CPSC 3720 SPRING 2018

Code:Blocks Plugin: Implementation File Generator

The Remaining Jerry's
Project Group E

Jace Riehl
Nathan Tipper
Vincent Cote

February 2nd, 2018

Table of Contents

Introduction

Welcome to the Implementation File Generator plugin for CodeBlocks developed by The Remaining Jerry's. This plugin will help the development process through code generation that is required by any developer in any project. The point of creating this project is to help save time for developers by removing the time waste of having to create the .cpp file manually every time that they create a .h file. We decided there was a need for this plug-in to help improve productivity for developers and focus on the more interesting parts of the code. With this plug-in you won't find yourself losing your train of thought while creating the .cpp file. This plug-in is intended for c++ developer using Code::Blocks. In the next sections we will be talking more in depth about how our team will implement this plug-in, our risk management, team management, and talk about our sprints.

Proposal

The idea behind the Implementation File Generator plugin is that we would like to simplify the class creation process. When a developer is creating a class, we start by making the interface for the class i.e. the header file. Once a header file is made, we have to go through the tedious process of rewriting all the signatures of our prototypes and then implementing the functionality of those functions. What the Implementation File Generator would do for us is the following:

- Create a .cpp file with the same name as the header file and save it to your /src folder or a designated folder given by the user.
- Include the header file from which the code was generated.
- Create all signatures within the header file with braces for the implementation and, if there is a return type or no variables names provided, create a return statement with a default value and default argument names.

Now the user no longer has to create an implementation file again. They simply need to create the implementation and configure their arguments and return statements to match their implementation. See figure 1.1 and 1.2 for visual reference of the idea.

The user will be able to configure what folder which folder to put the implementation file in. They will be able to choose from a .cpp or a .cc file. However, by default it will put the file in their srcs directory.

Use Cases:

A developer has created a header file and they would like to create a .cpp file. With our plugin he can save time by directly typing a keyboard command or clicking on the plugins button to immediately create the implementation file.

Project Management

Team Organization

All none code related documentation is to be in pdf format in order to keep a set standard for all team members. As this is a school project and is used as a learning tool for all team members, we understand that we are all designers, programmers, leaders and documentation gurus. With that being said the group is still structured with each member having a specific role. These roles are a guideline and will help the team know who to approach with specific issues concerning the different aspects of the project. The specific roles are as follows:

- **Design Lead:** Nathan Tipper

  Nathan will be in charge of design the main structure for the project. His main tasks as the design lead are to design the classes and UML diagrams for which our project will be based off of and answer any questions that people may have. He will ensure that everyone is following the design during the daily agile meetings.

- **Repository Admin:** Vincent Cote

  Vince will watch the 'health' of the repository to ensure that no one is making any errors when merging into the master branch. He will be in charge of ensuring that everything in the repository is properly maintained, making sure that there are no dead branches, no one is pushing useless files that will clutter the repository, and assigning merge requests to proper milestones.

- **Kanban Sensei:** Jace Riehl

  Jace will be in charge of maintaining the Kanban board on GitLab, along with the issues section of the repository. He will ensure that everyone is keeping up with the tasks that are labeled on the board and that everyone is on track to complete the sprint. His duty is to bring these up during the scrum daily meetings to ensure the progress on the project is on track.

Risk Management

With any project, we need to assess the risks involved in creating the project. The Remaining Jerry's have identified the following risks and the solutions to those risks if they arise:

- **Feature creep** - With a project that is so openly designed with no limitations other than time, it is easy to design a plug in that cannot be made within the allocated time. Therefore, throughout the development process, we will build modular code which can be built upon if time is permitted. We will reserve the right to extend the Implementation File Generator, allowing for other modules which add functionality for users to explore for the projects.
- **Lack of experience -** Throughout our school experience our assignments have been focused on small programs that process a relatively simple task. Also, these assignments only required our knowledge of C++ to complete. Integrating functionality to an already existing application is a task we have not looked at yet. Therefore, a great deal of research is needed to complete this project. We will need to assess what the design constraints are early so we don't run into any implementation problems later in the project. If any team members are not knowledgeable

about any tools that we're using for this project it is their responsibility to bring it up in the daily meetings. From there a team member who is knowledgeable on that tool will guide the developer become familiar with it.

