Milan Polytechnic

Software: development models in household appliance applications

Prof. William Fornaciari

Politecnico di Milano - Dept. of Electronics and Information william.fornaciari@polimi.it

Milan Polytechnic

Embedded systems: the appliance case

- Embedded systems
  - Dedicated to specific applications, programming or reprogramming impossible or infrequent, not end-user in general
  - Optimisation of many requirements, e.g. cost, performance, size, power, time to market, resource constraints, reliability, safety, maintainability, ...
  - Pervasive, ubiquitous, invisible... they are now consumer, included in white goods
- Complex design
  - Limited resources, interacting sw and hw: how to integrate?
  - Do I need an operating system? How do I select it?
  - Time-to-market: first project often 'messy' and low-cost, poorly structured methodology with reuse difficulties
  - Use of outsourcing: problem of controlling know-how and product quality. One has to manage evolution of product families
  - Off the shelf vs. custom: do you accept risks? Which technologies?
  - Current students know too much high-end sw technologies, but the market is different... e.g. the washing machine is not programmed in Java (for now!)

Sw embedded: appliance case

- Objectives
  - Small memories and computing power
  - Limited reparability/upgradability
  - Hard and soft real time
  - Simplified user interfaces
  - Historical legacy
  - Costs
  - still costs!
- Constraints
  - Reliability
  - Security
  - Evolution over time
  - Specialisation on similar product families
  - Decoupling of user interface and control
  - Don't base success only on a designer's talent

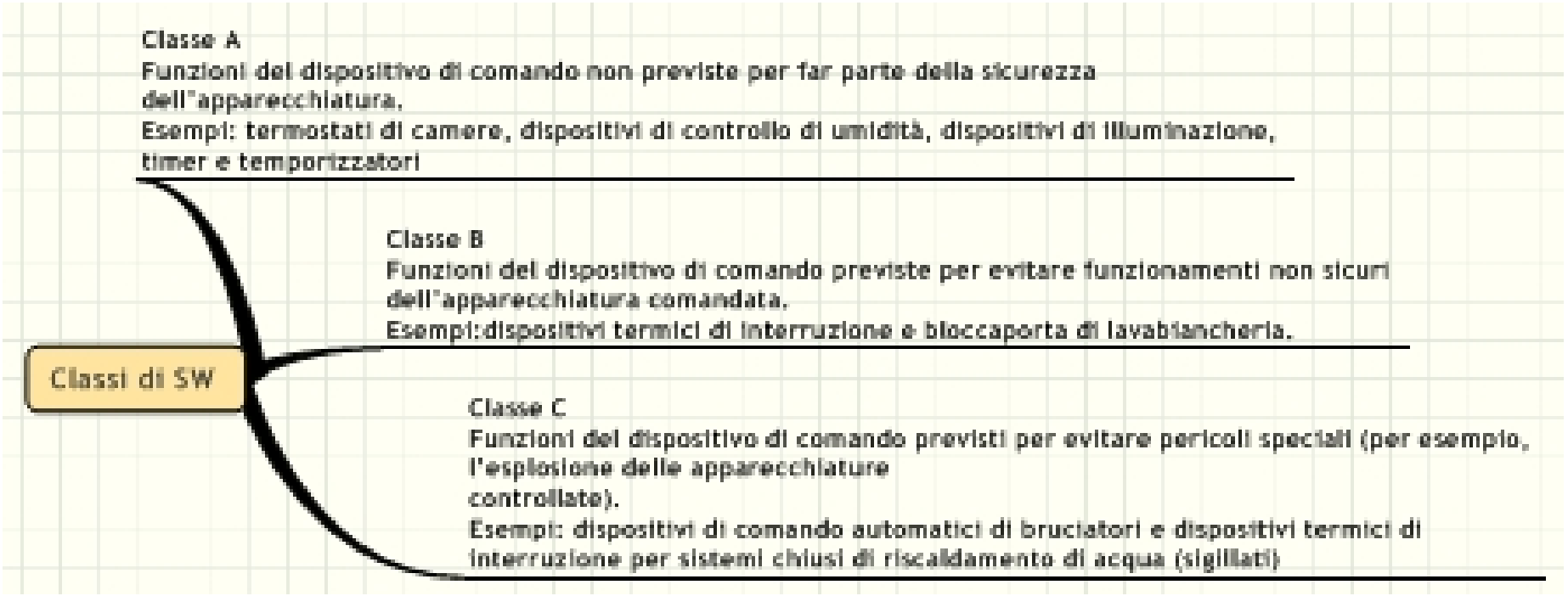
Does it drive the market or regulation?

