# 12 Software Implementation

The goal of most software engineering projects is to produce a working program.

The act of transforming the detailed design into a valid program in some programming language, together with all its supporting activities is referred to as implementation.

Most of the text in this chapter is taken from [16].

The implementation phase involves more than just writing code. Code also needs to be tested and debugged as well as compiled and built into a complete executable product (Figure 12-1).

We usually need to use a Source Code Control (SCC) tool to keep track of different versions of the code.

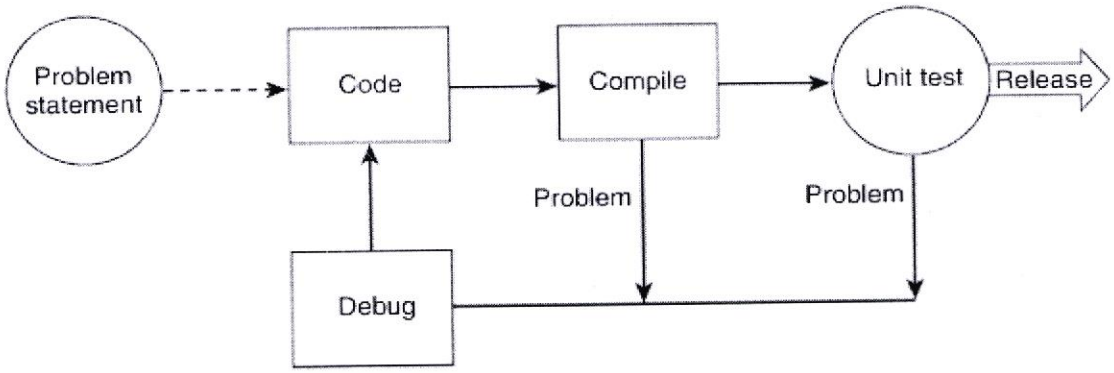



**Figure 12-1: Software Implementation [16]**

In many cases the detailed design is not done explicitly (in the Design Phase) but is left as part of the implementation. Doing the detailed design as part of the implementation is usually faster, but it may result in a less cohesive and less organized design, because the detailed design of each module will usually be done by a different person.

In small projects, the detailed design is usually left as part of the implementation. In larger projects, or when the programmers are inexperienced, the detailed design will be done by a separate person.

Here are some keywords for good implementation:

- **Readability**: The code can be easily read and understood by other programmers.
- **Maintainability**: The code can be easily modified and maintained. Note that this is related to readability, but it is not the same; for example, this involves the use of e.g., Hungarian notation, in which variable names include abbreviations for the type of variable.

- **Performance**: All other things being equal, the implementation should produce code that performs as fast as possible.
- **Traceability**: All code elements should correspond to a design element. Code can be traced back to design (and design to requirements).
- **Correctness**: The implementation should do what it is intended to do (as defined in the requirements and detailed design).
- **Completeness**: All the system requirements are met.

In this chapter, we will go through the following topics regarding implementation:

In this chapter, we will discuss the following:

- Programming Style and Coding Guidelines
- Comments
- Debugging
- Code Review
- Refactorization

## 12.1 Programming Style & Coding Guidelines

Almost all software development organizations have some sort of coding guidelines. These guidelines usually specify issues such as naming, indentation, and commenting styles, etc.

It is strongly recommended that you be consistent in your notation to avoid confusion when others are debugging or maintaining your code later. Especially in large software projects there are usually some programming conventions. These conventions may seem to be of little value at first, but they may become extremely helpful during the maintenance of the code.

Here are some recommendations:

- **Naming**: This refers to choosing names for classes, methods, variables, and other programming entities.
- **Separating words and capitalization**: Many times, a name will be composed of more than one word. In human languages, we use spaces to separate words, but most programming languages will not allow us to do so. (do_something, doSomething, DoSomething)
- **Indentation and spacing**: Indentation refers to adding horizontal spaces before some lines to better reflect the structure of the code. Spacing refers to both spaces and blank lines inserted in the code.
- **Function/method size**: Many studies have shown that large functions or methods are statistically more error-prone than smaller ones.

- **File-naming issues**: Having a standard for specifying how to name the files, which files to generate for each module, and how to locate a given file from a module is very advantageous.
- **Particular programming constructs**: Different programming languages support different features; although they usually have good reasons to include certain features, there are many that can be misused and need special precautions.

