BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

SPECIALIZATION COMPUTER SCIENCE IN ENGLISH

DIPLOMA THESIS

DEVELOPING AN IDE FOR LINUX OPERATING SYSTEMS

Supervisor
Lect. univ. dr. Radu DRAGOŞ

Author
Marian VOMIR

2015

Table of Contents

1. Introduction

The purpose of this thesis is to present a research on Integrated Development Environments (IDEs) for the C programming language and to also present the approach taken when developing a brand new IDE that will be used for C development under Linux operating systems.

First developed between 1969 and 1973 in order to implement the UNIX operating system, C is still relevant today, after more than 40 years from its inception. Whether being used to develop operating systems, embedded software, even other programming languages or simply as an inspiration for the design of other programming languages, the popularity of this "quirky, flawed and enormously successful" [8] language cannot be overstated.

When developing C applications on Linux systems, there are a variety of Integrated Development Environments to choose from. We can see that IDEs, in general, have come a long way over the years by looking at the *Borland Turbo C* [22] IDE in 1987, which was one of the best for its time, and then coming back to the present day to look at products like *Code::Blocks* [17], *NetBeans* [18] and CLion [19] .

Even with the multitude of products that we can choose from, there is a small niche which not many are paying attention to. The popular IDEs are heavy-weight and are meant for real-world projects. In the university, however, students which are just starting to learn C still have to use one of these heavy-weight and feature-packed products in order to develop simple laboratory projects. It is safe to assume that, most likely, they will only use a small percentage of what these IDEs have to offer.

Students make mistakes in their programs, a lot of mistakes, and C is a language that is very easy to go wrong with. We could go as far as saying that it is unforgiving, especially comparing it to more modern languages such as Java or the very high-level Python. A simple and intuitive IDE with excellent code analysis and powerful diagnosis capabilities is essential for someone learning C (or even programming in general, for that matter).

Therefore, we need to come up with a concept for a **light-weight, portable and easy-to-use** IDE, which incorporates only a handful of highly optimized features. By *optimized* we can understand convenient, easy to use or helpful for a novice programmer.

The thesis is structured in five chapters.

Chapter 1, representing this introductory part, has outlined the context, the motivation and the main objectives of this thesis, while also mentioning the contributions of others.

Chapter 2 focuses on an overview of the C programming language, highlighting its popularity and relevance in both past and present times.

Chapter 3 discusses the most popular Integrated Development Environments which are currently available on Linux systems, highlighting the advantages and disadvantages of each one. It also presents the possible choices of technologies that could be used for implementing our application and the pros and cons of each technology.

Chapter 4 presents the application itself from two distinct perspectives. First, it focuses on the development process, describing the important steps of the software development life cycle and showing the approaches taken at each step in developing the IDE. Second, it showcases the final product by providing a walkthrough over all its features.

Chapter 5, the conclusion, provides an interpretation of the results obtained during this research, while also briefly mentioning future work and extensions that will be brought to the application.

2. Theoretical Aspects

2.1 The C Programming Language

2.1.1 Why C?

"C is quirky, flawed, and an enormous success." - Dennis M. Ritchie [8]

Even Dennis Ritchie, the inventor of C, admits it. But how "flawed" and how popular is C actually?

First, let's look at some examples to address the former issue:

Example 1.

Peter Van Der Linden, in his book, *Expert Programming: Deep C Secrets*, shares his experience with us:

The $20 Million Bug
This was a bug in an asynchronous I/O library, whose functionality was needed by a customer. The bug was holding back a sale of $20 million worth of hardware. Considering this, the development team immediately got to work in order to find it. After intensive debugging sessions, the problem was found. A statement in the code read:
x==2;
This was intended to be an assignment statement (normally "*x=2;*"), but since the programmer accidentally hit the "=" key twice, this statement would now compare x to 2 and simply discard the result. [9] It basically does nothing.

Example 2. [7]

```
int f();
```

Q: How many arguments does this function accept?
A: Any number.

Example 3. Sequence points [7]

```
int a = 41;
a = a++;
```

What's the value of a? 41, 42?
Correct answer: This is undefined.

A variable can only be updated once between sequence points. A sequence point is a program execution moment in which all side effects of previous statements are guaranteed to have been performed and no side effects from subsequent statements have yet to be performed. [7]

Another thing to consider is order of evaluation, which is also unspecified. Consider the following statement *[9]*:

*int x = f() + g() * h();*

The variable x is well-defined, as is the precedence of the operators, but the order of the function evaluations themselves is NOT guaranteed. If, let's say, all three functions would modify a global variable, it is not guaranteed that *h* would be the last to do so (as one would intuitively assume).

Example 4.

We've all been there: One of the first laboratories involving the C language, and we hear the teacher: "**arrays are basically just pointers**". This is only half true. [9]

Let's assume two files:
- file1.c has the following: *int p[] = { 10, 15, 20 }*;
- file2.c has the following: *extern int *p;*

Basically, file2 imports symbol p from file1.

So what happens when, in file2.c, we have:

p[0];

We are basically telling the compiler: go to the location of the pointer variable p, get the bytes from there (which represent an address), add the offset (in this case 0) to that number and then go to that resulting address. This will always result in garbage data, and will most likely **crash the program**.

Why? Array variables always have their value equal to the address in memory. Pointers don't (if they do, it's just a coincidence). What actually happens here is that, when attempting to get the value of p (the pointer), the compiler actually ends up getting the value of the first element of the array p (which represents an address ... or at least the compiler thinks it should, because we declared p as a pointer). If *p* were declared *extern int p[];* in file2.c to match the definition from file1.c, the compiler would behave as one would normally expect.

2.1.2 The Success of C

Even if it's "flawed", C is one of the most successful programming languages, being widely used nowadays and having a history dating all the way back to 1970.

How come?

The success of C is due to many factors. First, and possibly most significant, is that C was developed by programmers for day-to-day use. It was not intended to be a demonstration or a concept language. It provided no constraints and no rigor, instead concentrating on providing power and never getting in your way. This might seem a little harsh for novices and it is. C is better suited for professionals rather that beginners. It does not provide any protective environment, nor does it give feedback on the mistakes you make.

C has the the productivity of a high-level language, but also the power to get the best performance out of the personal computer without having to delve into assembly code.

The extensibility of C was also a reason why it became so popular. By using library functions, the language can be extended in many ways, even beyond what the original designers could have ever thought of. [2]

Despite sometimes being mysterious to the beginner but also the more advanced programmer, C remains simple and small, thus translatable with simple and small compilers. The data types it provides are well grounded into what real machines provide. The language is also sufficiently abstracted from the machine, such that it can be portable. Although portability was not a high-priority concern when C was originally designed, it managed to provide programs, including operating systems, that could run on machines anywhere from personal computers to supercomputers. [8]

2.1.3 Uses of C in Modern times

Some of the uses of C in modern times are:

- Embedded systems
- Operating systems (considering that C was actually developed for writing the Unix operating system)
- Intermediary language for other software. For example, the Bison parser generator (also known as “yacc”) produces parsers in the form of C source code, which can then be compiled.

2.2 Summary

Although old and far from perfect, C was and still is a widely used language. It has gained popularity due to its simplicity, portability and extensibility and also due to the fact that it was built by programmers for programmers. It always provided what one would need to work with, without getting in the way.

3. General Context

3.1 Related Work

With C being such a popular and widespread language, numerous tools have been created to boost the productivity of C programmers. Integrated development environments have come a long way over the years, from the simple (yet complex for its time) Borland Turbo C in 1987, to nowadays' IDEs such as Code::Blocks, NetBeans, KDevelop, QtCreator, CLion etc.

3.1.1 Code::Blocks

Code::Blocks [17] is a free cross-platform IDE built around the idea of plugins. Highly customizable, it offers a lot options for extending it to fit specific needs. It also features good code completion. One major disadvantage of Code::Blocks is that the diagnostic system is somewhat weak compared to other IDEs and on top of that, it signals errors and warnings only when compiling, not also while typing.

3.1.2 NetBeans

NetBeans [18] is an IDE which supports C/C++ and also Java and PHP, offering the possibility to get only the package for the language(s) you need. It is also highly customizable, featuring its own Plugin Portal. The main drawback is that it has much longer load times compared to other IDEs and stuttering also occurs during use.

3.1.3 KDevelop

KDevelop [20] is a free, open source IDE for Linux and Mac, featuring a light-weight user interface. Although KDevelop uses the powerful Clang parser, it does not use it at its full potential, omitting many semantic warnings during code analysis.

3.1.4 QtCreator

QtCreator [21] is a very powerful IDE, with excellent code completion and good source code diagnosis. While it does offer the possibility to create plain C/C++ projects, it is very Qt-centric. It usually comes bundled with the Qt Framework, making it one of the most heavy-weight and largest IDEs when it comes to space occupied. It also has an interface which is somewhat more cumbersome to use than most other IDEs. It does not come with a tabbed editor as standard and it cannot display both code and design windows side by side.

3.1.5 CLion

No discussion about IDEs would be complete without mentioning something from JetBrains, in this case: CLion [19]. One of the most feature-packed IDEs on the market, CLion has, apart from excellent code analysis and completion, the possibility of code generation and an integrated debugger. Its main downside, however, is the fact that there is no free version.

3.2 Potential Technologies for Developing the Application

In order to develop the IDE, we need to look at two types of technologies. First, we need a framework or toolkit to build the graphical user interface (GUI) and second, we need a tool to perform the static analysis in order to discover errors and potentially flawed behavior in the code.

3.2.1 For the Graphical User Interface

Being an application that requires a graphical user interface (GUI), we must look at languages that have toolkits for GUI design. Some of the more popular options for developing desktop graphical user interfaces on Linux systems are Python, Java and C++.

3.2.1.1 Python

For Python we have:

- the TkInter toolkit, but unfortunately, there is no IDE for drag-and-drop UI design, therefore everything must be done programmatically.
- PyQt – integrates well with the Qt framework and is supported by Qt.
    - can be used in conjuction with the QtCreator designer.

3.2.1.2 Java

Swing is the most popular GUI toolkit based on Java. The NetBeans IDE provides a plugin which generates Java code from a drag-and-drop designer.

3.2.1.3 C++

C++ is backed by the huge Qt framework, which offers, aside from GUI development, a lot of other functionalities: data structures, generic algorithms, file I/O, SQL database access, XML and JSON parsing, threading, networking, graphics etc.

3.2.2 For the Static Analysis

The core feature of the IDE, static analysis is used to determine *potential flaws in the code* before it even gets executed. These flaws can range from errors, which will prevent the code from even compiling in the first place, to very obscure and hard-to-detect semantic errors.

Only some semantic errors can be detected and these are usually presented as warnings. They do not restrict the code from compiling, but highlight potentially dangerous or undefined behavior. For example: the programmer failed to initialize a variable before its first use (common mistake, especially with pointers), or is attempting to access an out-of-bounds index on an array of a known size.

Apart from diagnosing the code, static analysis can also be used to provide relevant *code completion* suggestions at any location in the source code.

To be able to perform static analysis, we need a parser, the component of the compiler which performs syntactic (and also semantic) analysis.

A compiler is a software that translates a source program into an equivalent target program. It is split in two parts: analysis and synthesis.

The analysis part takes the source code and creates an intermediate representation of the program. If this part of the compiler detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages. [1]

The synthesis part is the one which takes the information provided by the analysis part and generates the target program.

Figure 3.1 illustrates the typical phases of a compiler.