Example: Automatic electrical control devices for domestic and similar use

- **General Prescriptions**
  - The manufacturer must provide adequate information to confirm this:
  - that a suitable control device is chosen;
  - that the control device can be mounted and used in such a way that it can meet the requirements of this standard;
  - and that appropriate tests can be carried out to ensure compliance with this Standard.

- **Methods of providing information**
  - Information must be provided using one or more of the following methods:
  - With Marking (C): provided by marking on the control device itself, except in the case of an integrated control device, where the marking may be on an adjacent part of the equipment, provided it is clear that it refers to the control device
  - **With Documentation (D): provided to the user or installer of the control device, and must consist of legible instructions.**
  - **With Declaration (X): provided to the person responsible for testing for the purpose of testing and in the manner agreed between him and the manufacturer.**

Software classes

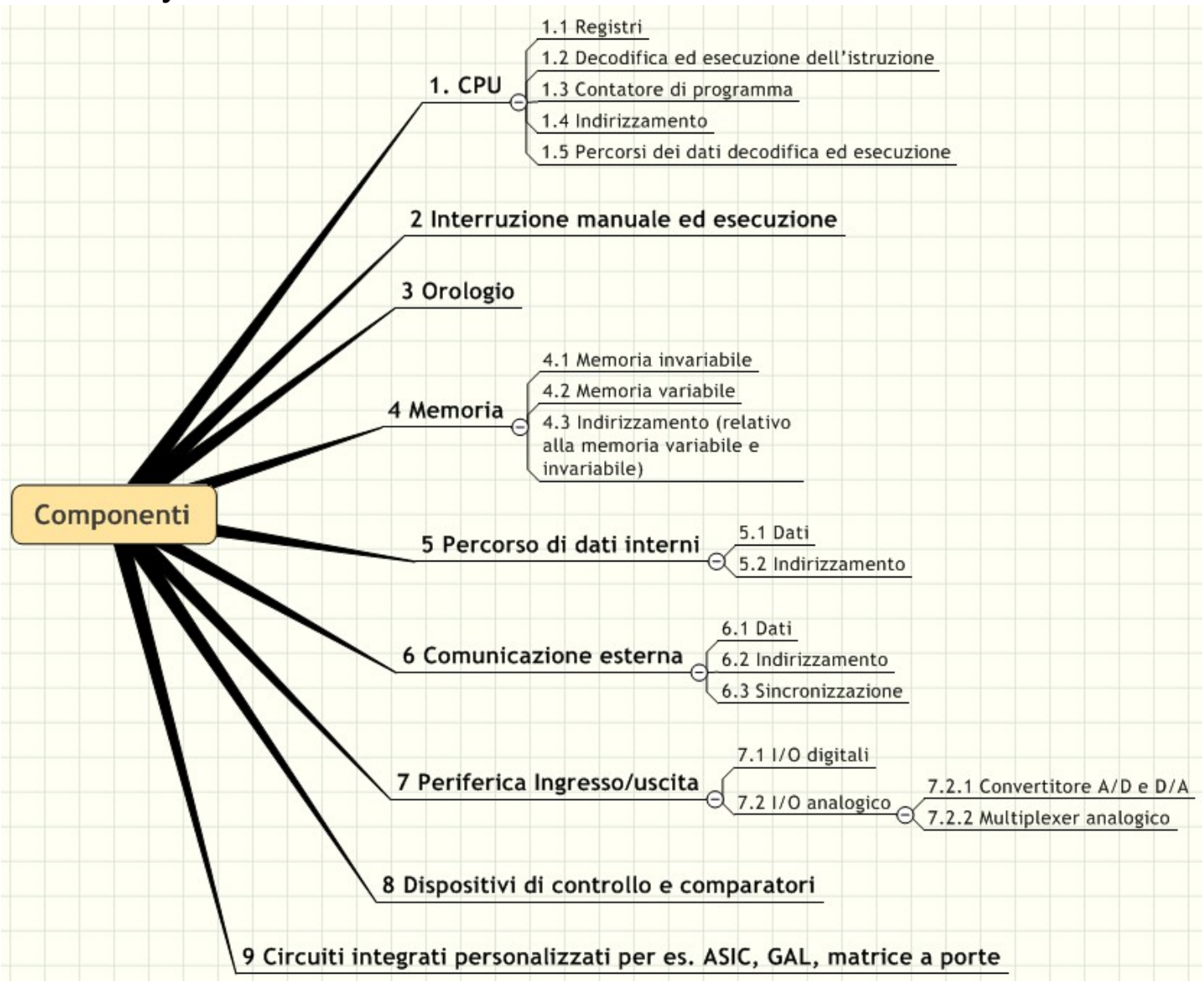


Milan Polytechnic

Software Doc: Example of Requirements .....there's a lot to do!

- **Documentation of the SW sequence**
- **Programme documentation**
- For controllers with a function declared as Class B or C software, information must only be provided for safety-related software segments. Information on non-safety-related segments must be sufficient to establish that they do not affect the safety-related segments.
- The software sequence must be documented and, together with the sequence of operation of prescription 46 which must include a description of the control system philosophy, device scope, data flow and synchronisations.
- Programming documentation must be provided in the programming language declared by the manufacturer.
- The data and safety segments of the software sequence, the malfunctions of which could cause non-compliance with requirements 17, 25, 26 and 27, must be identified. This identification must encompass the sequence of operation and may, for example, take the form of a fault tree analysis which must include the faults/errors of Tab. H.11.12.7 that can be derived from this nonconformity. The software fault analysis must relate to the hardware fault analysis H.27.