### 12.1.1 Naming Convention

We have different naming convention/notation such as:

- Camel notation
- Pascal notation
- Hungarian notation

**Camel Notation**

For variables and parameters/arguments

Examples: "myCar", "backColor"

**Pascal Notation**

For classes, methods and properties

Examples: "ShowCarColor"

**Hungarian Notation**

For controls on your user interface we either use "Pascal notation" or "Hungarian notation", but stick to one of them!

Examples: "txtName", "lblName"

**Acronyms**

Examples: "DBRate", "ioChannel", "XmlWriter", "htmlReader"

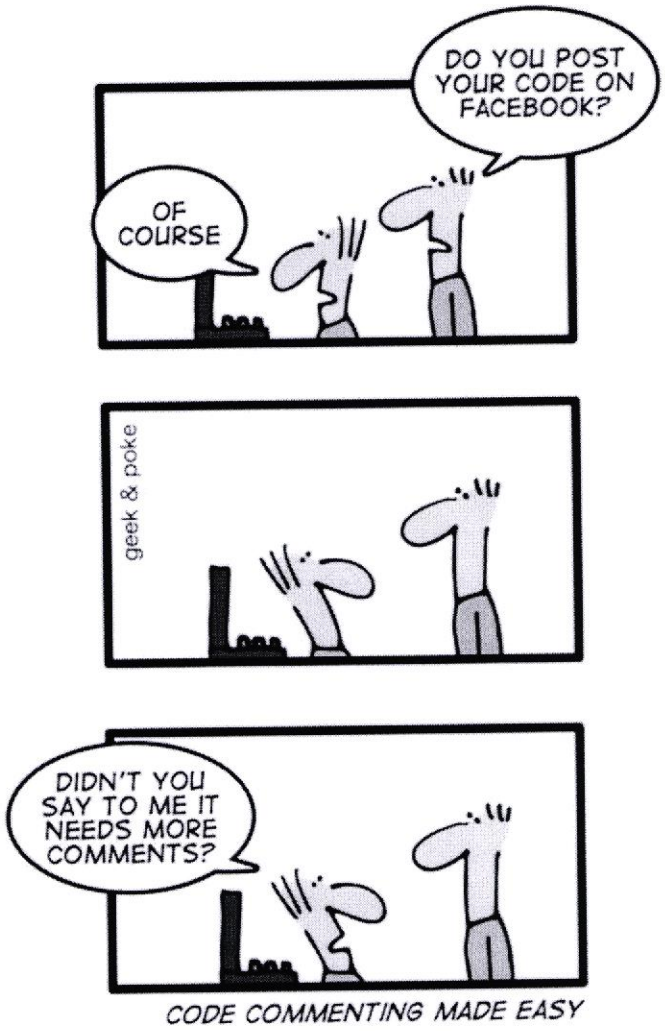
## 12.2 Comments

Comments are very important and can significantly aid or hurt readability and maintainability.

There are two main problems with comments:

- they may distract from the actual code and make the program more difficult to read and

- they may be wrong.




We may classify comments into 6 different types:

1. Repeat of the code
2. Explanation of the code
3. Marker in the code
4. Summary of the Code
5. Description of the code intent
6. External references

These are explained below:

- Repeat of the code

- These kinds of comments tend to be done by novice programmers and should be avoided.

  Bad Example:

  ```
  // increment i by one
  i++;
  ```

- Explanation of the code
  - Sometimes, when the code is complex, programmers are tempted to explain what the code does in human language.
  - In almost every case, if the code is so complex that it requires an explanation, then it should be rewritten.

- Marker in the code
  - It is common practice to put markers in the code to indicate incomplete items, opportunities for improvement, and other similar information.
  - We recommend using a consistent notation for these markers, and eliminating all of them before the code is in production.
  - Sometimes programmers put markers in the code to keep track of changes and who made them. We believe that information is better tracked with version management software, and recommend doing so.

- Summary of the code
  - Comments that summarize what the code does, rather than just repeating it, are very helpful in understanding the code, but they need to be kept up to date.
  - It is important to ensure that these comments are summarizing the code, not just repeating or explaining it.
  - In many cases, the code that is being summarized can be abstracted into its own function, which, if named correctly, will eliminate the need for the comment.