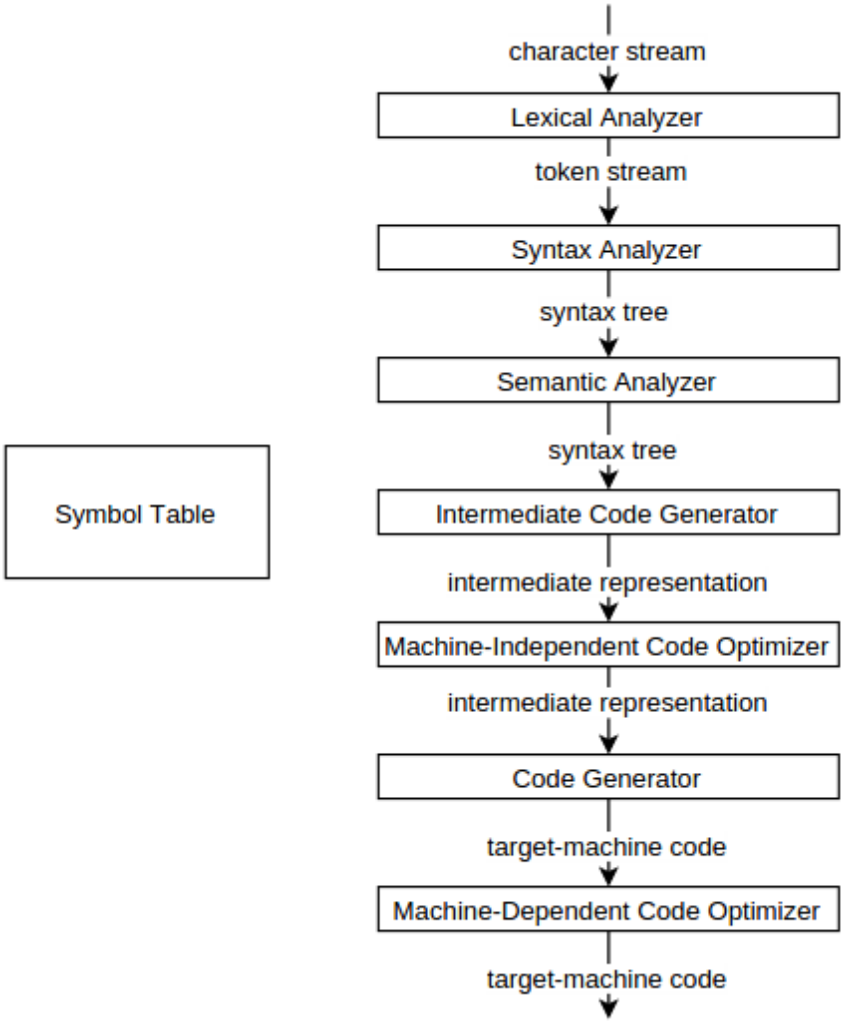

Figure 3.1 Phases of a compiler [1]

We note that the analysis phase starts with the lexical analysis and goes up until semantic analysis, inclusively. The final products of the analysis phase are:

- the Syntax Tree (or Abstract Syntax Tree, AST)
- the Symbol Table

After that, the synthesis phase uses the results obtained from the analysis phase to generate intermediary and object code. Both the intermediary and object code can be optimized. However, these optimizations are optional. [1]

Since we are not interested in generating object code, we only need to consider the analysis part of the compiler for the IDE, especially the semantic analysis part.

3.2.2.1 Bison

Bison is a parser generator. It is also known as YACC – Yet Another Compiler Compiler. It generates the C/C++ source code of a parser, given a grammar specification file. The parser is of type LALR (Look-Ahead Left-Right, meaning it parses the input starting from the left, using right-most derivation and one token of look-ahead.)

Bison is generally used in conjunction with Lex (also known as Flex), which is a lexical analyzer (scanner) generator. Lex generates the C/C++ source code of a finite automaton that can tokenize input (split into meaningful text chunks called tokens) based on regular expressions.

Lex and Yacc allow rapid application prototyping, easy change and easy maintenance. Although mostly used in compilers, applications that look for patterns or have some sort of command input (like a terminal) are also good examples of the applicability of Lex and Yacc. [6]

Grammar files have three sections:

- *declarations* – contains Bison-specific declarations and can also contain C declarations or *#include* statements.

- *rules* – contains the grammar itself, written in a form very similar to Backus-Naur Form (BNF).

e.g.
Backus-Naur Form: *<expr> ::= <term>|<expr><addop><term>*
Bison specification: *expr : term | expr addop term ;*

For each grammar rule, we can specify *semantic actions*, where we can insert C/C++ that will be executed whenever the parser performs reduction by that rule.

- *supporting C code* – this section contains auxiliary code that might be needed. Any code that goes here is placed directly in the generated file, without further processing.

Bison also reports grammar ambiguities and can also solve some of them using simple precedence and associativity rules. It also provides means to recover from errors in order to continue parsing.

Thus, the main advantages of Bison are its simplicity, ease of use and the fact that it can be easily configured and adapted.

However, Bison does have some huge disadvantages over other technologies when it comes to using it for static analysis.

First of all, although it is simple and configurable, it is also low-level and minimal. It comes with nothing out of the box. Thus, the programmer has to take care of everything except the parser itself. This means that, for a fully-fledged language such as C, we would have to design and implement every aspect of the language from the ground up.

C is also a language which is not entirely context-free, thus the same statement can have different meanings depending on the context.

Take for example:

*X * a;*

This could mean either a multiplication of the variables *X* and *a* if *X* is a variable or a declaration of a variable named *a*, which is of type *pointer to X*, if *X* is a type.

Another example:

X(a);

This could mean either a call to function *X* with argument *a*, if *X* is a function, or a declaration of a variable named *a* of type *X*, if *X* is a type.

Therefore, we must take these situations into consideration, too.

Another aspect we need to look at is the fact that the type system in C is hierarchical. As such, we would have to implement a composite, tree-like structure to hold the user-defined types.

Yet another feature of C is the preprocessor, which can be used to include other files (typically headers) in C sources, define macros and even determine which parts of the code are to be compiled.

e.g.

```
#ifdef A
        int a  = 0;
#endif
```

If *A* would not be defined, the code *int a = 0;* would not reach the compilation phase.

Another disadvantage of Bison is that it does not provide an API, making it harder to integrate into other applications.

To summarize, the fine-grained level of customization offered by Bison comes at a price: great complexity. To create a really powerful static analysis tool using Bison is a huge task.

3.2.2.2 Clang

Clang is an open source compiler for C/C++/Objective-C. It is written in modern C++ using STL. [5]

Clang is designed as an API, therefore it is easy to integrate into IDEs for source code analysis, completion, refactoring etc. This is one of its main design goals. [12]

Before going further, we need to outline some basic Clang terminology [10]:

The *translation unit* is the basic compilation unit. It consists of the contents of a source file together with all included headers (either directly or indirectly), minus the code excluded during preprocessing.

The *index* is a data structure that contains multiple translation units that are related and would be linked together. The index can be used for cross-referencing between translation units, such as when a variable is defined in one file and imported in another (via the *extern* storage class specifier). Libclang (detailed below) provides an API dedicated to cross-referencing, both between translation units and within the same translation unit's AST.

A *cursor* refers to a node of the AST, each having a parent and potentially multiple children, the root being the translation unit itself.

The libclang API

Clang being under heavy development, it is possible for the API to change. For an IDE, a more stable API would be desired. This is where libclang comes in.

The goal of libclang is, in one word, *stability*. The library offers an indexer for multiple files, a parser and supports parsing of unsaved documents. [5]

Libclang should be used whenever we want to work at a higher level of abstraction, such as iterating through the Abstract Syntax Tree (AST), without needing to delve into every little detail that the AST can provide. [11]

Some functionalities that libclang provides are:

- traversing the AST using cursors and retrieving information about the cursors
- retrieving source code locations and mapping between cursors and source code

- code diagnosis
- code completion

3.3 The Technologies of Choice

In the previous section, we have gone over all the popular technologies from which we can choose, together with their advantages and disadvantages. However, we can see that some are clearly more fit for purpose than the others.

3.3.1 Base Language: C++

C++, unlike Java and Python, compiles into native code, making it much faster than any language that runs in an interpreter or a virtual machine.

Aside from that, C++ also has all the object-oriented features, such as: encapsulation, abstraction, polymorphism and inheritance. What C++ lacks in comparison to Java are interfaces, but these can be simulated to a certain degree with the use of purely abstract classes (classes having only abstract methods) and multiple inheritance. Thus, in practice, the lack of interfaces does not present a problem. Another thing that C++ lacks is a Garbage Collector. However, with proper use of destructors and smart pointers, we can easily prevent memory leaks while at the same time not having to deal with the performance overhead of a Garbage Collector.

The Standard Template Library (STL) must also be mentioned, since it is a great addition over plain C++, providing easy-to-use, generic (template) data structures and algorithms.

The fact that C++ provides the extensibility, flexibility and maintainability of object-oriented programming, while, at the same time, offering the raw, unmatched performance of natively compiled code, makes it an ideal choice for developing the application.

3.3.2 Framework: Qt

With more than one thousand built-in classes [13], many of which can be subclassed to fit specific needs, saying that the Qt framework offers a lot of functionality can be seen as modest.

Apart from the typical GUI components such as TextEdit, LineEdit, Button, Splitter etc., Qt provides other, diverse functionalities.

e.g.

- generic data structures, generic algorithms, regular expressions, file management, SQL database access, XML and JSON parsing, processes, threading, networking etc.

Qt also provides many features which can be used out of the box and come in handy when developing a new IDE, such as:

- text completer – useful for presenting code completion suggestions while typing,
- syntax highlighter – highlights code based on regular expressions,
- file system watcher - notifies when files or directories change,
- directory model/file system model – which, when coupled with a tree view GUI component, can present the contents of a particular directory and also allow navigation within the file system,
- tab widgets – which, among many other uses, when coupled with text editing widgets, will form the IDEs code editor.

3.3.3 Static Analysis Tool: Clang

While Bison, or any other parser generator for that matter, can be used to generate powerful parsers, optimized for finding the most obscure, potentially dangerous errors, Clang comes with sufficient versatility and power out of the box and also has an API specifically designed to be used within IDEs. [5]

Clang provides clear and expressive error messages, trying to be as user-friendly as possible. This is done through a diagnostics engine whose purpose is to process errors and convert them to user-friendly messages. [5]

Clang is also customizable when it comes to diagnosing the source code, since we can explicitly specify what kind of warnings we want it to look for, thus trading between diagnosis power and parsing speed.

3.4 Summary

To develop the IDE, we must find the right tools for the job. We must look for GUI frameworks and for software that can analyze code or generate a parser to be used for code analysis.

Since no technology is perfect, trade-offs must be made between performance, versatility, ease of use (which in turn leads to better productivity) and the amount of support each technology has, such as proper documentation or tools to assist development using that technology.

After pondering the advantages and disadvantages of each option, we come to the conclusion that the most fit-for-purpose tools are: C++, together with the Qt framework and Clang. All three are very versatile, have good documentation and are accompanied by great development tools. Furthermore, they can be use together seamlessly and offer very high performance, due to native compilation.

4. The Application

4.1 The Development Process

4.1.1 Requirements Specification

The first step when designing new software is to specify the requirements. A requirement is either a feature that a system must have or some constraint that it must satisfy in order for the client to accept it.