- Examples of other information that can be included in documentation for software systems:
  - Functional specifications including a reset procedure following power failure.
  - Study of the programme module including a description of the equipment and user interfaces.
  - Detailed study, including description of memory utilisation.
  - Code list including programming language identification, comments and list of subroutines.
  - Test specifications.
  - Installation, operation and/or maintenance manual

Example of failure analysis



The software (regardless of motivation)

- Secure code
  - the first step is quality and comprehensibility
- Documentation Development
  - software for embedded applications is no less complex than the others... on the contrary
- Code quality
  - Style and testing
  - Use of standardised development tools and flows

The source code (1)

- The code is used to **communicate**
  - with the machine
  - with other developers
  - with testers
  - with certifiers (?)
- The written code must be able to be read by others
  - It is read by people more often than by machines
- The code can also have a long life
  - The more useful the code, the more it will be read for longer and by more people
- The meaning and purpose of the written code may not be easily understood
  - the developer is also responsible for the comprehensibility of the code

The source code (2)

- The code we write must be maintainable
- It must be relatively easy to modify the code to change its behaviour
  - The problems for which we write code, or the environment in which it operates, evolve over time, so the code must be adapted
  - The extensibility of an application is also an internal property
  - Defects must be corrected
- The most successful programmes are used in many different areas and are therefore also the ones that require the most maintenance

The source code (3)

- The code must be reliable
  - error is inevitable
  - trust in code written by others
  - trust that well-written code indicates well thought-out code
- Performance
  - Fast but incomprehensible code can be improved with much effort
  - Understandable code can be optimised
- Do not waste resources, including programmers' time
  - If developers deal with well-written code, they save time

Software quality (1)

