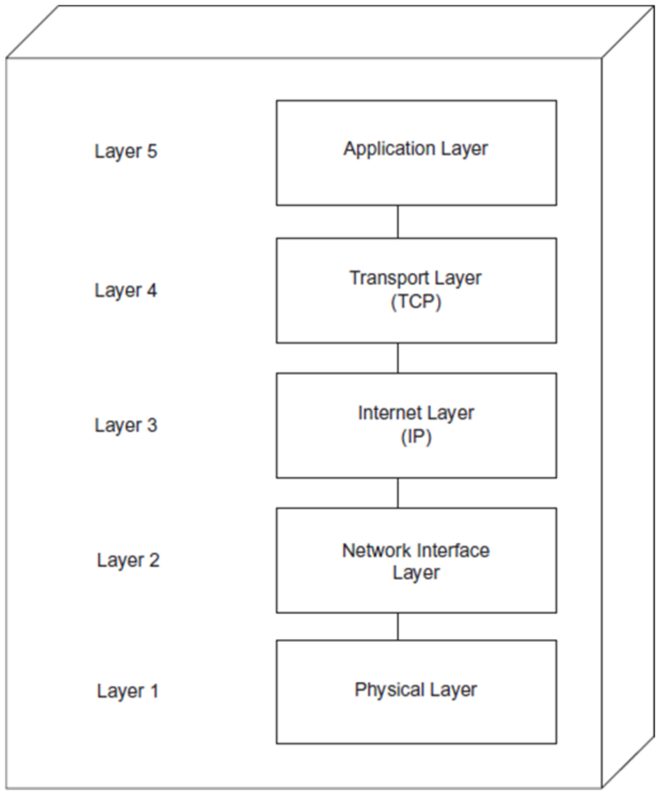


Figure 26. The **LAYERS OF ABSTRACTION** pattern in UML. (Source: [Gomaa, 2011].)

ABSTRACTION AND SOFTWARE ENGINEERING

The goals of software applications are so much more ambitious than they used to be because we are now able to do so much more than we used to be able to do. And what we've been able to do has improved so much because we have built **abstractions** to aid us.
— Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener

The **history of software engineering** is the history of **increasing the level of abstraction** over time [Papajorgji, Pardalos, 2014, Section 1.1]. (In some sense, this means that the **'distance' between the human and the machine** on which software would run **increased** over time.)

Table 1 presents a per-decade snapshot of the **history of software engineering from the perspective of abstraction**.

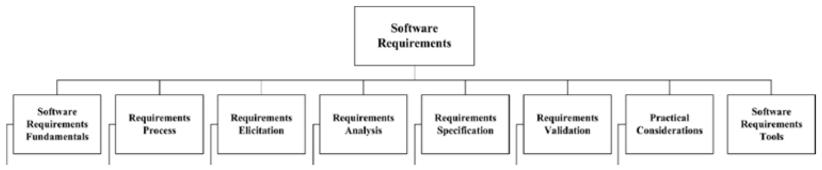
Circa	**Event**	**Description**
1940	Assembly Language	The mnemonics were a higher level of abstraction over the 0's and 1's of the machine language. Impact: • Quality of Process (Productivity (Less Time Consuming, Less Error-Proneness)) • Quality of Product (Reduction in Errors)
1950	3GL	The macroinstructions were a higher level of abstraction over the microinstructions (composed of 0's and 1's of the machine language). Impact: • Quality of Product (Portability)
1960	Structured Programming	The decision statements were abstractions of control mechanisms (such as looping and `if-then` statements) incorporated in certain high-level programming languages. Impact: • Quality of Process (Predictability (Top-Down Decomposition), Maintenance) • Quality of Product (Reduction in Errors)
1970	Abstract Data Types	The abstract data types hid the implementation of the structure of that data. The abstract data types were incorporated in certain high-level programming languages. Impact: • Quality of Process (Productivity, Maintenance)
1980	Object-Orientation	The object-oriented approach abstracted knowledge from the domain and encapsulated it into objects. Each object had one or more responsibilities. The objects communicated with each other in order to make use of each other's capabilities. Impact: • Quality of Process (Productivity (Manage Complexity), Maintenance)

1990	Design Patterns	The design patterns abstracted a combination of solution to a recurring problem in a given context. The use of design patterns in software engineering moved the level of abstraction higher, closer to the problem level, and away from the machine language level. Impact: • Quality of Process (Productivity (Reusability))
2000	UML	The different aspects of a system (including object-oriented design) were abstracted and modeled visually in a manner that was independent of any particular programming language. Impact: • Quality of Process (Productivity (Reusability), Communication with Stakeholders)

Table 1. An assortment of events in the history of abstraction in software engineering.

SWEBOK®

The **Guide to the Software Engineering Body of Knowledge (SWEBOK)** [IEEE, 2014] has a number of Knowledge Areas (KAs). These KAs can be represented at **different levels of abstraction**, as shown in Figure 27.



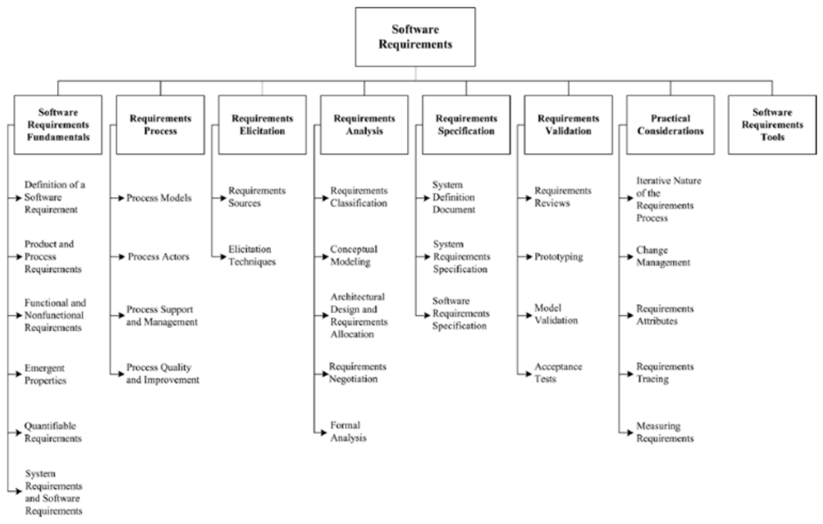

Software Requirements
Software Requirements Fundamentals
Requirements Process
Requirements Elicitation
Requirements Analysis
Requirements Specification
Requirements Validation
Practical Considerations
Software Requirements Tools
Definition of a Software Requirement
Product and Process Requirements
Functional and Nonfunctional Requirements
Emergent Properties
Quantifiable Requirements
System Requirements and Software Requirements
Process Models
Process Actors
Process Support and Management
Process Quality and Improvement
Requirements Sources
Elicitation Techniques
Requirements Classification
Conceptual Modeling
Architectural Design and Requirements Allocation
Requirements Negotiation
Formal Analysis
System Definition Document
System Requirements Specification
Software Requirements Specification
Requirements Reviews
Prototyping
Model Validation
Acceptance Tests
Iterative Nature of the Requirements Process
Change Management
Requirements Attributes
Requirements Tracing
Measuring Requirements

Figure 27. The Software Requirements Knowledge Area in the SWEBOK, at two different levels of abstraction. (Source: SWEBOK3 [IEEE, 2014].)