Therefore, in the requirements specification phase, we need to find out the purpose of the system and in what conditions or environment it will operate. [3]

Requirements are of two types:

- *functional*: which describe how the system interacts with the environment (be it either external systems, or the user), in short: *what it should do*.

- *nonfunctional*: which focus on the qualities of the system itself and *how it operates*, rather than what it does. Qualities include: usability, performance, maintainability.

Requirements specification is the refined during analysis, by structuring and providing a more formal description.

The requirements specification and the analysis are the same in essence, the only difference being the type of language that is used. The requirements specification is presented in natural language and is more addressed to the client, while the analysis documents are more formal and geared more towards the developers. However, both of these phases of software development focus on providing a point of view from the user's perspective. [3]

4.1.1.1 Identifying the Requirements for the Application

For our application, the requirements in natural language could look something of the form:

“Develop a lightweight Integrated Development Environment for the C language, for students to use in order to do lab-work. The application must have a lightweight interface with easily identifiable components, must provide good code diagnosis and must be easy to use for basic scenarios such as editing files and creating and running small projects for labs like Operating Systems, Computer Networking etc.

The application must be available for use on Linux systems.”

Given this description, we can construct the requirements specification, keeping in mind that we need need to look at the requirements from two perspectives: *what the application does* and *how the application works.*

The requirements specification that we can construct from this description is:

Functional requirements:

1) Maintaining a Project
    1.1) Create Project
    1.2) Open Project
    1.3) Close Project
    1.4) Build Project
    1.5) Clean Project
    1.6) Rebuild Project
2) Running a Project
    2.1) Run Project
    2.2) Debug Project
        2.2.1) Debugger features:
            2.2.1.1) Execute instructions step by step
            2.2.1.2) Set breakpoint
            2.2.1.3) Run to breakpoint
            2.2.1.4) Report requested values in real time
3) Editing Project Files
    3.1) Tabbed Editor
        3.1.1) Each Tab must have:
            3.1.1.1) Syntax Highlighting
            3.1.1.2) Code Completer
            3.1.1.3) Error and warning detection capability
    3.2) Basic file management features:
        3.2.1) Create New File
        3.2.2) Open Existing File
        3.2.3) Save File
        3.2.4) Save File As…

Non-functional requirements:

The application must:

- work on Linux platforms (both 32-bit and 64-bit), thus the application should be 32-bit,
- be fast to start up and be responsive,
- be small enough such that it would be portable.

4.1.1.2 Identifying the Use Cases

In order to formalize the requirements, first we need to identify the use cases and the actors, which correspond to the functional requirements. Use cases express the system's behavior from an external point of view. They describe functionality that yields a visible result for the actor. Actors are entities that interact with the system. They can be users but also other systems. [3]

The Use Case Diagram

We have one actor: the Programmer (if we want to use a more general notion), or Student (if we want to be very specific).

To simplify the diagram, we use two relations between the use cases: [3]

- *include (<<include>>)*: - whenever a use case is triggered, all use cases which it includes will also be triggered. On the diagram below, dotted lines with no annotation represent *include* relations.

- *extend (<<extend>>)*: - a case which extends another case might be triggered by it, but only in some situations.

E.g. on the diagram below (Figure 4.1):

- *include*: when the Programmer wants to trigger the use case Run Project, he will also trigger Build Project (because the project must be up-to-date before the run).

- *extend*: when the Programmer closes a tab, the Save File use case will only be triggered if the file has unsaved changes.

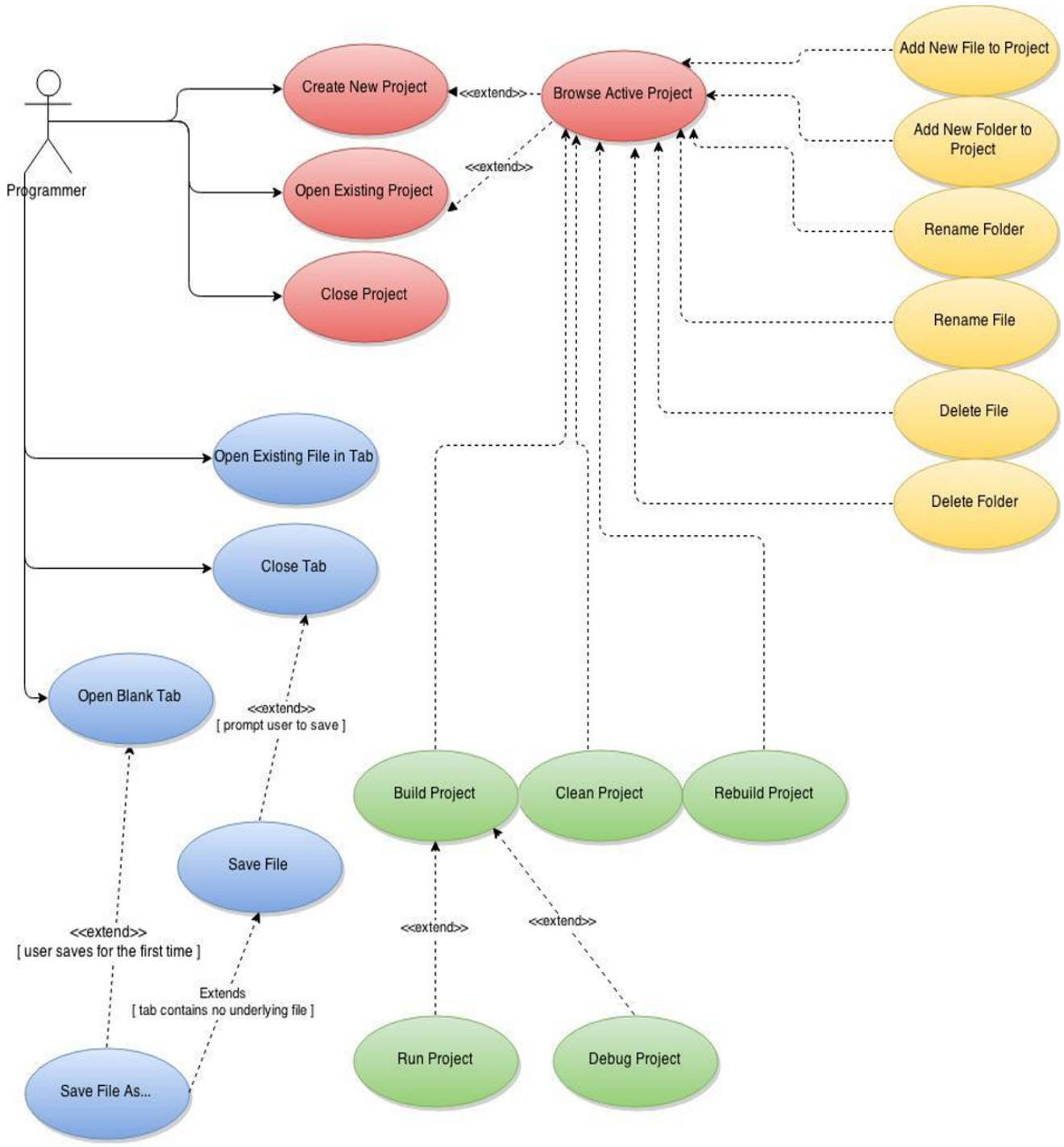

Figure 4.1 Use Case Diagram

4.1.1.3 Describing the Use Cases

Now that we have outlined all the use cases for our IDE, we need to describe each use case. For this, we will use a template similar to the one Bernd Bruegge and Allen Dutoit present in their book, *Object Oriented Software Engineering Using UML, Patterns, and Java (3rd edition)*. [3] The template we will use has the following format:

- ID – a unique identifier for the use case
- name – the title of the use case (which appears on the diagram)
- actors – actors involved in the use case
- description
- preconditions (entry conditions)
- postconditions (exit conditions)
- normal flow
- alternate flow
- exceptional flow

Let's take a simple example: the *Open Existing Project* use case, shown in Table 1, below.

ID	2
Name	**Open Existing Project**
Actors	Programmer
Description	Opens an existing Project.
Precond	Actor must have permissions for the path where .proj file is located.
Postcond	The folder in which the .proj file is located is assumed to be the project folder. Subfolders “src”, "obj" and “bin” are created if they do not exist.
Normal Flow	1. Actor performs *Open Project* action. 2. Actor is presented a dialog containing a text input and a browse button for him to choose the project file. 3. If project file is valid, information about the project is displayed. 4. Information can be changed before opening the project. (compiler, make utility, debugger, extra compiler and linker flags) 5. Actor clicks OK button. 6. Project's "src" folder is displayed in the Project Explorer. 7. Project's name is set above the Project Explorer. 8. If project information was changed, it is saved to the .proj file.
Alt Flow	1. If Actor clicks Cancel button, dialog is closed and nothing changes.
Exceptions	1. If project file is invalid (extension is not .proj), or Actor does not have permission, Actor is informed and no further action is taken.

Table 1. Use Case Description

4.1.2 Analysis

Analysis is the phase of the software development life-cycle that refines and formalizes the requirements specification in an attempt to ensure correctness, completeness, consistency and lack of ambiguity. [3]

To formalize the behavior and visualize the interaction between objects for each use case, we use interaction diagrams.

Sequence diagrams are a type of interaction diagrams that present the objects and the interactions between them over time. The interactions are represented horizontally, while the timeline is represented vertically.

The importance of the sequence diagrams lies in the fact that they create a bridge between use cases and objects by showing the distribution of behavior among the objects for each use case. They also allow visualization of the lifespan of each object. [3]

Figure 4.2 presents an example of a UML sequence diagram corresponding to the *Open Existing Project* use case.

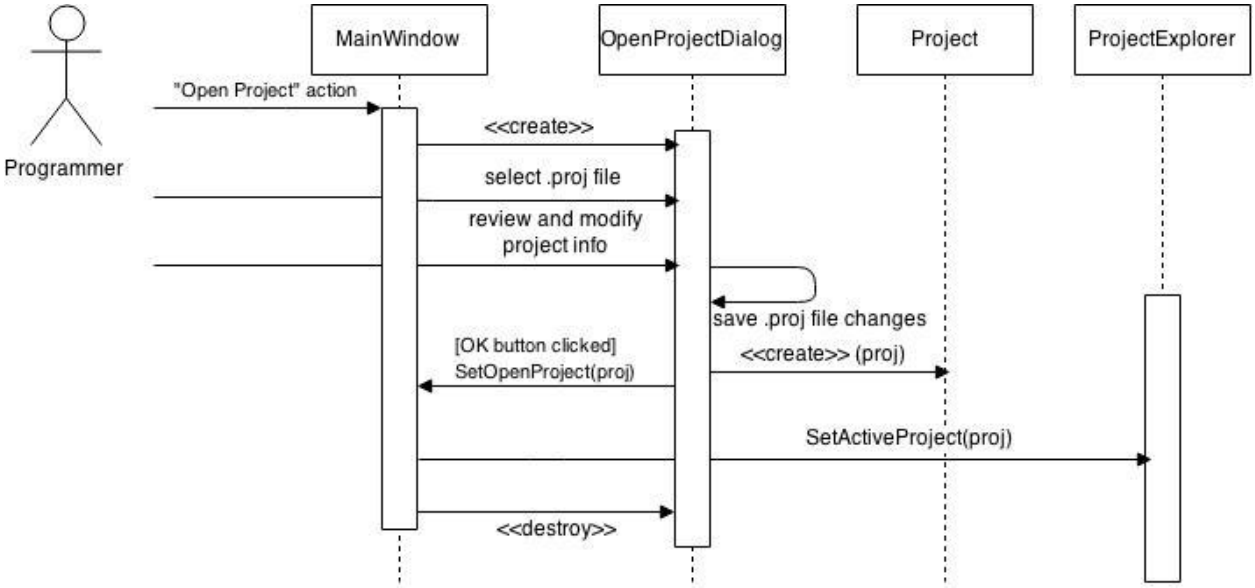

Figure 4.2 Sequence Diagram for the *Open Existing Project* Use Case

We can see that the diagram explicitly depicts the exact lifetimes of objects, by using interactions such as <<create>> and <<destroy>>.