- What is software quality
- Software certification is easier if its quality is higher (??)
- The quality aspects of the code to be considered are:
  - Robustness or reliability
  - Readability
  - Fairness
- Robustness -> development process
- Readability -> programming style
- Correctness -> testing

Software robustness (1)

- Reliability is one of the aspects of software that must be taken into account when determining its quality

- Although the term 'quality' refers to a subjective assessment, software reliability can be 'measured' through objective criteria, called metrics

- Due to the increasing pervasiveness that embedded software has achieved in today's devices, flaws in the software are causing increasingly significant inconveniences

Software robustness (2)

- The need to objectively measure software quality stems from the need to apply techniques from traditional engineering fields, such as reuse, to software development
- The need arises from both a technical and a legislative point of view
  - The software must not exhibit undesirable and unintended behaviour
  - Software defects can cause damage to stored data, to the system on which the software runs or even to people, in the case of embedded systems in transport and medical equipment
- Apart from the criticality of the individual software application, the pervasiveness of software is increasing and, if the growth rate is confirmed, we will reach a point where software will be the crucial element of tomorrow's society
  - If tomorrow's society will rely on software, its quality will have to live up to expectations

Milan Polytechnic

Reliability in the development process (1)

- Requirements
  - In order to develop an application correctly, the programmer must know its expected behaviour, at least in parallel with the development phase and with an adequate level of detail
    - Achieving the appropriate level of detail may not be technically or economically feasible
    - Refining the requirements too much wastes time and resources
    - Not refining the requirements to an adequate level leads to an application that does not know how to react to unforeseen situations
- Design
  - Design specifies how a programme should be made, at least at a high level
  - High-level design makes it possible to separate the problems encountered in defining architecture, such as programme structure and concepts, from coding problems, which are aimed at processing information

Reliability in the development process (2)

- Programming languages
  - The evolution of programming languages seeks to provide tools that allow more and more work to be delegated to computers, reducing the possibility of introducing defects on the part of the developer
    - The introduction of virtual machines enables the detection of software defects at run-time
  - In other words, the evolution of programming languages must make it possible to handle increasingly complex and larger programmes
- Testing
  - Given a programme or part of a programme, tests can be performed, either manually or automatically, to determine whether the software behaves as dictated by the requirements
    - The automatic execution of test suites makes it easy to check whether changes made to a programme have changed its correctness

Code quality

- When can we say that the code is of quality?
- How can quality be 'measured'?
- Metrics exist:
  - automatically verifiable
  - verifiable through inspection
- Or:
  - Process metrics, which are calculated by introducing 'sensors' into the development process
  - Code metrics, calculated directly on source code
- In general, code is of quality when it has features and a structure that
  - do not promote
  - do not hide
  - help to avoid

  the introduction of defects into the code, both when the code is released and when it has to be modified/maintained

Process metrics

- Process metrics are the most objective, as they measure quality from the effects it generates
- Fairness measurement
  - Number of defects found after software release
- Measure of maintainability
  - Average time taken to correct a detected defect or to introduce a new feature
- These quantities are difficult to measure automatically
  - Require a change to the development process in order to track quantities of interest
  - Require developers to collaborate on the measurement of quantities

Code Metrics

- When metrics process cannot be used, we rely on code metrics
  - These metrics are less objective than process metrics, as it is possible to influence their value without
- Is the code documented?
  - A metric can be: number of comment lines / number of total lines
  - This metric may not be reliable
    - the developer can influence the measurement

```
/* The function calculates the lowest common
denominator */
void mcd(int * a);
```

```
/* The function calculates
the lowest common denominator */
void mcd(int * a);
```

Improving robustness

- Metrics may provide a quantitative indication of robustness, but they do not improve it
- Robustness can be improved through
  - Static code analysis
  - Code Inspection
- Static code analysis makes it possible to detect possible 'dangerous situations' that could become software defects at run-time
  - For example, compiler-generated warnings represent the most rudimentary form of static analysis
- There are sets of code writing rules that aim to produce more robust code
  - For example, there are MISRA rules that favour robustness and portability
  - There are commercial tools that perform automatic verification of MISRA rules
  - MISRA rules are numerous: incremental adoption is recommended