ABSTRACTION AND MODELS

The essence of the **Model-Driven Architecture (MDA)**, a framework for Model-Driven Software Development, is abstraction [Kleppe, Warmer, Bast, 2003; Mellor, Scott, Uhl, Weise, 2004; Brambilla, Cabot, Wimmer, 2012]. MDA has applications in various areas, as shown in Figure 28.

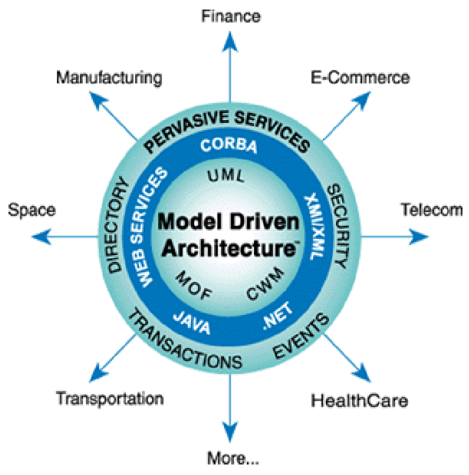


Figure 28. The applications of Model-Driven Architecture (MDA). (Source: OMG.)

ABSTRACTION AND USER INTERFACE DESIGN

EXAMPLE 1

It is possible to present elements of a user interface, such as a **menu**, at different levels of abstraction, at different times of interaction, as shown in Figure 29.

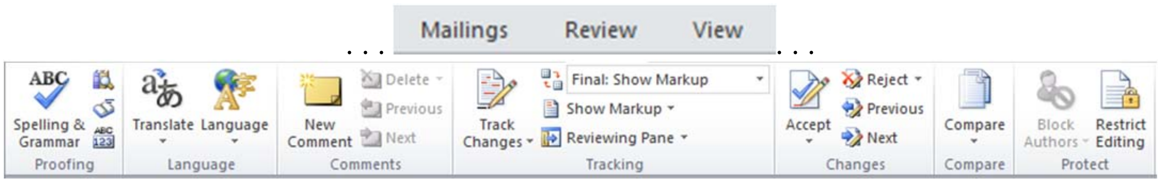
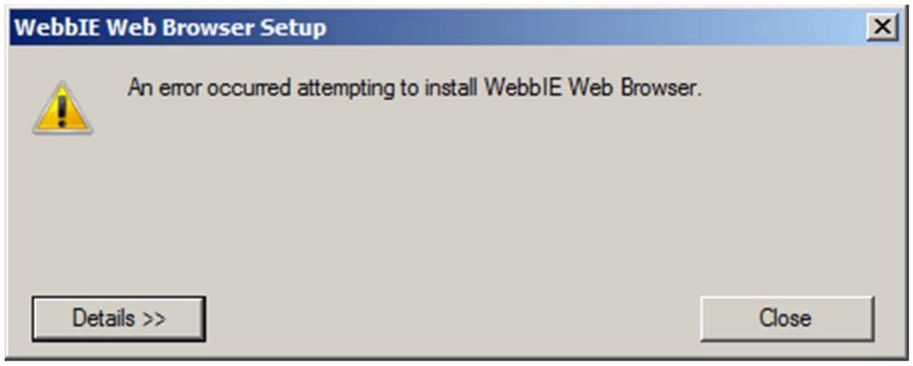


Figure 29. The top-level menu in Microsoft Word abstracts the items in it as well as the lower-level menus in it. (The lower-level menus do the same, **recursively**.)

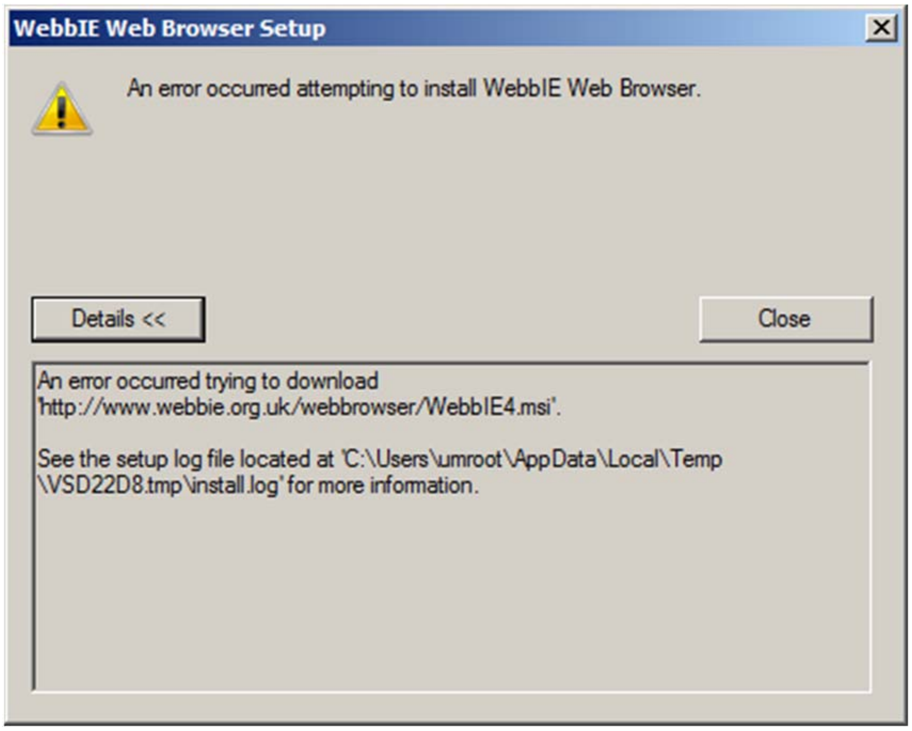
EXAMPLE 2

It is possible to present **error messages at different levels of abstraction**, as shown in Figure 30.


WebbIE Web Browser Setup

An error occurred attempting to install WebbIE Web Browser.

(a)



An error occurred attempting to install WebbIE Web Browser.

```
An error occurred trying to download
'http://www.webbie.org.uk/webbrowser/WebbIE4.msi'.

See the setup log file located at 'C:\Users\umroot\AppData\Local\Temp
\VSD22D8.tmp\install.log' for more information.
```

(b)

Figure 30. An error message with (a) less details, aimed perhaps towards a novice or non-technical user, and (b) more details, aimed perhaps towards an expert or technical user.

SIGNIFICANCE OF ABSTRACTION IN SOFTWARE ENGINEERING

It has been pointed out that **abstraction is the 'key'** to several aspects in software engineering [Kramer, Hazzan, 2006; Booch, Maksimchuk, Engle, Young, Conallen, Houston, 2007] and related disciplines, including **conceptual modeling** [Daniels, 2002; Lieberman, 2007; Boca, Bowen, Siddiqi, 2010, Chapter 1], **authoring patterns** [Rising, 2007], and so on. For example, it has been pointed out that **abstraction is the key to designing 'good' software** [Wirfs-Brock, Wilkerson, Wiener, 1990, Page 4].