4.1.3 Design

During the design phase, we describe the structure of the system. To do this, we create class diagrams, which illustrate the system from a structural point of view, in terms of classes, attributes, operations and the relations between them. [3]

The representation of a class in UML consists of a box having three parts stacked each on top of another. The first (top) part states the name of the class, the middle part displays its attributes and the lower part displays the methods (also called operations). Conventionally, the

first letter of any class name is uppercase. [3] This naming style is commonly referred to as PascalCase or UpperCamelCase.

4.1.3.1 Design Patterns

A design pattern can be thought of as a template which has been refined over time to solve recurring problems.

Design patterns are partial solutions to common problems, e.g. separating an interface from its different implementations or adapting to a legacy system.

While the number of classes included in a pattern is small, through delegation and inheritance, they provide an extensible and maintainable solution. [3]

A design pattern is composed of four parts:

- name
- problem description
- solution
- set of consequences

- name – Naming patterns allows us to refer more easily to them. It also lets the designers think at a higher level of abstraction and allows having a common, well-defined vocabulary.
- problem description – states what are the conditions that must be fulfilled in order to consider applying the pattern.
- solution – describes the actual structure of the pattern: what classes are there, what is each one's responsibility and how it interacts with the others.
- set of consequences – these represent the impact the pattern has on the design, both in terms of result and in terms of trade-offs. Impacts may occur on different characteristics of the system: flexibility, extensibility, portability etc. [4]

In the following sections, we will go through various components of the IDE and illustrate what design patterns were applied and compare them to other candidate patterns in terms of the complexity they add and the extensibility and flexibility they provide.

4.1.3.2 The Editor

The Editor's basic function is to display and allow modification of the contents of a file. However, not all files are the same. A page associated with a basic, plain text file will have different requirements from a C source or header file (.c and .h, respectively).

How do we decide which page type we need? A simple (and sufficient) guess would be to use the file extension. Therefore, we can say that a file extension uniquely identifies a file type, thus an editor page type.

Enter the ***Factory*** design pattern:

The Factory pattern is a creational design pattern (as the name implies) whose intent is

to create objects without the client needing to know anything about the creation process. The objects that the factory creates are accessed through a common interface or superclass. [14]

Which implementation (or subclass) the factory creates is usually determined by an ID or a key, passed as a parameter to the *Create* function.

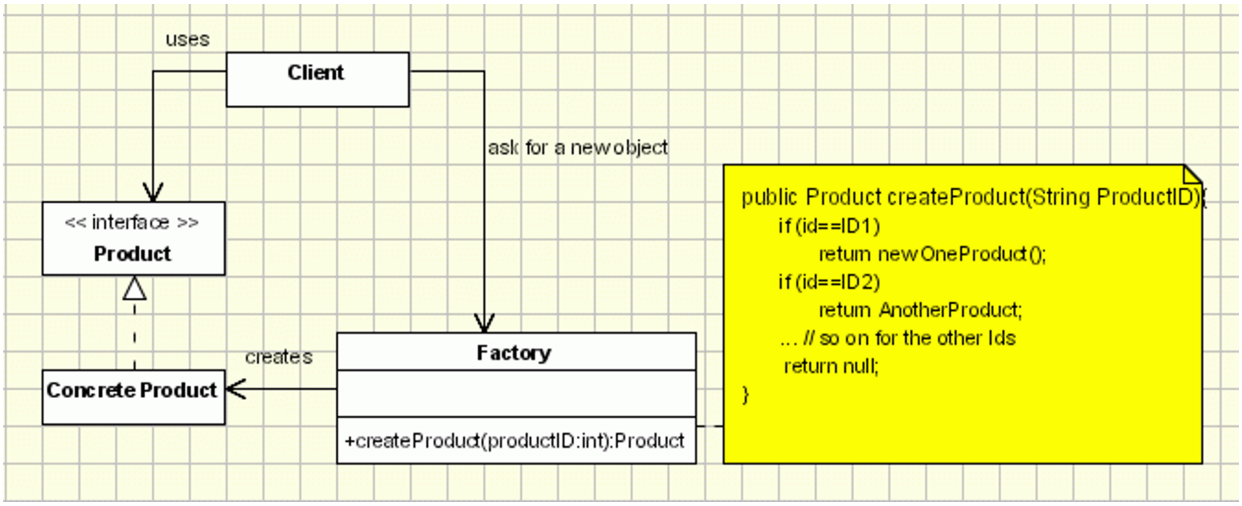

Figure 4.3 Simple example of the *Factory* design pattern [14]

Figure 4.3 shows a simple example of the Factory pattern class layout, together with a very basic implementation of the Create method.

However, this (usually first approach) of designing the Factory itself leads to a problem: while we have indeed isolated the construction logic of each object type and we can add new types without affecting the others, the Create method of the Factory still needs to be updated each time we add or remove something.

How do we get around this extensibility problem?

Alternative 1 – <u>Reflection</u>:

Reflection is the ability of a language to inspect and modify the behavior of the application at runtime. It can be used to retrieve information about the class itself, its methods, fields, constructors etc. [15]

Knowing this, the Factory can be given an extra method, which would associate a file extension with a class (for example, by using a map data structure, where the value would represent a reference to the class information). The Create method would use the key to get the class information and, through reflection, gain access to its constructor and call it, creating a new object of that class. [14]

Disadvantages of this alternative:

- First, the obvious one: not all languages support reflection. C++, for example, doesn't.

- Second, reflection has a performance overhead. This is a minor disadvantage for our particular case, since creating an editor page is not a feature that requires high performance. - Third, reflection can also break encapsulation, since it can access any field or method and use it. [15]

Alternative 2 – The *Factory Method* pattern:

The *Factory Method* pattern (also known as *Virtual Constructor*) allows us to define an interface which will be used to create objects, but only subclasses decide which class to instantiate. Therefore, the **creation of objects is deferred to subclasses**. [4] We will call these classes *Creators*.

Although it adds more complexity to the application, the *Factory Method* pattern can be **used in conjunction with the Factory** to allow new types to be added without modifying the Factory itself. The Factory will defer the creation of the object to a specific Creator, which is associated to a key. We don't need to modify anything when we decide to add a new page type to our application. We just need to implement/extend two interfaces/classes. Everything else stays the same.

Therefore, if we don't want to use reflection to access the constructors directly, or if the language doesn't support reflection at all, we place an extra layer of abstraction over the constructors of the classes (hence the name *Virtual Constructor*).

The client only needs to know about the Editor, nothing else.

Figure 4.4 illustrates the design of the Editor which uses the Factory in conjunction with the Virtual Constructors (Creators). The *EditorPageFactory* maps the file extensions to their respective page Creators.

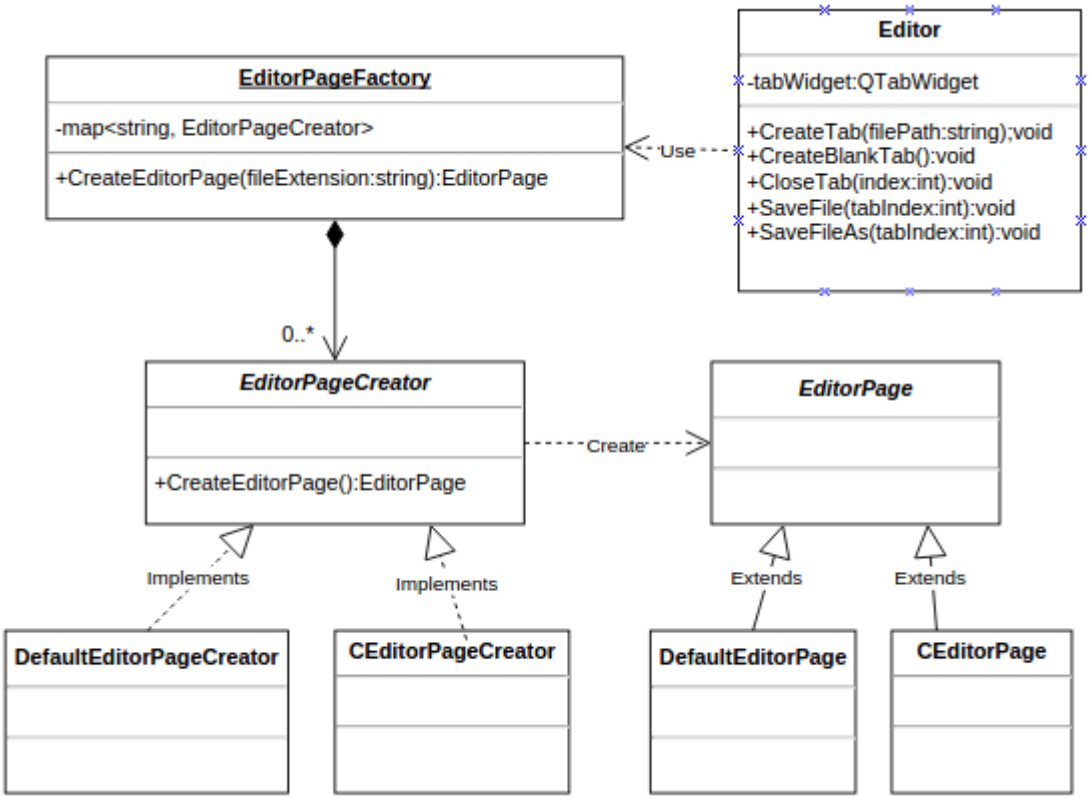

Figure 4.4 The design of the Editor

4.1.3.3 The Project Explorer

The Project Explorer's architecture is Model-View-Controller (MVC). MVC is an architectural pattern which decouples an application into three parts, each with its own responsibility: [16]

- the Model stores data.
- the View presents information.
- the Controller manipulates data.

Two of these three components can be used out of the box from the Qt Framework.

- The QDirModel is used to hold information about the files and folders of the project.
- The QTreeView is used to display the project structure.

The Project Explorer is used to set, change or close the active project and perform administrative tasks such as adding, renaming or deleting files and folders. It has a reference to the project tree and also to the project model. Therefore, the Project Explorer acts like a Controller.

The file operations are trivial and are already partially implemented; we only need to implement stricter permission checks so we can inform the user in case of permission issues.

That being said, we will focus on how the project itself is managed.

1. How do we map the project file to a Project object?
We use a conversion algorithm. From Project to file and vice-versa.

2. What if we want, in the future, to change the format of the project file without affecting existing code?

The *Strategy* design pattern

The *Strategy* pattern is a behavioral design pattern whose intent is to define a family of algorithms, encapsulate each one, and make them interchangeable. This allows independence of the algorithm from the client. [4] In other words, the Project Explorer doesn't need to know about how the project is stored and retrieved.