Example of MISRA rules

Software maintenance can introduce errors

```
int foo( int a )
{
      if( a > 10 )
            return 0;
      else
            return a++;
}
```

```
int foo( int a )
{
      if( a > 10 )
            return 0;
      else
            a = a + 2;
            return a;
}
```

Example of MISRA rules

This rule forces the introduction of curly brackets also in code blocks consisting of a single instruction

This rule prevents the introduction of possible errors during maintenance

```
int foo( int a )
{
	if( a > 10 )
	{
		return 0;
	}
	else
	{
		return a++;
	}
}
```

```
int foo( int a )
{
	if( a > 10 )
	{
		return 0;
	}
	else
	{
		a = a + 2;
		return a;
	}
}
```

Inspection

- Code inspection consists of periodic meetings between developers who 'look at' certain software sections
  - There is a check-list of checks that developers must perform on the code to be inspected
- Inspection cannot be applied to all software, as it is excessively time-consuming
- It is suggested that code inspection sessions be held on the most complex or the most reused or the most recent modules
  - Allows the detection of defects, errors or inconsistencies
  - Standardises the formatting style among developers
  - It helps the development team to get used to the MISRA rules that project managers have decided to adhere to

Milan Polytechnic

Improved readability (1)

Miglioramento della leggibilità (1)

- A development team, or even the entire organisation, must have precise rules about the style of writing code
- The style rules in code writing are called 'style patterns'.
  - These are practices concerning pure code writing, not design
  - Often, they are supported by the IDE
  - Often, they are verifiable
- The whole code is written according to the same rules, so it is easier for the reader to orientate himself
- Style patterns must be shared and adopted by all members of the development team

Improved readability (2)

Miglioramento della leggibilità (2)

- Initially, it might be useful to flank the patterns stylistics, an automatic formatting tool
  - Such tools allow 'forcing' a programming style and can compensate for any 'oversights' or old habits of programmers
  - Possible freely distributed tools
    - AStyle, easy to use, but inflexible
    - indent, very flexible and configurable, but a little more complex

```
int foo(int a){
        if(a>10)
                return 0;
        else
                return a++;
}
```