- **Busy schedules -** All members of the team are taking classes which require a lot of attention. Therefore, coordination will be key to success throughout the development process. Slack, and weekly meetings will need to well structured in order to keep workflow positive and moving in the right direction.
- **Changes in implementation** - If it is required that we change the implementation, we will talk about it in a daily scrum meeting or hold a new meeting if necessary, and agree on a plan to create new implementation. These plans may include simply modifying the UML diagrams and the classes or in the worst case reverting back to a previous version of the project.

Agile Processes

Our team will be using the agile processes scrum and kanban to implement this plug-in. The GitLab board will be our main use of kanban to keep track of where we are in the sprint. Each sprints board will be split into four section; backlog, to-do, doing, and completed. The backlog will be used to store ideas that will be dealt with when all of the other to-do's are done. If they aren't completed in that sprint they will be moved to the next sprint, this also relates to the scrum process we're using. We will be incorporating scrum by splitting our project into three two week sprints. Each of the sprints will take from the backlog of the previous sprint and the team members will put it into action. When possible we have quick daily meetings saying where we are in the process, explain and problems we're having, and assign new tasks for the team member.

Development Process

Coding Conventions

Throughout the lifetime of this project, our team has decided to follow the following coding conventions:

- **Starting blocks on the same line of any scope.**
  - Ie :

```
Class Class1 {
  ...
};
```

- **Using camelCase for both local variables (attributes) and functions (members). For member variables that are private, we will use put a underscore before the variable to indicate it is private data.**
  - Ie:

```
int thisIsAVariable = 1;
void thisIsAFunction() {
        ...
}

int _thisIsAPrivateVariable;
```

- **Class names are to be capitalized and then use camelCase**
  - Ie:

```
Class ThisIsAClassThatDoesNotFollowExistingConventions { ... } ;
```

- **Variables, functions, and class names should be clear and concise and be named with respect to their function.**
  - Example:

```
int numberOfJerrys = 3;
```

- **Comments should be short and give a broad overview, leaving the variable, function, and class names to speak to the functionality of the code.**

- **All nested scopes should be indented a level in for a readability.**

- **Each header file should start with a ifndef, def statement to ensure header files are not duplicated throughout the code. The defined statement should be read as the name of the class in all capital letter followed by "_H."**
  - Example:

```
#ifndef SOMECLASS_H
#define SOMECLASS_H
```

```
class SomeClass { ... };

#endif // SOMECLASS_H
```

- **Use accessor methods for users to use instead of public attributes.**

- **Documentation of all members in header files is recorded and is required to have all of the following:**
    - ❖ What the function does.
    - ❖ Description of parameters and what the return value is.
    - ❖ How the function modifies the object.
    - ❖ The pre-condition and post-conditions of the function.
    - ❖ Any restrictions the function may have.

Requirements

Essential

- There is a button which will look at a header and generate the .cpp and all signatures are generated with scope operators and the header file included.
- The functions auto-created to the .cpp files have the correct return type with default values.
- An abstract method is not generated in the .cpp file.
- If changes have been made, make sure you it checks with the user to override the current .cpp file.

Desirable

- New functions will be added to the .cpp file automatically after a each function is added to the .h file (without hitting a “run” button).
- If the button is pressed after the .cpp file is created it will generate the new functions and delete the old not listed functions.
- Add a wizard to set up where to locate the .h and and where to put .cpp files for different file structure in a project.

Optional

- Add refactor tool to rename functions across the entire project .
- As private variables are typed, ask user if they would like to add a getter and a setter for the variable.
- Ask if the user wants to add initialization list for their private.

Procedures for Configuration Management

Code Review Process

Every member of the team is involved in the Code Review process. Each member will have a branch for the work that has been assigned to them. When a member has completed the work, they will create a merge request to merge their work back into the master branch. All other members will be notified and any other member is responsible for the review of the code and the merging process. The philosophy behind this is that the member who created the merge request is not able to merge it back into the master. It must be reviewed by another member and that member can choose to either merge back if they think the work is complete and all guidelines are followed, or they may choose to leave a comment that it should be fixed in some way or ask for another member to review it as well.

Communication Tools

The primary methods of communication for the duration of the project will be face to face meeting as well as using slack for text based communication regarding general information, random information, design documentations, structure and repository documentation; all of which are subdivided into their own channels. If face to face meetings are not feasible due to scheduling conflicts the team is to arrange a video conference call on Discord at the earliest opportunity.