Knowing this and taking a look at the answer to the previous question, it makes sense to apply the Strategy pattern to the conversion algorithm.

First, we define the common interface: the *ProjectFileConverter*, having the two methods which allow conversion back and forth between the file and the Project object.

Second, we implement this interface to allow storage and retrieval of the project from a JSON file.

Whenever we want to store the project in the different format, we just reimplement the *ProjectFileConverter* interface to suit our needs, for example: XML files or binary files.

Figure 4.5 shows the diagram expressing this design. Note: for simplicity, the Model and View classes have been omitted, they are only mentioned as references in the *ProjectExplorer* class.

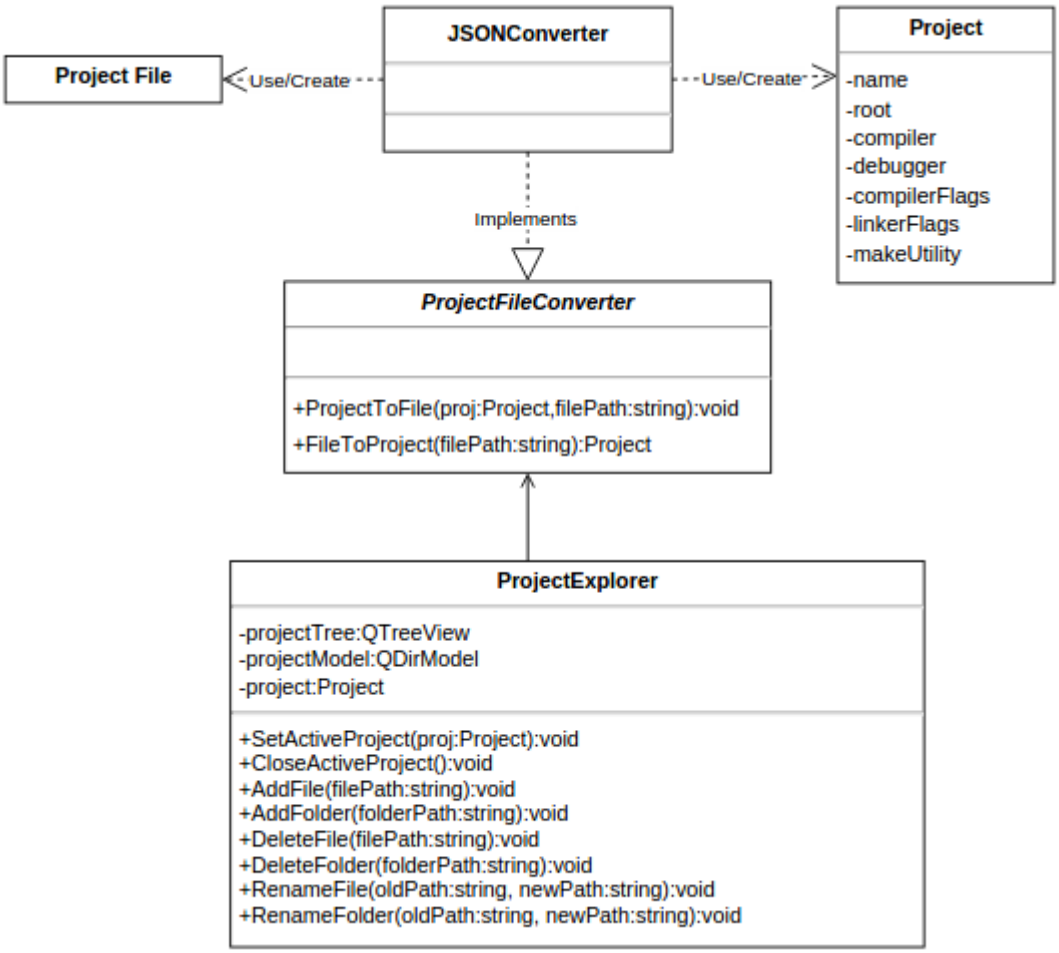

Figure 4.5 The design of the Project Explorer

4.1.3.4 The Project Builder and the Project Runner

The Project Builder is used to build, clean and rebuild the project. The Project Runner is used to start the project executable in a terminal and report the exit status when the process finishes. Both provide updates of their activity.

There are many approaches to building a project, one of the simplest being with the use of makefiles and the GNU *make* tool [23].

A makefile is a file which describes dependencies among files and actions to be taken whenever there is a change in a file on which another depends. A very simple makefile example:

```
all: main
main : main.o module1.o module2.o
        gcc -o main main.o module1.o module2.o
module1.o : module1.c
        gcc -o module1.o -c module1.c
```

```
module2.o : module2.c
        gcc -o module2.o -c module2.c
main.o : main.c
gcc -o main.o main.c
```

The file illustrates dependencies of object files (.o) on source files (.c) and the dependency of the final executable on all object files. Whenever a source file changes, it is recompiled (using *gcc* in this case) to produce the equivalent object file, which in turn changes, so the executable is linked again, too.

We will need an algorithm which generates the makefile and then delegates the build process to *make*.

As discussed in the previous section, whenever we are dealing with a family of algorithms that must be interchangeable, it makes sense to apply the Strategy pattern:

- define an interface: *ProjectBuilder*, having methods to build, clean and rebuild the project

- implement it, in this case: *MakefileBasedProjectBuilder*.

To provide updates about the processes, we will use an interface called *OutputWriter*, which will be used by both the project runner and the builder.

Again, applying the Strategy pattern, the *OutputWriter* will be implemented by a *ListOutputWriter*, which displays the output of the processes in the output window of our application.

If we ever wanted to process the output in a different way, for example to store it in log files, we simply need to reimplement the OutputWriter interface.

To add additional behavior to an existing implementation of OutputWriter, we could use the *Decorator* design pattern (also known as Wrapper), which allows dynamic extension of functionality. [4]

Figure 4.6 shows a diagram of the design we have discussed in this subsection.

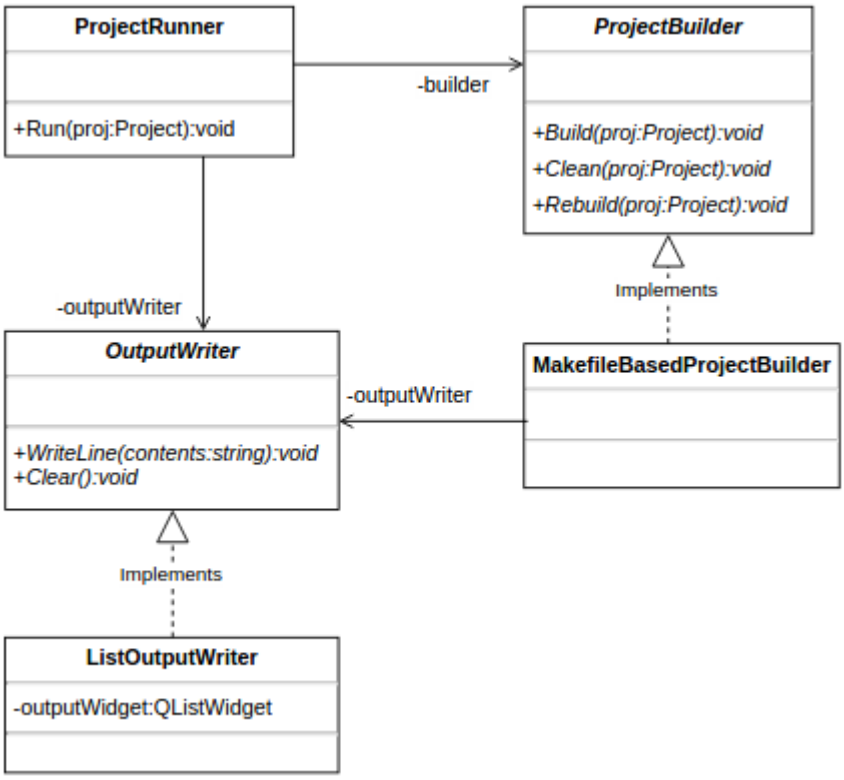

Figure 4.6 The design of the Project Builder and the Project Runner

4.1.4 Implementation

Now that we have the class diagram, we can translate it into code. Since we are working with Qt, it makes sense to use the highly optimized QtCreator IDE for implementing the application. The advantage of QtCreator over other IDEs is that it reduces the project management overhead by automatically generating boilerplate code. This is especially useful when dealing with GUI classes.

The GUI design is also facilitated by the visual drag-and-drop designer that QtCreator offers, which is very intuitive and feature-packed. It allows us to:

- easily add widgets, set their properties and group them into layouts,
- create actions, assign them keyboard shortcuts and map them to handlers,
- add submenus and menu items to the already existing menubar (if we subclass *QMainWindow*),
- morph widgets: for example we can transform a *QTreeView* in a *QTableView*.

Another thing we have to consider is a version control system, in order to keep track of changes and to have a backup of the code. Git is a good choice because it is free and cloud-based. Also, QtCreator offers a graphical interface for managing the Git repository, but the Git console application can also be deemed sufficient.

4.2 The Final Product

4.2.1 Getting Started

When you first start the application, you will notice that the main window is split into four components, as seen in figure 4.7.

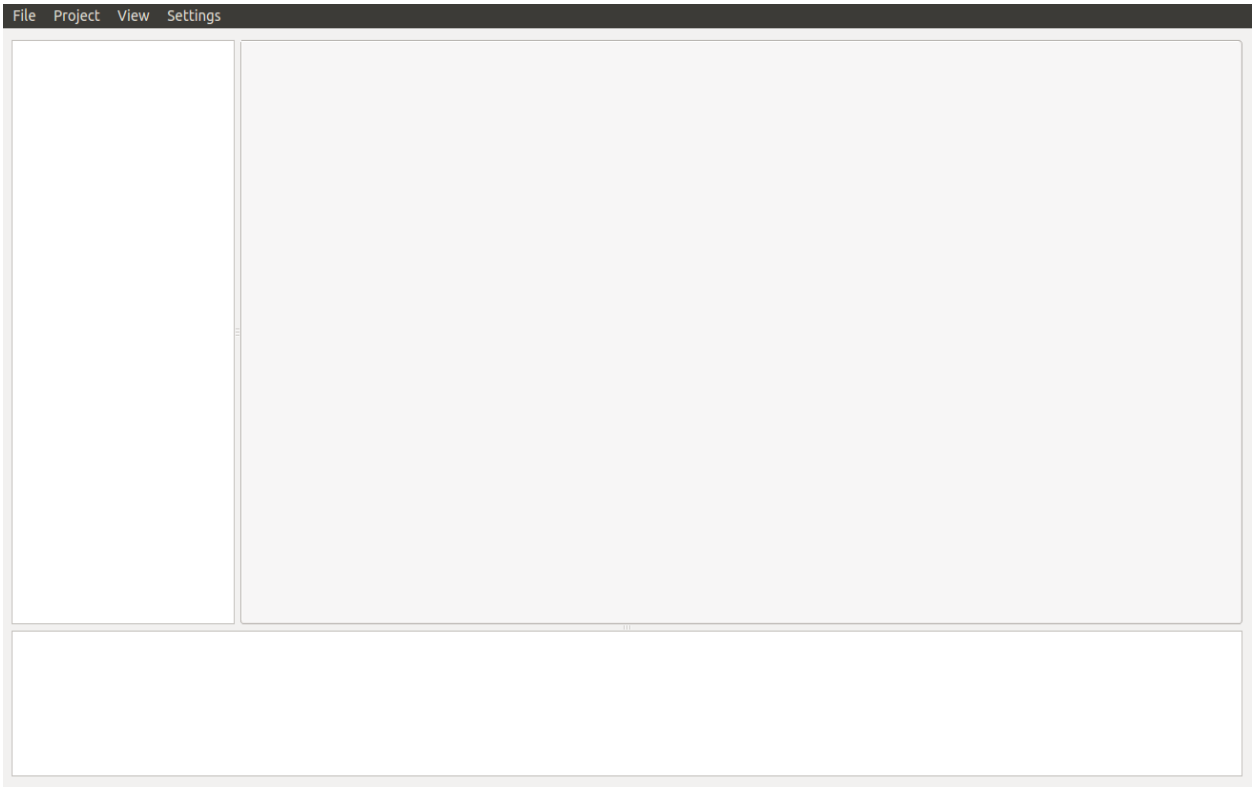

Figure 4.7 Main screen of the application

Menu - at the top of the screen
Output Window - at the bottom of the screen
Project Explorer - to the left
Editor - center-right, between the other components

4.2.2 Menu

The menu bar contains four submenus, as seen in figure 4.8.