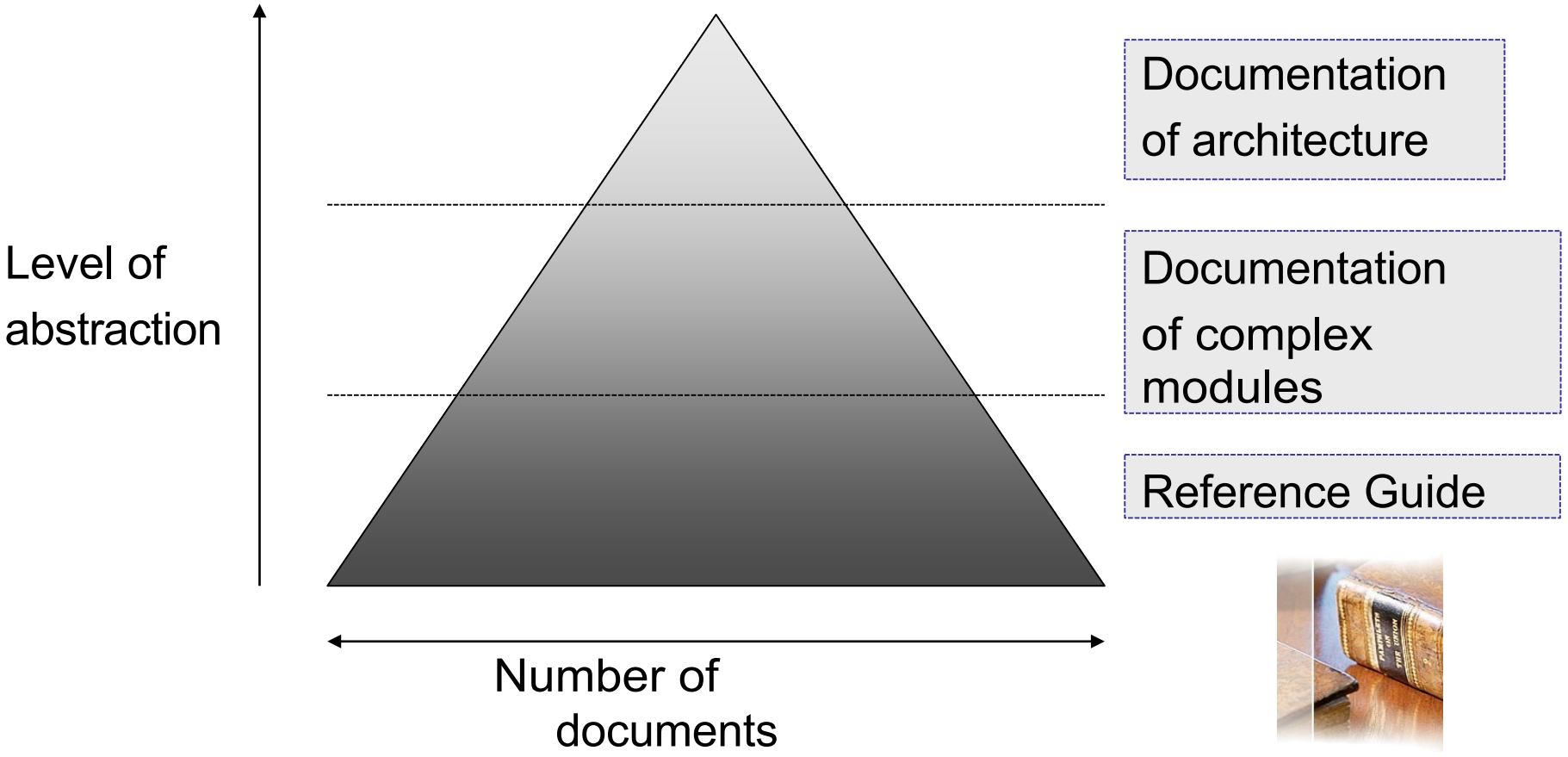
```
int foo( int a )
{
        if( a > 10 )
                return 0;
        else
                return a++;
}
```

Software Documentation

- The software certification process relies on documentation
- The documentation can be classified into:
  - Project documentation
  - Reference Guide
  - Documentation of test cases

Improving documentation

- The documentation should mainly cover three aspects:



Documenting the system architecture

- Describes the project as a whole
- It can also include comprehensive descriptions of the simplest modules
- Any detailed cross-module information can be 'confined' in appendices
- The update frequency of this document must be low
  - If, for the sake of consistency, the document undergoes continuous and frequent updates, this means that
    - Includes too much detailed information
    - Architecture is still evolving strongly

Improvement of documentation More detailed documents

- In order not to overburden the discussion of the architecture, more complex modules, or groups of related modules or entire sub-systems can be explained in separate documents - For example, the communication protocol could be
  documented in a separate document
- It is considered appropriate not to provide a document for each module, in order to avoid the proliferation of documents with low information content
- The update frequency of these documents may be higher than that of the architecture documentation, although it does not have to correspond to that of software changes

Improvement of documentation
The reference guide

- The reference guide is a document detailing all data types and functions throughout the platform
- Usually, this type of documentation has a very rapid rate of obsolescence, especially during the code development phases
- Therefore, the need arises to use appropriate tools to automatically generate the reference guide
- Use of automatic documentation generation tools from comments
  - Doxygen simply requires a formatting of comments
  - Many modules already have meaningful comments
    - Simply add the tags for Doxygen and the reference guide is ready
  - Doxygen can generate browser-navigable html documentation
  - Doxygen can generate documents in rtf or latex, to produce manuals
  - You can also have inclusion graphs, function call graphs and the class diagram generated