There are numerous examples of abstractions in software engineering.

For example, **cost estimation** of a new application is an **abstraction of the predicted reality** (number of engineers, expected size of the project, schedule of deliverables, and so on), **software models** are abstractions of reality (in the sense that UML models are simplified descriptions of the actual structure and behavior of a software system to be developed) [Snoeck, 2014, Section 1.3.1], **inheritance relationship in a UML Class diagram** is based on the idea that a generalized class is more abstract than a specialized class, and so on.

It is the principle of abstraction that allows the separation of problem domain concerns from solution domain concerns.

SAME REALITY, DIFFERENT ABSTRACTIONS

For the **same reality**, there can be **different abstractions**. Indeed, creating **multiple levels of abstraction** is one way to manage complexity (specifically, **control complexity**).

EXAMPLE 1

The **same software system** can be modeled by **different modeling languages**. In such a case, each model provides a view of the software system for some purpose.

EXAMPLE 2

A UML Class Diagram can be expressed at (least) **three different levels** of abstraction, as shown in Figure 31, where each level has a different purpose.

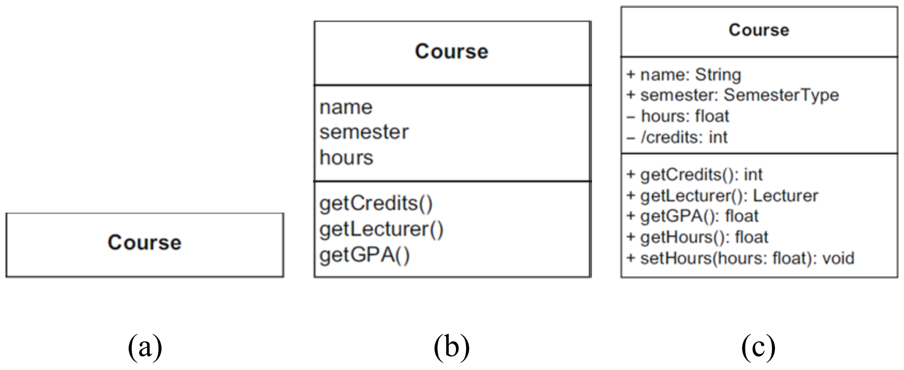
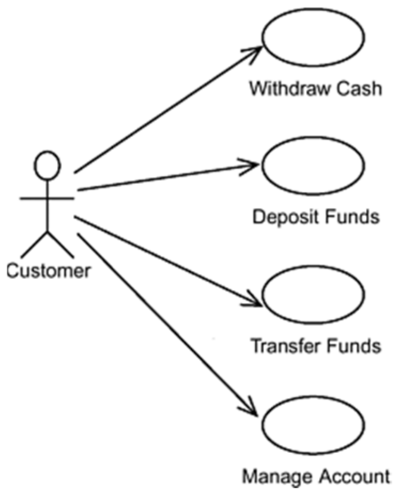
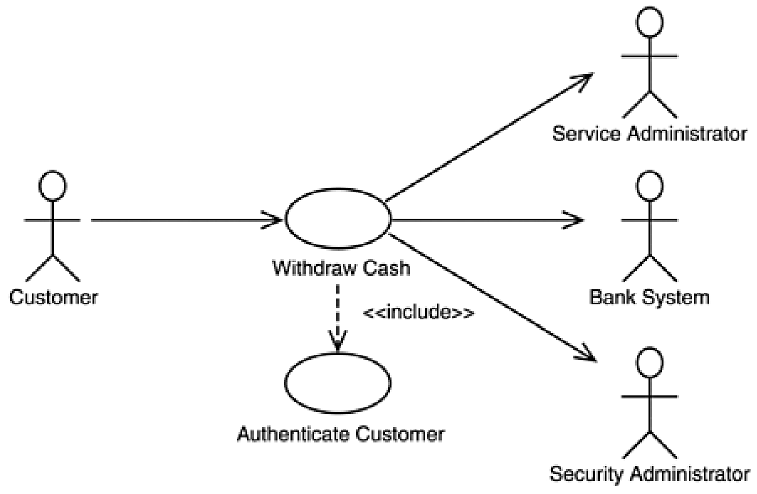


Figure 31. A UML Class Diagram at different levels of abstraction. (Source: [Seidl, Scholz, Huemer, Kappel, 2015].)

EXAMPLE 3

A UML Use Case Diagram can be expressed at (least) **two different levels** of abstraction, as shown in Figure 32, where each level has a different purpose.



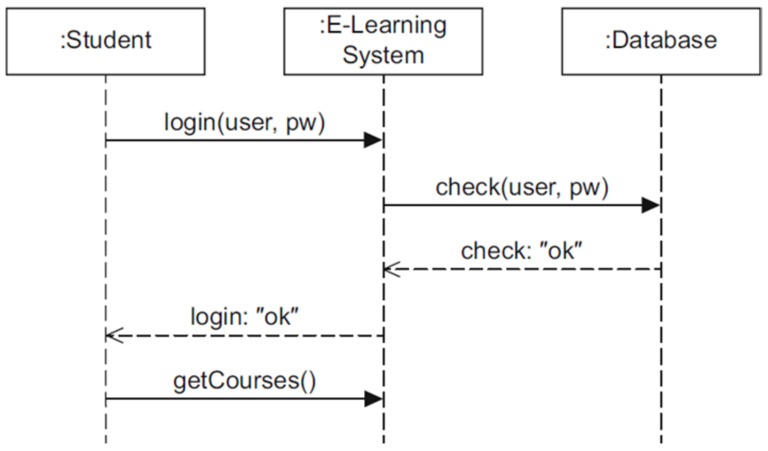


(a) (b)

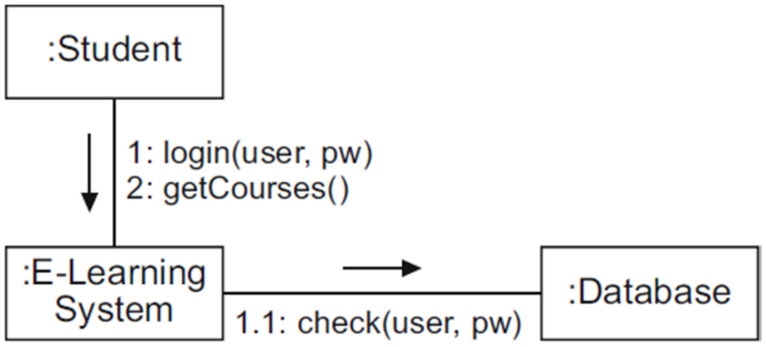
Figure 32. A UML Use Case Diagram at different levels of abstraction. (Source: [Bittner, Spence, 2002].)