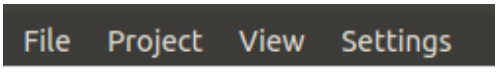

Figure 4.8 The menu bar

4.2.2.1 File Menu

The file menu allows basic file operations and also allows exiting the application.

- **New File (ctrl-N)** – Opens a blank tab in the text editor, having a default name. Note that the asterisk (*) tells you that the file has unsaved changes or is not on the disk at all. (Figure 4.9)

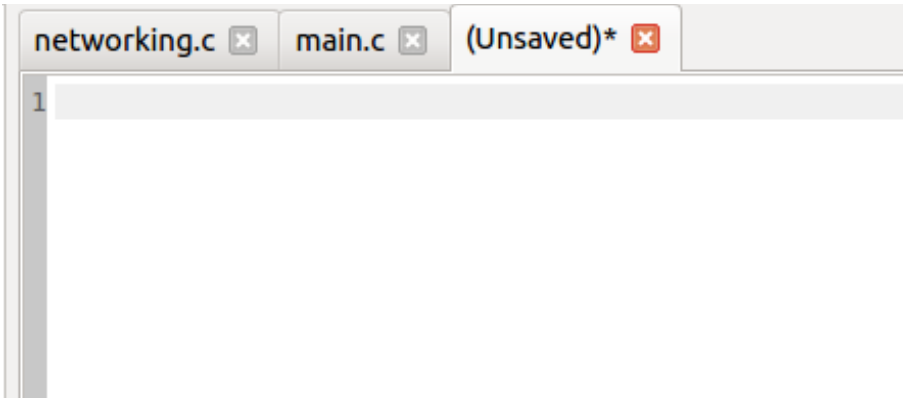

Figure 4.9 A tab with an unsaved file

- **Open File (ctrl-O)** – Allows you to select a file or more to open (Figure 4.10). Each file you open will be placed in a separate tab in the editor (Figure 4.11).

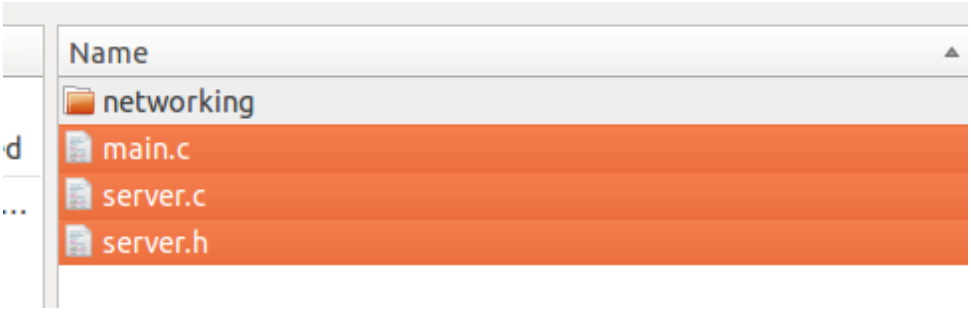

Figure 4.10 Selecting multiple files to open

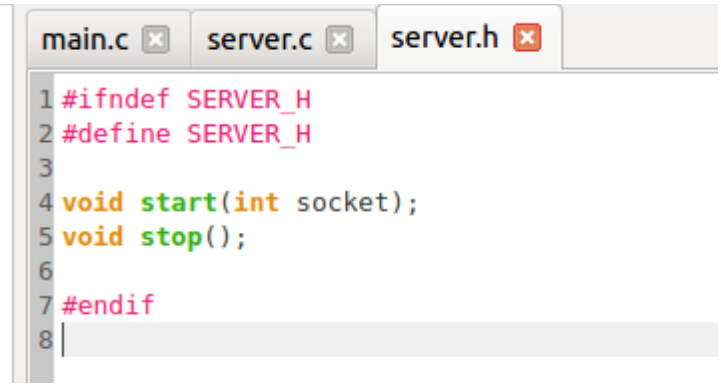

Figure 4.11 Selected files opened in the editor

- **Save File (ctrl-S)** – Saves the current file to disk. If the file has never been saved, you will be asked to select a location and give it a name (similar to Save File As …)
- **Save File As …** – Allows you to save the current file under a different name.

4.2.2.2 Project Menu

- **Create New Project** – Brings up a form allowing you to create a new project, illustrated in Figure 4.12.
  Requirements:
  - the fields Project Name and Containing Folder are mandatory
  - the Project Name must not contain any spaces or slashes.
  - the Containing Folder path must be absolute (must start from root)

- you can also browse for the Containing Folder, Compiler, Debugger and Make using the browse button of each field
- if you start typing an absolute path, you will be assisted with suggestions.

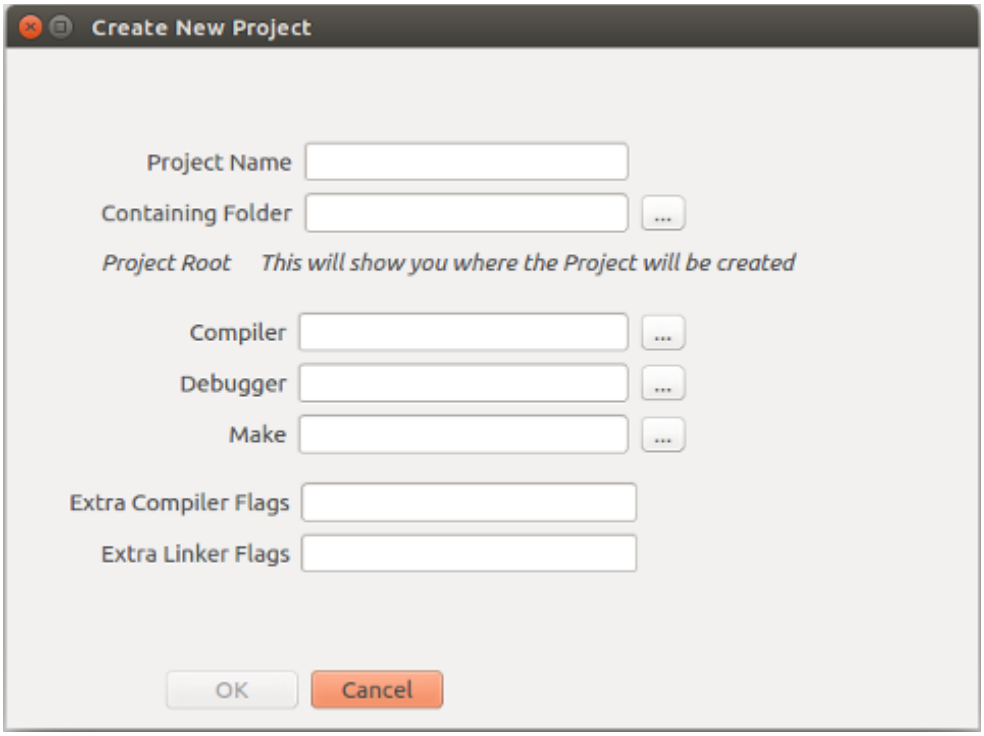

Figure 4.12 A blank project creation form

Extra Compiler Flags – these are additional flags that will be used by the compiler when compiling each file (-Wall, meaning “all warnings”, is included by default, so you don't need to specify it here)

Extra Linker Flags – these are the flags that will be used by the linker in the final step

of the build, when it links all object files into the executable. Usually, this is where you would place any extra libraries to link with, e.g. *-lpthread* or *-lm* (POSIX threads or math).

Figure 4.13 shows an example of a filled project creation form.

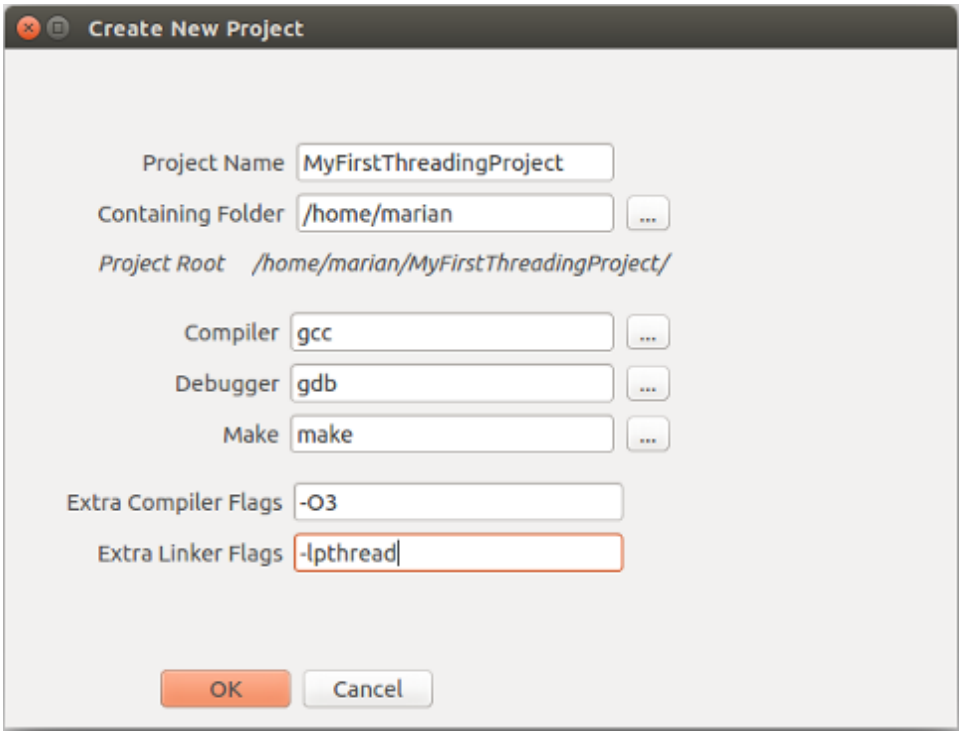

Figure 4.13 A filled project creation form

After you press the OK button, you will notice the Project Explorer displays the name of your new project in the upper part. On the disk, a folder will be created for the project (Figure 4.14), containing:

- *.proj* file – contains the project information you specified
- *src* folder – holds all source files of the project
- *obj* folder – holds all the object files of the project
- *bin* folder – holds the final executable

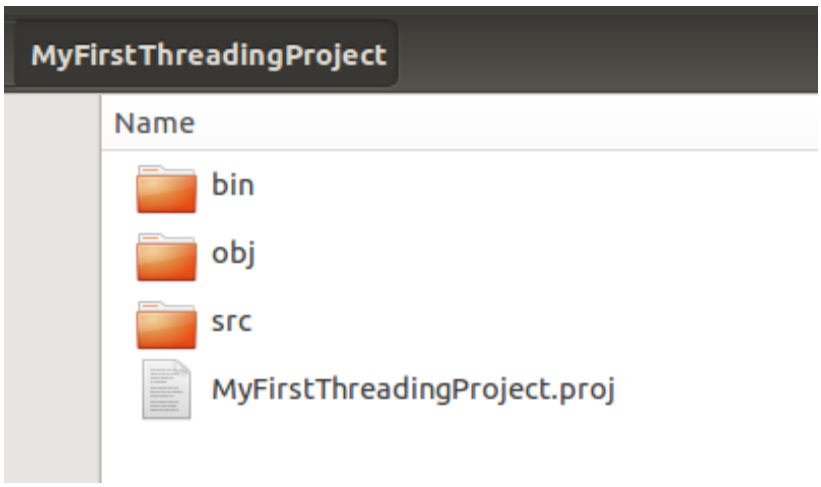

Figure 4.14 The structure of the project folder

- **Open Project** – Allows you to open an existing project.