Change Management

Any group member will be able to issue bug reports that they encounter from the program through GitLab. When a bug is found, the reviewer who identified the bug(s) will assign them to the developer who wrote the function/class. Only the reviewer who identified the specific bug is allowed to close it. Once the bug has been resolved, the developer is to report back to the reviewer who will test it again until it is agreed that it is fixed or that it is more of a feature than a bug; at which point the issue will be closed through GitLab.

Design

You can find our UML diagrams for the design of our project along with what our final product will look like in the appendix section of this document, figure 2.0, 1.1, and 1.2. We decided to split our class into a couple classes that will allow us to follow the SOLID+Dry principles.

Sprint Retrospection

Sprint #1

We decided to focus our sprint around learning the Codeblocks API and having our deliverable be a generated file. We used the scrum and kanban agile processes to keep ourselves up to date with everything going on with the project. Since most of the project was dedicated to scraping the Code::Blocks documentation for relevant information, Vincent and Jace had the job of reading through the documentation and testing the classes while Nathan did a majority of the coding that would be pushed to the repository. This worked exceptionally well for us. While Nathan coded, Vincent and Jace could find and test relevant information for him to use.

This sprint had its challenges but overall it ended well. As expected a majority of this sprint was spent reading and learning the documentation of the Code::Blocks API. Scraping the API for key information was difficult at first. Most of the documentation does not explain what each class does which lead to frustrations among the group. However, these were soon solved when we found the different manager classes and their key attributes and functions. Using the manager classes ended up being the bulk of what our project would depend on. In the end of this sprint we managed to meet and exceed our expectations. It was expected from the beginning of pitching this project to Dr. Anvik that we would only be able to create a new file. However, in this sprint we've managed to create a new .cpp file from a .h file which has an include for the .h file. This new .cpp file opens in a new window, adds it to the project, and reloads the UI so that the the file will show up in the project.

One issue we ran into was with creating the Makefile. We quickly realized that we needed to find all the dependencies on the system and where not able to locate all of them. The lack of a Makefile ended up also affecting continuous integration. We talked with Dr. Anvik about this issue and chose to look into making a Makefile for the class. We were required to push continuous integration implementations to sprint 2. However, we continued with our system level testing to ensure it works as expected. Since this sprint was mainly about learning the API and creating a blank file, so it won't be a big hindrance to our sprint. The work we did in this sprint was mostly prototyping to make sure we understand the calls we are making using the Code::Blocks API. Now that we have a proof of concept, sprint #2 will include properly implementing our prototype including documentation and unit testing.

We initially had a problem with getting our example plugin to install into Code::Blocks and it didn't give us any hint as to why. After reviewing our code we had to resort to asking Dr. Anvik about this problem due to our lack of experience with API's. He found the problem and it was a quick fix where we only had to change a few lines. There was one another major problem that we had. The only way we could get the new file to show up in Code::Blocks was to reload Code::Blocks but this would be a hindrance to the user as it would prompt them to save the project because Code::Blocks had changed and would close the file structure tree. We found another class that we could use that reloaded just the UI instead of the whole IDE and this fixed every problem that we had associated with it.

Our main agile practices were scrum, the kanban boards contained on the GitLab repository, and we used slack as our main communication tool . Whenever progress had been made to the project we did a scrum stand up meeting the next day. These agile processes worked well for our team. Having the daily

meetings helped us stay on top of our duties, and increased our productivity. The scrum meetings combined with slack was a immensely productive way of improving our communication.

The three of us have worked together on most class projects for the last year. With that being said we know what to expect from each other and have found good ways to work together. Each of us stayed to our individual roles that were assigned in the team organization section , the remaining responsibilities that we're specific to each of our roles were split how we saw fit in the scrum meetings. An example of these tasks would be who was going to write this report.

As mentioned by Dr. Anvik, 80% of programming as a software developer is focused on working on existing code. This sprint showed us the importance of learning how to go through documentation efficiently in order to find the information needed to accomplish a task. We also learned the importance of writing effective documentation when writing code. The Code::Blocks documentation is a good example of how insufficient documentation can be frustrating for other developers trying to use the code.