EXAMPLE 4

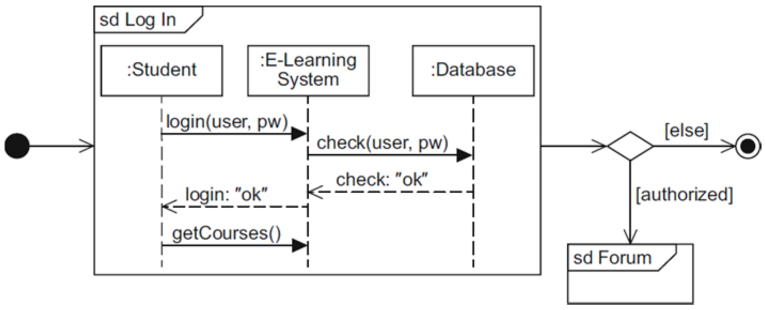
A UML Sequence Diagram, a UML Communication Diagram, and a UML Interaction Overview Diagram, each describe the **same situation**, namely a login process, but with a **different emphasis**, as shown in Figure 33.



(a)



(b)



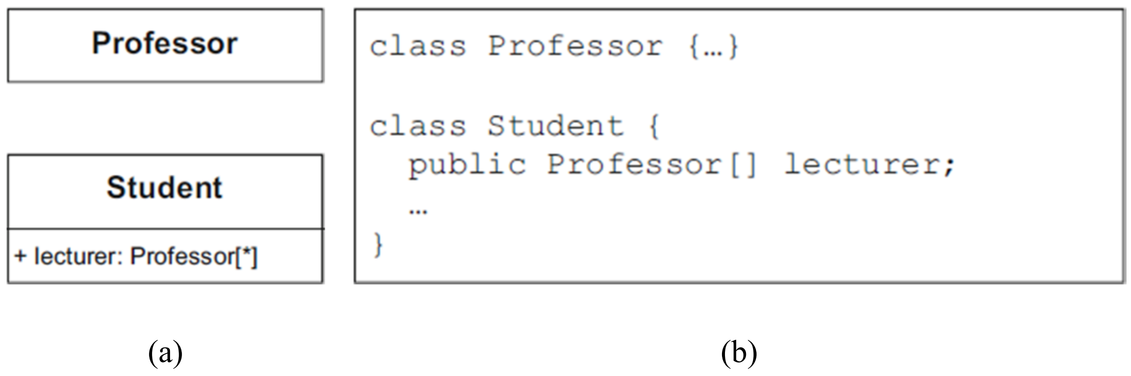
(c)

Figure 33. The same reality represented by two different types of UML interaction diagrams, (a), (b), and (c). (Source: [Seidl, Scholz, Huemer, Kappel, 2015].)

EXAMPLE 5

A **design** can be viewed as an **abstraction of implementation**.

The UML Class Diagram can be used to model design, and an object-oriented programming language, such as Java, can be used to model implementation, as shown in Figure 34.



```
class Professor {…}

class Student {
  public Professor[] lecturer;
  …
}
```

Figure 34. (a) A UML Class Diagram in which the student-professor relationship is modeled and represented by an attribute in the class `Student`, and (b) the design in (a) is translated into an implementation in Java. (Source: [Seidl, Scholz, Huemer, Kappel, 2015].)

WHAT IS AND WHAT IS NOT ABSTRACT, AND FOR WHOM

For the same reality, what is abstract and what is not (detail) depends on the **purpose** of abstraction. The purpose of abstraction varies with stakeholders (such as architects, designers, programmers, maintainers, and users).

For example, the **comments in SGML or XML** are supposed to be **hidden from the users, but not from the maintainers**.

FROM ABSTRACT TO CONCRETE, AND BACK

The process of moving from **abstract to concrete** is called **reification**, and moving from **concrete to abstract** is called **dereification**.

EXAMPLE

In UML, an association class can be represented in an abstract form and in a corresponding concrete form, as shown in Figure 35.

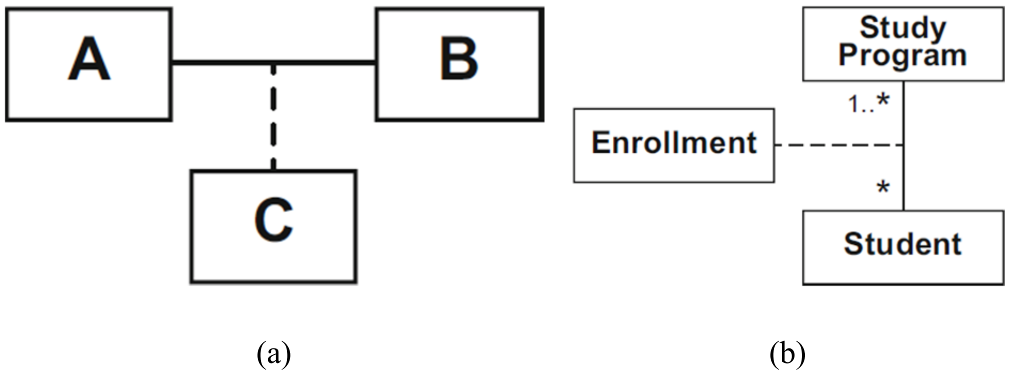


Figure 35. (a) An abstraction of an association class, and (b) its concretization. (Source: [Seidl, Scholz, Huemer, Kappel, 2015].)

The act of **reification and dereification** is, for example, common in moving between a model and its serialization, respectively.

ABSTRACTION VERSUS INFORMATION HIDING

In computer programming and software engineering literature, abstraction and information hiding [Parnas, 1972] are sometimes used **interchangeably**.

The significance of information hiding has been given in [McConnell, 1996].

LIMITATIONS

There are evident limits of abstraction [Fine, 2008]. It is simply **not possible to abstract everything** that is relevant, or an **abstraction may not have any meaning** to some.

For example, certain **human emotions** (say, happiness) or **behavioral states** (say, comfortableness) are relevant to **interactive software systems**, but they do not lend themselves easily to abstraction.

For example, the color **red** has **no meaning** for somebody who **does not have the required sensory functionality**. (The person in question may have a type of red color blindness.)

12. PRINCIPLE 5. GENERALITY

We think in generalities, but we live in detail.
— Alfred N. Whitehead

The principle of generality is based on the premise that a **generalized problem might be more widely applicable than the original problem**. The solutions to a general problem tend to be more reusable than the original problem.

In general, let P_1 and P_2 be two programs that solve the **same problem**. If everything possible by P_1 is possible by P_2 (but not conversely), then P_2 is more general than P_1.

A PROCESS OF GENERALITY

It is said that **ideas precede their conceptualization**. In order to achieve generality, it is a common practice to move from **specific to general**, although it may not always appear as such. (For example, it is not all that useful to have a generalization relationship between two use cases when the parent is an abstract use case and there is only one child (that is, the inherited or specialized) use case.)

EXAMPLE 1

There are numerous examples of generality in **industrial engineering**. For example, it is known that manufacturers produce **automobiles, hair care products, luggage**, and so on, for the consumption by **general public**.