- When you click Open Project, you are required to select the *.proj* file corresponding to the project you want to open.

- After you confirm the selection, you will be presented with a form similar to the one you use when creating a project. Modify the project's settings if you need to and then click OK.

- The active project (if any) will be closed and the project you selected will be displayed in the project explorer.

- **Close Project** – Closes the active project.

- **Build Project (F7)** – Builds the Project.

- first, a makefile is generated, having the name *YourProjectName.makefile*, and is placed in the project's root directory (the one containing the *.proj* file)

- this makefile is then processed by the *make* tool you specified in the project's settings.

- the object files are placed in the *obj* directory

- the final executable is placed in the *bin* directory

- the output of the build process is displayed in the Output Window. Check this to see if all went well, or if there were some problems that need to be addressed.

Figures 4.15 and 4.16 show a sample project and its build output, respectively.

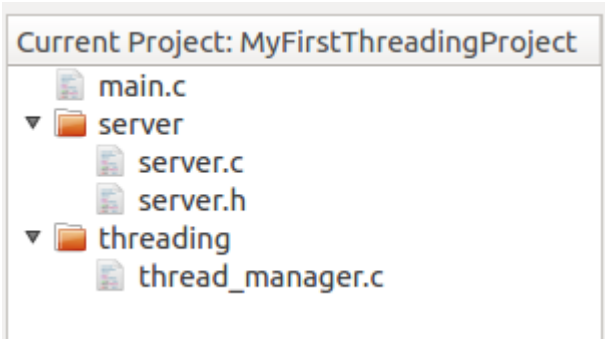

Figure 4.15 Sample project to be built

```
gcc -Wall  -c src/Threading/thread_manager.c -o obj/Threading/thread_manager.o -O3
gcc -Wall  -c src/Server/server.c -o obj/Server/server.o -O3
gcc -Wall  -c src/main.c -o obj/main.o -O3
gcc -o bin/MyFirstThreadingProject obj/Threading/thread_manager.o obj/Server/server.o obj/main.o -lpthread
```

Figure 4.16 Sample build output

- **Clean Project** – Cleans the project, deleting all object files (with the *.o* extension) from the obj directory and the executable from the bin directory.
  - displays output in Output Window. (Figure 4.17)

```
find obj/ -name '*.o' -type f -delete
rm bin/MyFirstThreadingProject
```

Figure 4.17 Sample clean output

- **Rebuild Project (F8)** – Cleans and then builds the project again. (Basically a combination of Clean and Build)
  - display output in Output Window (Figure 4.18)

```
find obj/ -name '*.o' -type f -delete
rm bin/MyFirstThreadingProject
gcc -Wall  -c src/Threading/thread_manager.c -o obj/Threading/thread_manager.o -O3
gcc -Wall  -c src/Server/server.c -o obj/Server/server.o -O3
gcc -Wall  -c src/main.c -o obj/main.o -O3
gcc -o bin/MyFirstThreadingProject obj/Threading/thread_manager.o obj/Server/server.o obj/main.o -lpthread
```

Figure 4.18 Sample rebuild output

- **Run Project (F6)** – Builds the project to ensure it is up-to-date and then executes it in the terminal. (Figure 4.19)
  - displays output in Output Window

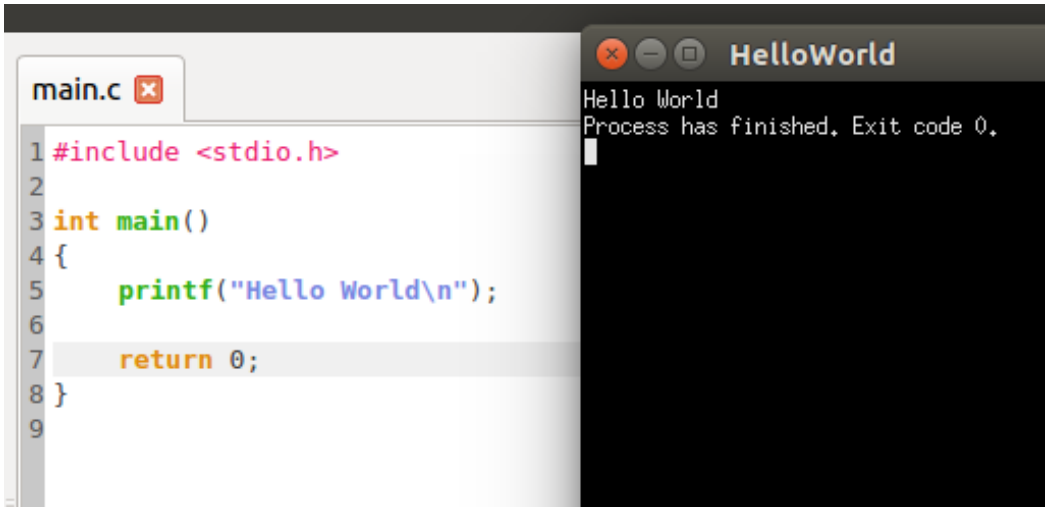

Figure 4.19 Example of a project running

- **NOTE**: if the build fails, the project will not start. (Figures 4.20 and 4.21)

Figure 4.20 An error in the code

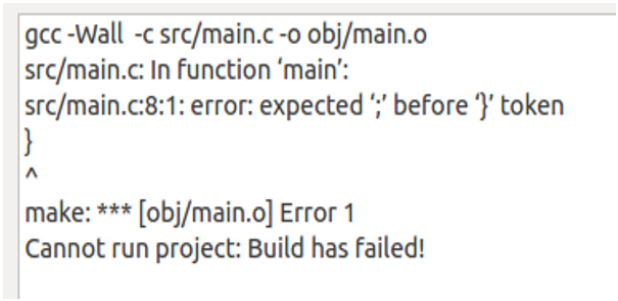

Figure 4.21 Build failing because of a syntax error

4.2.2.3 View Menu

The View Menu is useful if you want to hide the project explorer and the output window in order to get more space for the text editor. (Figure 4.22)

Shortcuts:

- toggle the project explorer's visibility: Ctrl-1
- toggle the output window's visibility: Ctrl-2

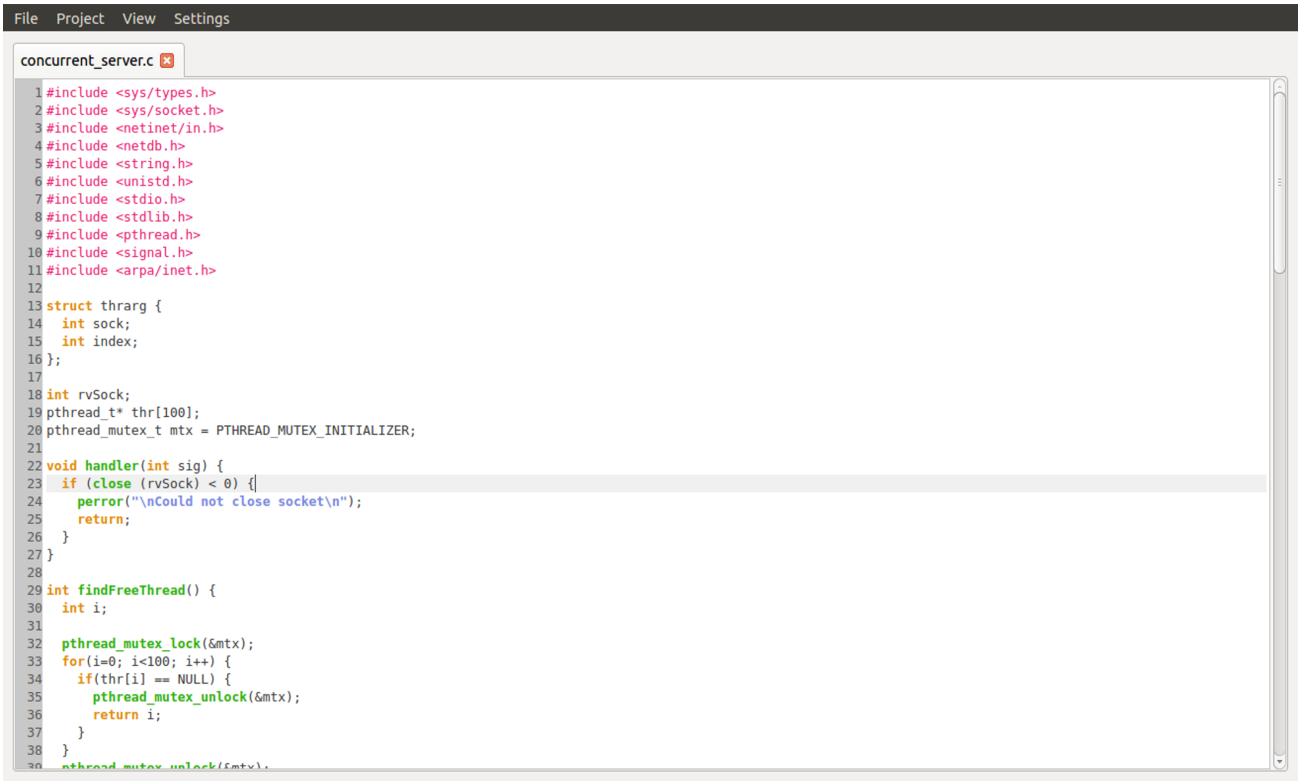

Figure 4.22 The Editor taking up all available space

4.2.2.4 Settings Menu

The Settings Menu allows you to configure the tools and settings used by the projects.