- Description of the code intent
  - These are the most useful kinds of comments; they describe what the code is supposed to do rather than what it does.
  - These are the only kinds of comments that override the code. If the code does not fulfill its intent, then the code is wrong.

- External references

- These are comments that link the code to external entities, usually books or other programs.
- Many times, these can be viewed as a kind of intent statement, as in, "This function implements the XYZ algorithm, as explained in . . .," but we believe such comments require special attention.
- There may also be external prerequisites for the code, such as the existence of initializing data in the database tables.

The trade-off that comments imply should be recognized. Comments can help clarify code and relate it to other sources, but they also represent some level of duplication of the code.

6. Effort is invested in their creation and, above all, in their maintenance.
7. A comment that does not correspond to the actual code that it accompanies can cause errors that are very hard to find and correct.
8. Another danger comments present is that they can be used to justify bad coding practices. Many times, programmers will be tempted to produce code that is too complicated or too hard to maintain, and add comments to it, rather than rewrite it to good standards.
9. In fact, many experts recommend avoiding comments completely, and produce what is called "self-documented code"—that is, code that is so well written that it does not need any documentation.
10. Comments have their place, especially in the form of describing the programmer's intent.

## 12.3 Debugging

Debugging is about different techniques for finding and fixing bugs (errors that make your code not work as expected) in your code.

1. It is difficult to write code without errors (bugs), but e.g., Visual Studio and other tools have powerful Debugging functionality (break-points, etc.)
2. The Compiler will also find syntax errors, etc.
3. For more "advanced" bugs other methods are required (Unit Testing, Integration Testing, Regression Testing, Acceptance Testing, etc.).
4. The focus here will be on these methods, while Debugging is something you learned in Programming courses.

In debugging we have 4 phases:

- Stabilization/Reproduction
    - The purpose of this phase is to be able to reproduce the error on a configuration, and to find out the conditions that led to the error by constructing a minimal test case
- Localization
    - The process of localization involves finding the sections of the code that led to the error. This is usually the hardest part, although, if the stabilization phase produces a very simple test case, it may make the problem obvious.
- Correction
    - The process of correction involves changing the code to fix the errors. Hopefully, if you understand what caused the error, you have a good chance of fixing the problem.
- Verification
    - The process of verification involves making sure the error is fixed, and no other errors were introduced with the changes in the code. Many times, a change in the code will not fix the error or may introduce new errors.

## 12.4 Code Review

We all are human beings. You may do some mistakes irrespective of your experience in a technology or module. If you just review your code by a second eye, those mistakes might have caught at that time only. This way you can reduce the no. of bugs reported by the testers or end users (Figure 12-2).

If you are working in a geographically distributed team, your coding conventions may differ and if you have some strict coding guidelines, this code review process will make it possible to recheck the standards in the code that you have written.

1. There are some possibilities of repetitive code block which can be caught during a code review process. Refactoring can be done based on that.
2. Unused code blocks, performance metrics etc. are some additional check points of doing a review.
3. If you are new to development, this code review process will help you to find out your mistakes and help you to improve them. This is a perfect knowledge sharing mechanism.