EXAMPLE 2

In software engineering, a **specific accounting problem** can be generalized to a **spreadsheet**; **specific users** can be generalized to **actors** (as defined in a use case context); and a program that can **add a natural number and an integer** can be generalized to **add two complex numbers**.

EXAMPLE 3

The quadratic equation is given by $ax^2 + bx + c = 0$, where x is a variable; a, b, and c are constants; and $a \neq 0$.

Let P be a program to **compute the solutions of the quadratic equation**. There are many possible Ps. Usually, programs are made more general by accommodating **larger class of possible inputs**. In this sense, the **generality** of P can be understood on (at least) two dimensions: (1) the **type** (say, natural number, integer, real number) of a, b, and c, and (2) the relative **size** (say, small or large) of a, b, and c.

EXAMPLE 4

It could be noted that generality is **not an 'automatic' or 'obvious' property** [Gustafsson, 2011, Section 2.1].

For the polynomial equation

$$a_0 + a_1x + a_2x^2 + \cdots + a_nx^n = 0,$$

there exist close-form, explicit, solutions for n = 1 (monic), 2 (quadratic), 3 (cubic), and 4 (quartic), but **not** for n > 4. (This indeed is a consequence of the **Abel-Ruffini Theorem**.)

CURRENT TREND IN SOFTWARE ENGINEERING

It appears that generality is a trend for both commercial and non-commercial software products. For example, Adobe Photoshop, IBM Eclipse, IBM/Rational Rose Enterprise, Microsoft Office, and Mozilla Firefox have become increasingly general.

LIMITATIONS

There are certain side-effects of generality.

TEORIYA RESHENIYA IZOBRETATELSKIKH ZADATCH (TRIZ) OR "I NEVER PROMISED YOU A ROSE GARDEN"

The following examples are manifestations of **TRIZ**, a theory for **inventive problem solving** [Altshuller, 1994].

In some cases, **generality may come at the cost of development**. For example, a program that is able to handle a broader range of inputs is general. However, such a program may also be more **costly** to develop as it may consume more resources, such as personnel and time.

In some cases, **generality may come at the price of (loss of) efficiency**. For example, a computer program that espouses to be general should minimize the use of system calls to the underlying operating system. However, doing so, in turn, may sacrifice desired efficiency.

To summarize, it is **not always possible to have everything, all the time**. TRIZ is echoed in the following rule: **There Ain't No Such Thing As A Free Lunch (TANSTAAFL)**.

13. PRINCIPLE 6. ANTICIPATION OF CHANGE

Intelligence is the ability to adapt to change.
— Stephen Hawking

When it is not necessary to change,
It is necessary not to change.
— Lord Falkland

Change is inevitable — except from a vending machine.
— Robert C. Gallagher

"Change is Good, Donkey!"
— Shrek 2 (2004)

The principle of anticipation of change is based on the premise that change is **inherent** to software.

The goal of anticipation of change is to improve evolvability (maintainability). A change in the software product is part of software maintenance and, more generally, change management is part of **software management**.

UNDERSTANDING CHANGE

In an organic view of software, it is understood that software resides in an **ecosystem**, and both software and its ecosystem are evolving. Therefore, change due to evolution is **inevitable**. Furthermore, **the question of change is not 'if', but 'when'**.

CLASSIFYING CHANGE

There are various **types of change**: **market demand changes, requirements change, assessment/perception of quality changes (say, after errors are discovered), team configuration changes, information technology changes**, and so on.

There are different **ways to categorize change**: **internal or external, controllable or uncontrollable**, and so on. For example, a team member's non-participation in a software project due to sickness or accident, or variation in stock market, is an uncontrollable change.

SIGNIFYING CHANGE

It is not merely the change itself, but the **severity of impact** of change that is of concern and therefore of interest. (A change that has positive probability of occurrence, and can lead to potentially adverse circumstance, is considered a **risk**.)

EXAMPLE

For the sake of argument, consider comments C in source code S. Then a **lexical error** (say, typographical error) in C may not have a severe impact on S; however, a **semantic error** (say, C do not match the section of S to which they are supposed to apply) can have a severe impact towards maintenance.

SIGNIFICANCE OF ANTICIPATION OF CHANGE (OR "SKATING TO WHERE THE PUCK WILL BE")

[If] we are living in a world of rapid change, reflection in action needs to be balanced with anticipating the future.
— Barry Boehm

If a change is not anticipated, then the **cost of dealing with the impact of change** can be **prohibitive**.

In this document, portability and reusability are specific cases of evolvability. A software component is rarely **reused** as-is; it usually requires minor modifications or accommodations. The software components that do not conform to anticipation of change are **resistant to reuse** in other situations.

CAUSE-AND-EFFECT

Figure 36 shows a **general cause-and-effect diagram**, also known as the **Fishbone Diagram or Ishikawa Diagram**. Figure 37 and Figure 38 show specific examples of cause-and-effect diagram.

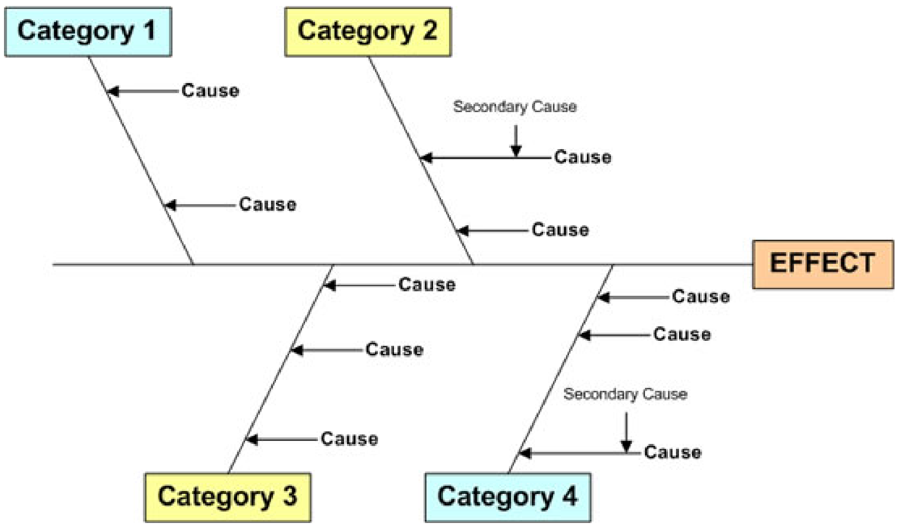


Figure 36. A general cause-and-effect diagram. (Source: Google Images.)