The Project tab allows you to modify these parameters for the active project. If there is no active project, this tab has blank, non-editable fields. (Figure 4.23)

The Default tab allows you to set default project tools and settings. These are suggested each time you create a new project. (Figure 4.24)

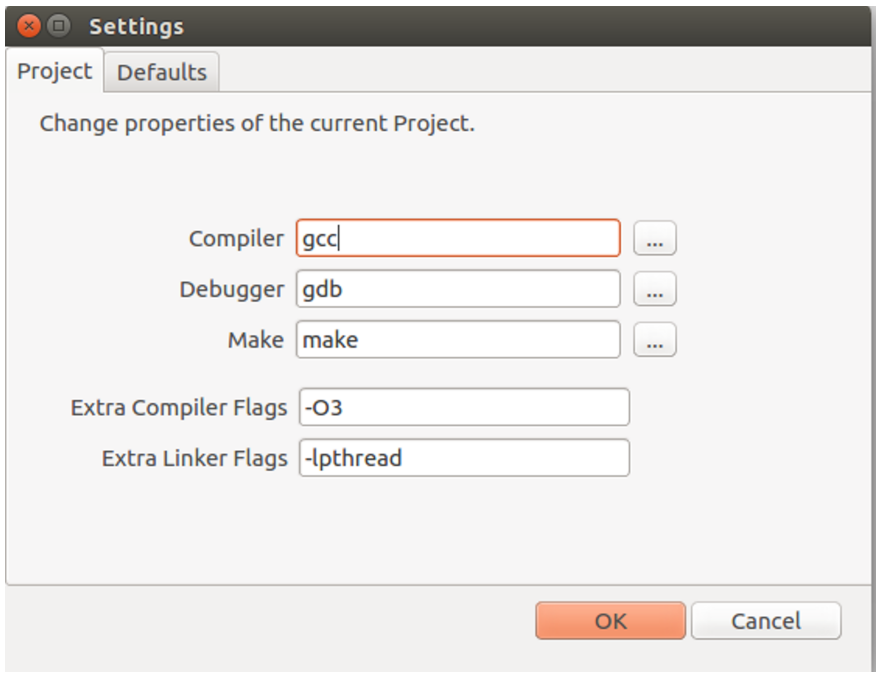

Figure 4.23 Project settings tab

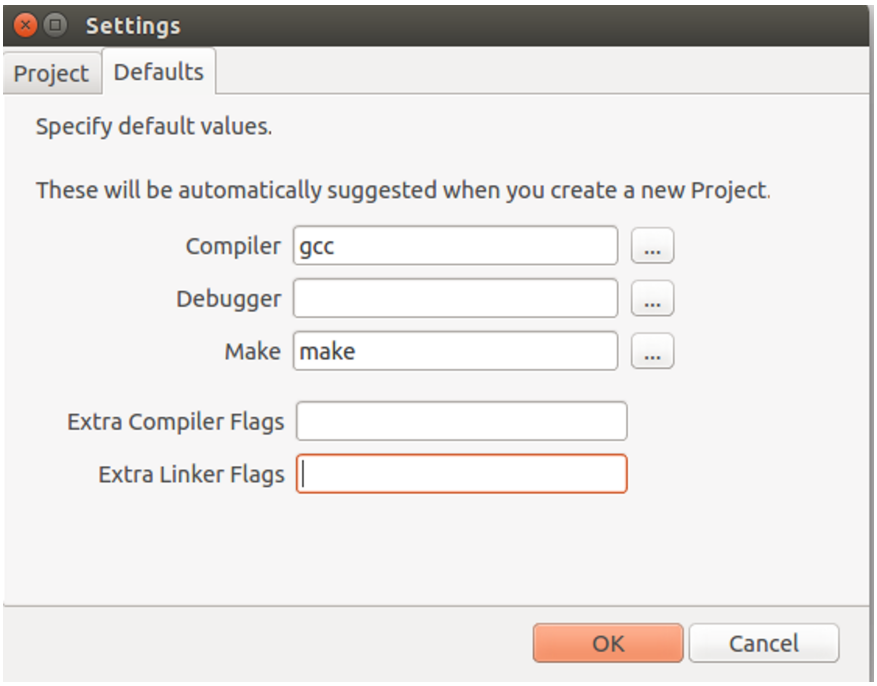

Figure 4.24 Default settings tab

4.2.3 Project Explorer

The Project Explorer, located on the left side of the screen, displays the current structure of your project.

Managing the contents of your project

- to create a new file or folder, right-click on the Project Explorer, anywhere on the empty space, and select either New File or New Folder.
- if you click on an existing file, you will be given the options to rename or delete it.
    - renaming can also be done by simply selecting the file and begin typing
    - **deletion is permanent**, but you will be asked to confirm it
- if you click on a folder, you will be given the options to rename it, delete it or add a new file or subfolder
    - note that deleting a folder will **delete it and everything inside permanently,** but you will be asked to confirm this operation.
- if you want to add many files at once, you can simply move them in the project's *src* folder on the disk and then use the *Refresh Project Explorer* option (shortcut: F5). The explorer will update to reflect the changes.

4.2.4 Output Window

The output window, located at the bottom of the screen, displays the output of various processes, such as building, cleaning or running a project.

4.2.5 Editor

The editor is where you write the code. The features you get in an editor page depend on what type of file is open in that tab.

For example, for a file with no extension, there will be a simple plain text editing area, while for a .c file, the text will be accompanied by syntax highlighting, a code completer to assist with code suggestions and a parser to help you discover potential flaws in your code. To see what's wrong in a particular location in your code, place the mouse cursor or the text cursor over the area which the parser highlighted.

Figure 4.25 shows the parser in action, highlighting warnings and errors found in the code.

```
1  #include <stdio.h>
2
3  int f() {
4      myStruct a;
5      int b = 0;
6      int c = 0;
7      if (b = c) ;
8      return 400000000000;
9  }
10 struct myStruct { int field1; int field2; field3; };
11 char* buildMessage() {
12     return "Hello" " " "World";
13 }
14 void sayHello(char* message) {
15     static int i = 1;
16     i = i++;
17     printf("%s for the %dth time\n", message, i);
18 }
```

Figure 4.25 Diagnostics in code

4.3 Summary

In order to build the final product, we have started from a list of informal requirements. During the analysis phase, we have refined these requirements to better understand them and to ensure they are correct, complete and consistent. In the design phase, we have engineered the structure of the application in a formal manner, by means of a class diagram.

We must always keep in mind that we must create a flexible and extensible design by deciding which design pattern is best for a given situation and also by pondering both the benefits and the complexity overhead that each pattern adds to our design.

5. Conclusion

From the research presented in this thesis, we can conclude that, while good for professional projects, most IDEs provide too many features for novices. We can say they are overly complicated. With this in mind, we have come up with a concept of a minimal but powerful IDE that students which are learning C programming would enjoy using.

Following the steps of the software development process, we have managed to create a product that suits the original concept. The IDE provides basic project management facilities and good code assistance, while also being portable.

In software development, requirements are never final, thus neither is the design nor the implementation. No matter how many features we implement, we must always be open to *change.* That is why the development process of this application was approached with a mindset focused on extensibility as a primary goal.

In the future, the IDE will be extended with the addition of the project debugging feature and it will be optimized with a vast array of convenience features such as code folding and side-by-side file editing. Last but not least, we can think about a plugin system, which will allow the creation and integration of various components into the core application.

Whichever features are added or changed and whichever future directions the application takes, we should always keep in mind the principles from which we started: light-weight, portable and tailored to novices.

Bibliography

[1] Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D.: *Compilers, Principles, Techniques, and Tools (2nd Edition)*, Addison-Wesley Pub., Boston, 2007

[2] Banahan, M., Brady, D., Doran, M.: *The C Book*, Addison Wesley, 1991

[3] Bruegge, B., Dutoit, A. H.: *Object-Oriented Software Engineering Using UML, Patterns, and Java (3rd Edition)*, Prentice Hall, 2009

[4] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley, USA, 1994

[5] Guntli C.: *Architecture of clang*, http://wiki.ifs.hsr.ch/SemProgAnTr/files/Clang_Architecture_-_ChristopherGuntli.pdf, 2011, 1-2 8-9

[6] Levine, J. R., Mason, T., Brown, D.: *lex & yacc*, O'Reilly Media, Sebastopol, CA, 1992

[7] Maudal O., Jagger J.: *Deep C (and C++)*, http://www.slideshare.net/olvemaudal/deep-c, 2011, 26 157 195

[8] Ritchie D. M.: *The Development of the C Language*, http://heim.ifi.uio.no/inf2270/programmer/historien-om-C.pdf, 13-14

[9] Van Der Linden, P.: *Expert C Programming: Deep C Secrets*, Prentice Hall, Englewood Cliffs, 1994

[10] Wilson M.: *An Introduction to Clang Part 2*, http://www.ics.com/blog/introduction-clang-part-2

[11] ***, Clang 3.7 documentation, *Choosing the Right Interface for Your Aplication*, http://clang.llvm.org/docs/Tooling.html

[12] ***, *Clang vs Other Open Source Compilers*, http://clang.llvm.org/comparison.html

[13] ***, Qt Documentation, *All Classes*, http://doc.qt.io/qt-5/classes.html

[14] ***, *Factory Pattern*, http://www.oodesign.com/factory-pattern.html

[15] ***, Oracle Java Documentation, *The Reflection API, Uses of Reflection*, https://docs.oracle.com/javase/tutorial/reflect/

[16] ***, *Model-view-controller*, http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

[17] ***, *Code::Blocks*, http://www.codeblocks.org/

[18] ***, *NetBeans*, https://netbeans.org/

[19] ***, *CLion*, https://www.jetbrains.com/clion/

[20] ***, *KDevelop*, https://www.kdevelop.org/

[21] ***, *QtCreator*, http://doc.qt.io/qtcreator/index.html

[22] ***, *Borland Turbo C*, http://en.wikipedia.org/wiki/Borland_Turbo_C

[23] ***, *GNU make*, http://www.gnu.org/software/make/

Glossary

Abstract Syntax Tree – a tree-like representation of the syntactic structure of the program.

API – see Application Programming Interface.

Application Programming Interface – an interface through which the functionalities of a software can be accessed without exposing the underlying implementation.

AST – see Abstract Syntax Tree.

Compiler – software that transforms code from a representation to a different but equivalent representation, for example: source code to object code.

Class diagram – UML diagram which describes the structure of a system.

Design pattern – partial design solution to a common, recurring problem.

Framework – software that provides generic functionality, which is then adapted to application-specific needs.

Garbage Collector – mechanism used in some programming languages to automatically reclaim memory which is unreachable, but still allocated.

GC – see Garbage Collector.

GUI – graphical user interface.

IDE – see Integrated Development Environment.

Integrated Development Environment – a software product comprised of various tools that aid in software development.

I/O – input/output.

JSON – JavaScript Object Notation, a format used to store and transfer data in the form of attribute-value pairs.

LALR parser – Look-Ahead Left-Right parser; a parser which analyzes the input starting from the left and uses one token of look-ahead. It produces the syntax tree via reverse right-most derivations.

Model-View-Controller - architectural pattern which decouples an application into three parts; model stores data, view presents information, controller processes user input and manipulates data.

MVC – see Model-View-Controller.

OS - operating system.

Parser – the component of the compiler which performs syntactic analysis.

Sequence diagram – UML diagram that presents the objects and the interactions between them over time. Interactions are horizontal, timeline is vertical.

Sequence point - program execution moment in which all side effects of previous statements

are guaranteed to have been performed and no side effects from subsequent statements have yet to be performed.

Standard Template Library – C++ library which provides generic (template) data structures and algorithms

STL – see Standard Template Library

UML – see Unified Modeling Language.

Unified Modeling Language – standard language used to model software systems.

Use case – an expression of the system's behavior from an external point of view.

Use case diagram – UML diagram which expresses the relations between use cases and how the user interacts with the system.

XML – eXtensible Markup Language; hierarchical, tree-like format used to store and transfer data. Each information is put between an open tag and a matching closing tag, which represent the attribute's name.

List of figures

List of tables