4. Find out the defects and correct them at the beginning before it commits to the source control system.

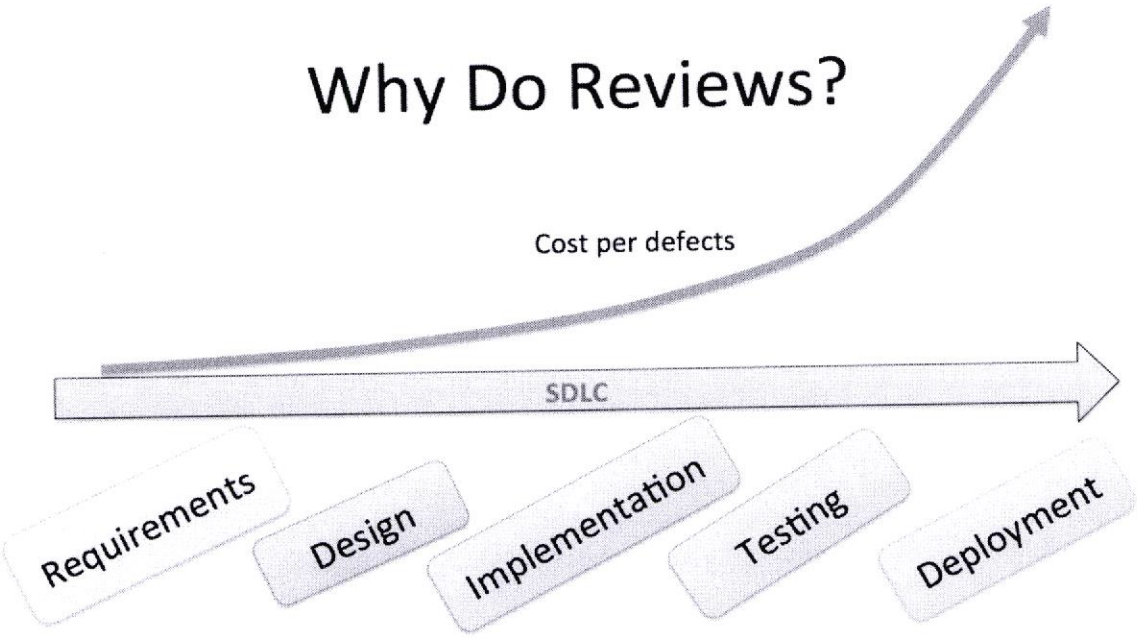
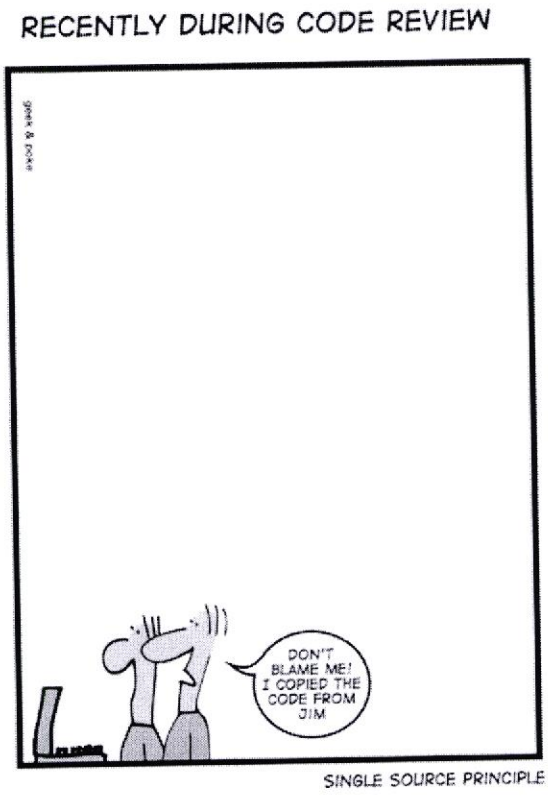



**Figure 12-2: Why you should do reviews**




Better code always starts with review process!

Here are some topics that should be checked during the Code Review process [12]:

- **Readability**: The code can be easily read and understood by other programmers.
- **Maintainability**: The code can be easily modified and maintained. Note that this is related to readability, but it is not the same; for example, this involves the use of e.g., Hungarian notation, in which variable names include abbreviations for the type of variable.
- **Performance**: All other things being equal, the implementation should produce code that performs as fast as possible.
- **Traceability**: All code elements should correspond to a design element. Code can be traced back to design (and design to requirements).
- **Correctness**: The implementation should do what it is intended to do (as defined in the requirements and detailed design).
- **Completeness**: All the system requirements are met.

## 12.5 Refactoring

Even when using best practices and making a conscious effort to produce high-quality software, it is highly unlikely that you will consistently produce programs that cannot be improved.

Refactoring is defined as

- the activity of improving your code style without altering its behavior
- a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior

Do you need to refactoring your code? – here are some symptoms:

- Coding Style and Name Conventions not followed
- Proper Commenting not followed
- Duplicated code (clearly a waste)
- Long method (excessively large or long methods perhaps should be subdivided into more cohesive ones)
- Large class (same problem as long method)
- Switch statements (in object-oriented code, switch statements can in most cases be replaced with polymorphism, making the code clearer)
- Feature envy, in which a method tends to use more of an object from a class different to the one it belongs

- Inappropriate intimacy, in which a class refers too much to private parts of other classes

Any of these symptoms (and more) will indicate that your code can be improved. You can use refactoring to help you deal with these problems.

You should Refactoring your continuously and especially after Code Reviews.