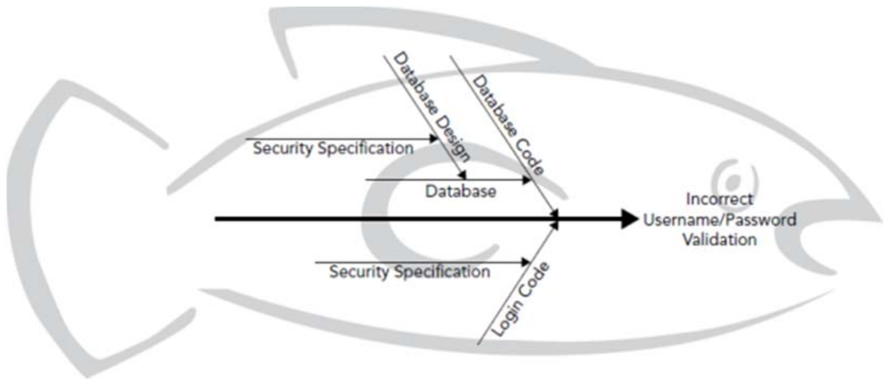


Figure 37. A specific cause-and-effect diagram. (Source: [Stephens, 2015, Chapter 10].)

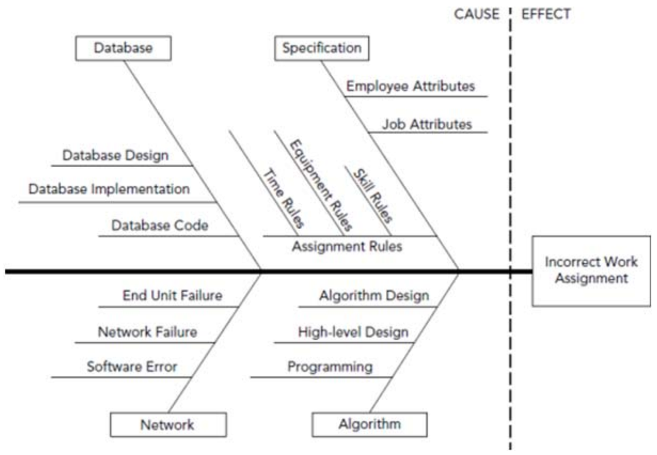


Figure 38. A specific cause-and-effect diagram. (Source: [Stephens, 2015, Chapter 10].)

There is a close relationship between change and **causality**, the philosophy of **cause-and-effect**. This is illustrated in the Table 2. It could be noted that for a **single effect**, there can be **multiple causes**.

Cause	**Effect**
Departure of Team Member	Delay in Schedule
Arrival of a New Competitor	Loss of Revenue
Poorly Commented Source Code	Difficulty in Human Interpretation
Strongly (Highly) Coupled Modules	Difficulty in Modification
Fault	Failure
(Source Code) Smell	Technical Debt

Table 2. An assortment of examples of cause-and-effect.

ANTICIPATION OF CHANGE AND 'ELEMENTS' OF SOFTWARE ENGINEERING

Most importantly, do not forget that you are **writing code for people, not for the compiler**. People, yourself included, will have to **maintain your code** going forward, so make everyone's life easier, including your own, and write testable JavaScript.
— Mark Ethan Trostler

The ability of software to evolve is **not automatic**. Therefore, a **preparation** for potential change is an **imperative**. This requires an anticipation of **how** and **where** the changes are likely to occur.

There are specific ways to **prepare for change** in different, but related, 'elements' of software engineering, as shown in Figure 6:

- **Project.** An acknowledgement of change, followed by support and decision making. These decisions include [Lethbridge, Laganière, 2005] **avoiding** commitments to new releases of technologies, **avoiding** commitments to libraries that are specific to particular environments, and **avoiding** commitments to software or special hardware from organizations that are less likely to provide long-term support.

- **Process.** A monitoring, modularization, and recording of the process (that includes making salient decisions **explicit**).

- **People.** The existence of documentation, including contingency plans, and training in risk management.

- **Product.** The placement of artifacts under a **configuration management system (CMS)**, modularization of design, avoidance of hidden dependencies in design, source code annotation (including comments), and so on. It is suggested that software artifacts (documents, data, models, source code) be created under the assumption that they will be **read**, **used**, or **modified** by **someone else**. Indeed, **Write Code as If You Had to Support It for the Rest of Your Life** is among the **97 Things Every Programmer Should Know** [Henney, 2010].

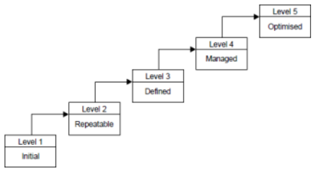
LIMITATIONS

There is a need to monitor change and take action if there is change. This can be enduring and not always humanly-feasible in certain software projects.

The realization of anticipation of change in practice and in a controlled manner requires appropriate **tools**.

For example, deployment of a CMS, in general, and version control system, in particular, can help monitor change and can assist in managing change. However, as with other tools, there are **indirections** in a commitment to a CMS.

14. PRINCIPLE 7. RIGOR



The principle of rigor aims to **assist and improve human reasoning**.

The purpose of rigor is to **induce order in software development**. It is a move away from **chaos** (approaches that are ad-hoc, based on intuition and heuristics) to **order** (approaches that embrace discipline, and are systematic). In other words, rigor is part of **making software engineering 'engineering'**.

EXAMPLE

In general, rigor is achieved by conforming to certain given **constraints**.

For example, one common approach to induce rigor is to enforce the use of **standards** (as defined by a standards organization) related to, say, software development process and/or product.

LIMITATIONS

The side-effects of rigor, depending on enforcement, such as it could **constrain independent thinking unnecessarily** and **limit flexibility**. This could be the case if the **rules are followed for the sake of following rules**, regardless of reflecting on their benefits.

To some, it may also seem that a commitment to **standards stifles creativity**.

15. PRINCIPLE 8. FORMALITY

The principle of formality aims to **assist and improve both human as well as machine reasoning**. In case of a machine, reasoning is based on a mechanical (specifically, mathematical) set of rules.

In general, formality can range on a discrete **spectrum**:

- Informal
- Partially-Formal
- Formal

- **Informal: Non-Mathematical Syntax, Non-Mathematical Semantics.** For example, a (structured) statement in a natural language, such as a user story statement, is considered as informal.

- **Partially-Formal: Mathematical Syntax, Not-Entirely-Mathematical Semantics.** For example, a UML model, a BNF grammar, an algorithm pseudocode, source code of a computer program, a DSSSL style sheet, an SGML DTD, and an XML schema are considered as partially-formal.

- **Formal: Mathematical Syntax, Mathematical Semantics.** For example, a schema in Z specification language is considered as formal.

REMARKS

Semi-Formal (Mathematical Syntax, Non-Mathematical Semantics) is a **special case** of Partially-Formal.

LIMITATIONS

The side-effects of formality include that it may not be appealing to non-technical stakeholders, and it loses its ‘charm’ if **automation** (usually, machine processing) is not possible or is prohibitively expensive.

For the same software system, some parts can be rigorous and other parts can be formal, with varying degrees.

16. CONCLUSION

The best designers sometimes disregard the principles of design. When they do so, however, there is usually some compensating merit attained at the cost of the violation. Unless you are certain of doing as well, it is best to abide by the principles.
— William Lidwell, Kritina Holden, Jill Butler

The SEP presented here are technically-oriented. It is possible that for certain domains, the aforementioned seven SEP are **insufficient**. For example, this is the case for Web Engineering [Kamthan, 2008]. Indeed, some other collections [Davis, 1995] do deal with human- and/or socially-oriented aspects of software engineering.