Our plan worked very well for us this time and since we've scraped most of the relevant information from the documentation, we won't have to do the same approach in the next sprint. In the next sprint Nathan will continue to be the lead programmer while Vincent and Jace provide programming support. These duties may include but are not limited to: writing unit tests, refactoring code, finding memory leaks, setting up continuous integration, and coding some implementation. Holding the daily scrum meetings was highly valuable to the success of our team. It helped us communicate our issues effectively and find a solution to them before they became a big problem. Kanban was also an effective agile strategy that helped us stay on track by having a visual representation of what had to be done. There are a couple things that we will integrate into our next sprint. Now that we have a working prototype, we'll spend more time designing the system so that problems won't arise later on in the sprint.

Appendices

Appendix A: Figures and Tables

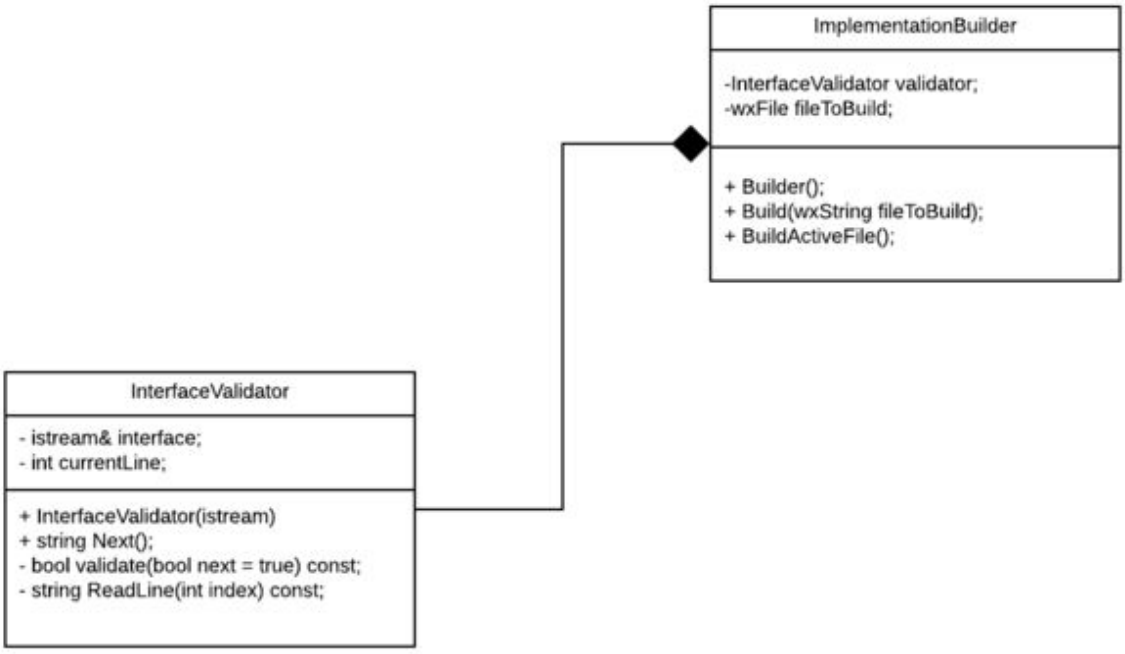
Figure 1.1

```
#ifndef CALCULATOR_H
#define CALCULATOR_H


class Calculator
{
    public:
        Calculator();
        virtual ~Calculator();

        int sum(const int, const int) const;
        int mult(const int, const int) const;
        int sub(const int, const int) const;
        int div(const int, const int) const;

    protected:

    private:
};

#endif // CALCULATOR_H
```

Figure 1.2

```
Calculator::Calculator()
{
    //ctor
}

Calculator::~Calculator()
{
    //dtor
}

int Calculator::sum(const int a, const int b) const {
    return -1;
}

int Calculator::mult(const int a, const int b) const {
    return -1;
}

int Calculator::div(const int a, const int b) const {
    return -1;
}

int Calculator::sub(const int a, const int b) const {
    return -1;
}
```

FIGURE 2.0


ImplementationBuilder
-InterfaceValidator validator;
-wxFile fileToBuild;
+ Builder();
+ Build(wxString fileToBuild);
+ BuildActiveFile();
InterfaceValidator
- istream& interface;
- int currentLine;
+ InterfaceValidator(istream)
+ string Next();
- bool validate(bool next = true) const;
- string ReadLine(int index) const;