In conclusion, the SEP are **fundamental** to software engineering **theory and practice**. If SEP appear to be 'common sense' or 'simplistic', then that is because of the tremendous time and effort that has undergone before their proposal.

ACKNOWLEDGEMENT

This document has benefited from the works of Gregory Butler and Peter Grogono. The inclusion in this document of an image from an external source is only for non-commercial educational purposes, and its use is hereby acknowledged.

REFERENCES

[Aiello, Sachs, 2011] Configuration Management Best Practices: Practical Methods that Work in the Real World. By B. Aiello, L. Sachs. Addison-Wesley. 2011.

[Alexander, Ishikawa, Silverstein, 1977] A Pattern Language: Towns, Buildings, Construction. By C. Alexander, S. Ishikawa, M. Silverstein. Oxford University Press. 1977.

[Altshuller, 1994] And Suddenly the Inventor Appeared. By G. Altshuller. Technical Innovation Center. 1994.

[Arbib, 1987] Brains, Machines and Mathematics. By M. A. Arbib. Second Edition. Springer-Verlag. 1987.

[Baecker, Marcus, 1990] Human Factors and Typography for More Readable Programs. By R. M. Baecker, A. Marcus. Addison-Wesley. 1990.

[Beck, Andres, 2005] Extreme Programming Explained: Embrace Change. By K. Beck, C. Andres. Second Edition. Addison-Wesley. 2005.

[Biffl, Aurum, Boehm, Erdogmus, Grünbacher, 2006] Value-Based Software Engineering. By S. Biffl, A. Aurum, B. Boehm, H. Erdogmus, P. Grünbacher (Editors). Springer-Verlag. 2006.

[Bittner, Spence, 2002] Use Case Modeling. By K. Bittner, I. Spence. Addison-Wesley. 2002.

[Boca, Bowen, Siddiqi, 2010] Formal Methods: State of the Art and New Directions. By P. P. Boca, J. P. Bowen, J. I. Siddiqi (Editors). Springer-Verlag. 2010.

[Boehm, 1983] Seven Basic Principles of Software Engineering. By B. W. Boehm. Journal of Systems and Software. Volume 3. Issue 1. 1983. Pages 3-24.

[Booch, Maksimchuk, Engle, Young, Conallen, Houston, 2007] Object-Oriented Analysis and Design with Applications. By G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Conallen, K. A. Houston. Third Edition. Addison-Wesley. 2007.

[Bourque, Dupuis, Abran, Moore, Tripp, Wolff, 2002] Fundamental Principles of Software Engineering - A Journey. By P. Bourque, R. Dupuis, A. Abran, J. W. Moore, L. Tripp, S. Wolff. Journal of Systems and Software. Volume 62. Issue 1. 2002. Pages 59-70.

[Brambilla, Cabot, Wimmer, 2012] Model-Driven Software Engineering in Practice. By M. Brambilla, J. Cabot, M. Wimmer. Morgan and Claypool. 2012.

[Brooks, 1987] No Silver Bullet: Essence and Accidents of Software Engineering. By F. P. Brooks, Jr. Computer. Volume 20. Number 4. 1987. Pages 10-19.

[Brooks, 1995] The Mythical Man-Month: Essays on Software Engineering. By F. P. Brooks, Jr. Second Edition. Addison-Wesley. 1995.

[Buschmann, Meunier, Rohnert, Sommerlad, Stal, 1996] Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. By F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. John Wiley and Sons. 1996.

[Cockburn, 2001] Writing Effective Use Cases. By A. Cockburn. Addison-Wesley. 2001.

[Cockburn, 2008] Using Both Incremental and Iterative Development. By A. Cockburn. CrossTalk. May 2008. Pages 27-30.

[Curtis, 2010] Modular Web Design: Creating Reusable Components for User Experience Design and Documentation. By N. Curtis. New Riders. 2010.

[Daniels, 2002] Modeling with a Sense of Purpose. By J. Daniels. IEEE Software. Volume 19. Issue 1. 2002. Pages 8-10.

[Davis, 1994] Fifteen Principles of Software Engineering. By A. M. Davis. IEEE Software. Volume 11. Issue 6. 1994. Pages 94-96.

[Davis, 1995] 201 Principles of Software Development. By A. M. Davis. McGraw-Hill. 1995.

[Dietrich, Killisperger, Stückl, Weber, Hartmann, Kern, 2013] Using UML 2.0 for Modelling Software Processes at Siemens AG. By S. Dietrich, P. Killisperger, T. Stückl, N. Weber, T. Hartmann, E.-M. Kern. In: Information Systems Development Reflections, Challenges and New Directions. R. Pooley, J. Coady, H. Linger, C. Barry, M. Lang, C. Schneider (Editors). Springer Science+Business Media. 2013. Pages 561-572.

[Dijkstra, 1982] On the Role of Scientific Thought. By E. W. Dijkstra. In: Selected writings on Computing: A Personal Perspective. E. W. Dijkstra (Editor). Springer-Verlag. Pages 60-66.

[El-Attar, Miller, 2006] Matching Antipatterns to Improve the Quality of Use Case Models. By M. El-Attar, J. Miller. The Fourteenth International Requirements Engineering Conference (RE 2006). Minneapolis-St. Paul, U.S.A. September 11-15, 2006.

[Fehling, Leymann, Retter, Schupeck, Arbitter, 2014] Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications. By C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. Springer-Verlag. 2014.

[Fine, 2008] The Limits of Abstraction. By K. Fine. Oxford University Press. 2008.

[Forward, Lethbridge, 2008] A Taxonomy of Software Types. By A. Forward, T. C. Lethbridge. School of Information Technology and Engineering. Technical Report: TR-2008-06. University of Ottawa. Ottawa, Canada. 2008.

[Fowler, 2001] Reducing Coupling. By M. Fowler. IEEE Software. Volume 18. Issue 4. 2001. Pages 102-104.

[Garrett, 2011] The Elements of User Experience: User-Centered Design for the Web and Beyond. By J. J. Garrett. Second Edition. New Riders. 2011.

[Ghezzi, Jazayeri, Mandrioli, 2003] Fundamentals of Software Engineering. By C. Ghezzi, M. Jazayeri, D. Mandrioli. Second Edition. Prentice-Hall. 2003.

[Golbeck, 2013] Analyzing the Social Web. By J. Golbeck. Elsevier. 2013.

[Gomaa, 2011] Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures. By H. Gomaa. Cambridge University Press. 2011.

[Gustafsson, 2011] Fundamentals of Scientific Computing. By B. Gustafsson. Springer-Verlag. 2011.

[Hayes, 2003] Do You Like Piña Coladas: How Improved Communication Can Improve Software Quality. By J. H. Hayes. IEEE Software. Volume 20. Number 1. 2003. Pages 90-92.

[Hazzan, Dubinsky, 2014] Agile Anywhere: Essays on Agile Projects and Beyond. By O. Hazzan, Y. Dubinsky. Springer. 2014.

[Henney, 2010] 97 Things Every Programmer Should Know: Collective Wisdom from the Experts. By K. Henney (Editor). O'Reilly Media. 2010.

[Hey, Pápay, 2015] The Computing Universe: A Journey through a Revolution. By T. Hey, G. Pápay. Cambridge University Press. 2015.

[IEEE, 2014] Guide to the Software Engineering Body of Knowledge (SWEBOK) Version 3.0. IEEE Computer Society. 2014.

[ISO/IEC/IEEE, 2010] ISO/IEC/IEEE 24765:2010. Systems and Software Engineering -- Vocabulary. The International Organization for Standardization (ISO)/The International Electrotechnical Commission (IEC)/IEEE Computer Society. 2010.

[ISO/IEC/IEEE, 2011] ISO/IEC/IEEE 42010:2011. Systems and Software Engineering -- Architecture Description. The International Organization for Standardization (ISO)/The International Electrotechnical Commission (IEC). 2011.

[Jacobson, Booch, Rumbaugh, 1999] The Unified Software Development Process. By I. Jacobson, G. Booch, J. Rumbaugh. Addison-Wesley. 1999.

[Kamthan, 2008] Towards Formulation of Principles for Engineering Web Applications. By P. Kamthan. In: Encyclopedia of Internet Technologies and Applications. M. Freire, M. Pereira (Editors). IGI Global. 2008. Pages 640-646.

[Katz, 2012] Designing Information: Human Factors and Common Sense in Information Design. By J. Katz. John Wiley and Sons. 2012.

[Kleppe, Warmer, Bast, 2003] MDA Explained: The Model Driven Architecture™: Practice and Promise. By A. Kleppe, J. Warmer, W. Bast. Addison-Wesley. 2003.

[Kramer, 2007] Is Abstraction the Key to Computing? By J. Kramer. Communications of the ACM. Volume 50. Issue 4. 2007. Pages 36-42.

[Kramer, Hazzan, 2006] The Role of Abstraction in Software Engineering. By J. Kramer, O. Hazzan. ACM SIGSOFT Software Engineering Notes. Volume 31. Issue 6. November 2006. Pages 38-39.

[Lethbridge, Laganière, 2005] Object-Oriented Software Engineering: Practical Software Development using UML and Java. By T. C. Lethbridge, R. Laganière. Second Edition. McGraw-Hill. 2005.

[Lieberman, 2007] The Art of Software Modeling. By B. A. Lieberman. Auerbach Publications. 2007.

[Machiavelo, 2012] On the Importance of Useless Mathematics. By A. Machiavelo. In: Raising Public Awareness of Mathematics. E. Behrends, N. Crato, J. F. Rodrigues (Editors). Springer-Verlag. 2012. Pages 397-408.

[McConnell, 1996] Missing in Action: Information Hiding. By S. McConnell. IEEE Software. Volume 13. Issue 2. 1996. Pages 127-128.

[McConnell, 1999] Software Engineering Principles. By S. C. McConnell. IEEE Software. Volume 16. Issue 2. 1999. Pages 6-8.

[Mellor, Scott, Uhl, Weise, 2004] MDA Distilled: Principles of Model-Driven Architecture. By S. J. Mellor, K. Scott, A. Uhl, D. Weise. Addison-Wesley. 2004.

[Meridji, Abran, 2010] Software Engineering Principles: Do They Meet Engineering Criteria? By K. Meridji, A. Abran. Journal of Software Engineering and Applications. Volume 3. Number 10. 2010. Pages 972-982.

[Meyer, 2014] Agile! The Good, the Hype and the Ugly. By B. Meyer. Springer International Publishing. 2014.

[Millett, Tune, 2015] Patterns, Principles, and Practices of Domain-Driven Design. By S. Millett, N. Tune. Wrox Press. 2015.

[Övergaard, Palmkvist, 2005] Use Cases: Patterns and Blueprints. By G. Övergaard, K. Palmkvist. Addison-Wesley. 2005.

[Papajorgji, Pardalos, 2014] Software Engineering Techniques Applied to Agricultural Systems: An Object-Oriented and UML Approach. By P. J. Papajorgji, P. M. Pardalos. Second Edition. Springer Science+Business Media. 2014.

[Parnas, 1972] On the Criteria To Be Used in Decomposing Systems into Modules. By D. L. Parnas. Communications of the ACM. Volume 15. Number 12. 1972. Pages 1053–1058.

[Parnas, 1978] Some Software Engineering Principles. By D. L. Parnas. In: State of the Art Report on Structured Analysis and Design. INFOTECH International. 1978. Pages 237-247.

[Rising, 2007] Understanding the Power of Abstraction in Patterns. By L. Rising. IEEE Software. Volume 24. Issue 4. 2007. Pages 46-51.

[Royce, 1970] Managing the Development of Large Software Systems: Concepts and Techniques. By W. W. Royce. In: Technical Papers of Western Electronic Show and Convention (WesCon). Los Angeles, U.S.A. August 25-28, 1970.

[Pugh, 2005] Prefactoring: Extreme Abstraction, Extreme Separation, Extreme Readability. By K. Pugh. O'Reilly Media. 2005.

[Salminen, Tompa, 2011] Communicating with XML. By A. Salminen, F. Tompa. Springer. 2011.

[Séguin, Tremblay, Bagane, 2012] Agile Principles as Software Engineering Principles: An Analysis. By N. Séguin, G. Tremblay, H. Bagane. The Thirteenth International Conference on Agile Processes in Software Engineering and Extreme Programming (XP 2012). Malmö, Sweden. May 21-25, 2012.

[Seidl, Scholz, Huemer, Kappel, 2015] UML @ Classroom: An Introduction to Object-Oriented Modeling. By M. Seidl, M. Scholz, C. Huemer, G. Kappel. Springer International Publishing. 2015.

[Stephens, 2015] Beginning Software Engineering. By R. Stephens. John Wiley and Sons. 2015.

[Stevens, Myers, Constantine, 1974] Structured Design. By W. P. Stevens, G. J. Myers, L. L. Constantine. IBM Systems Journal. Volume 13. Issue 2. 1974. Pages 115-139.

[Streekmann, 2012] Clustering-Based Support for Software Architecture Restructuring. By N. Streekmann. Springer Fachmedien. 2012.

[Van Hilst, Fernandez, 2010] A Pattern System of Underlying Theories for Process Improvement. By M. Van Hilst, E. B. Fernandez. The Seventeenth Conference on Pattern Languages of Programs (PLoP 2010). Reno, U.S.A. October 16-18, 2010.

[Vogel, Arnold, Chughtai, Kehrer, 2011] Software Architecture: A Comprehensive Framework and Guide for Practitioners. By O. Vogel, I. Arnold, A. Chughtai, T. Kehrer. Springer-Verlag. 2011.

[Wang, 2008] Software Engineering Foundations: A Software Science Perspective. By Y. Wang. Auerbach Publications. 2008.

[Wirfs-Brock, Wilkerson, Wiener, 1990] Designing Object-Oriented Software. By R. Wirfs-Brock, B. Wilkerson, L. Wiener. Prentice-Hall. 1990.