Appendix

Efficiency of Projectional Editing: A Controlled Experiment

Thorsten Berger, Markus
Völter, Hans Peter Jensen, Taweesap Dangprasert, Janet Siegmund

Abstract

This appendix contains details about our preparatory survey, the experimental material (incl. the task descriptions), and statistics not in the main paper. On publication of the main paper, this appendix will be published as a technical report.

Contents

1 Survey **1**
- 1.1 Questionnaire Design . . . 1
- 1.2 Analysis . . . 2
- 1.3 Results . . . 2

2 Experiment Material **6**
- 2.1 Repositories . . . 6
- 2.2 Training Session Handout . . . 6
- 2.3 Task Descriptions Handout for Groups Proj, ProjE . . . 9
- 2.4 Task Descriptions Handout for Group Par . . . 13

3 Detailed Additional Statistics **17**
- 3.1 Basic Editing and Errors . . . 17
- 3.2 Modification and Refactoring Operations . . . 19

1 Survey

The first phase of our study aimed at obtaining preliminary insights into perceived benefits and challenges of using a projectional editor. To this end, we surveyed professional developers using an online questionnaire. We used the results to formulate research questions, corresponding hypotheses, and to design experiment tasks.

1.1 Questionnaire Design

The questionnaire comprised 31 questions. They elicit the background of participants (e.g., previous programming experience, knowledge of MDD techniques, language-design experience); participants' perceptions of using the editor, including advantages and disadvantages compared to parser-based editors; and detailed experiences on source-code editing.

We used a mix of closed and open-text questions. The closed ones have a five-point Likert scale [1] ranging from *strongly agree* over *neutral* to *strongly disagree*. The open-text questions explore detailed editing aspects that participants found either positive or negative. Therefore, every Likert-scale question was followed by an open-text one. For instance, after requesting participants to state to which degree they agree with the statement that "it was easy to learn and understand the facilities of MPS", we asked them "which facilities were easier and which were harder to learn."

Consequently, the questionnaire was relatively long, taking around 20 minutes to complete. This length had no negative impact on the recruitment, since we invited a controlled sample of participants who are our industry contacts and known to us for having worked with MPS. The invitees were encouraged to forward the invitation.

1.2 Analysis

We analyzed the survey quantitatively (Likert-scale questions) and qualitatively (open-text questions). For the Likert-scale questions, we mainly used diagrams, while we analyzed the open-text questions with a grounded-theory-like approach [2, 3] by using open coding [4] to identify editing-related aspects, such as specific facilities of the editor, an experienced peculiarity, or a specific behavior that participants perceived as positive or negative when working with the editor. Identification of these aspects was based on reaching consensus among at least three authors. For each identified aspect, we calculated a ratio of positive and negative statements about it in the responses—to identify tendencies towards a positive or negative perception by the participants.

1.3 Results

We received 21 responses, with 19 of the participants also providing open-text answers. From these, we identified 49 editing-related aspects, which we structured into seven categories: *basic editing*, *errors*, *AST conformance*, *navigation*, *refactoring*, *language design*, and *general*. We removed aspects not directly related to projectional editing, such as CamelCase navigation, IDE user interface (*"MPS is rather logically structured"*), and all aspects from the categories *navigation* and *language design*. As a result, we obtain 18 aspects belonging to four categories: *basic editing* (10 aspects, sub-divided into *editing operations* and *other aspects*), *errors* (3 aspects), *AST conformance* (2 aspects), and *refactoring* (2 aspects).

In the following, we first present general results about efficiency and productivity of MPS and then discuss the identified aspects.

1.3.1 Overall Efficiency

Most developers agree that they can write code as fast as with a conventional, parser-based IDE (median: 2; min/max: 1/5). Only one developer strongly disagrees, but explains that he is a proficient Emacs user: *"Years of investment in Emacs are hard to beat."*

One user also disagrees, but indicates that this is because he is a novice MPS user. A second developer who disagrees states that while code entering may not be that efficient, it is less error-prone, increasing overall efficiency. The remaining participants state that after getting used to the different style of entering code, there is no difference in editing efficiency to a conventional editor.

We also asked about the general perception of productivity with MPS. Most developers are positive in this respect (median: 2; min/max: 1/5). While 28% express a neutral opinion, 40% agree, and 28% even strongly agree. Only one participant expressed strong disagreement. This participant also faced intensive learning effort and stated that becoming familiar with the environment was difficult, mainly since all of MPS' concepts were completely new to him. In contrast, he strongly agrees that he can write code as fast as with a parser-based editor, arguing that the code-completion facilities significantly contribute to the productivity. We conclude that after a learning phase, MPS lets developers work efficiently and productively.

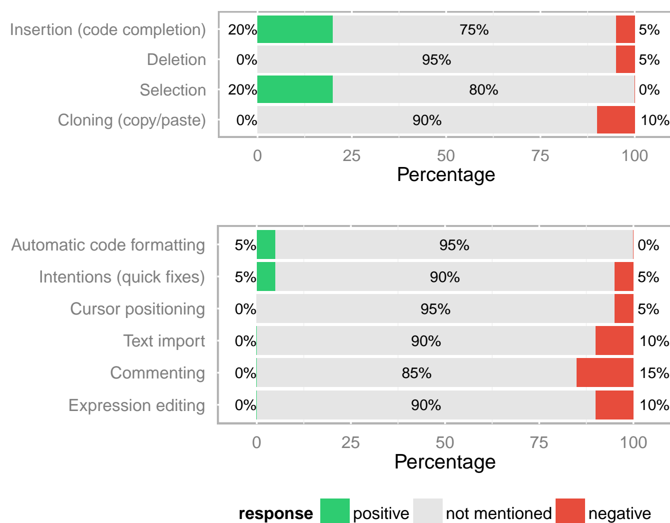


Figure 1: Perception of basic-editing operations (top) and other related aspects (bottom)

Users also see the bigger picture: All participants agree that they benefit from the modular language support of MPS (median: 1, min/max: 1/2), confirming one of its key benefits: *"Language composition is the main strength of MPS."*

Finally, the flexible notations provide a considerable benefit for developers (median: 2, min/max: 1/3), especially for integrating stakeholders from different domains: *"My DSL users are business people, not IT people. Being able to use mathematical notations for Sum and Product expressions, fraction bars for division, tabular notations for test cases is crucial."*

1.3.2 Basic Editing

This category summarizes responses about the fundamental editing of code. Fig. 1 presents our participants' perception of basic-editing operations (top) and of other mentioned aspects (bottom) in this category.

Basic-Editing Operations. Comments on *Insertion*—the ability to efficiently enter code—are dominated by the code-completion facilities of MPS, which are seen as a strong advantage: *"Code completion and working directly in AST make things easier."* In contrast, *Deletion* is criticized when mentioned. As it does not operate on a character-by-character basis, it often deletes too much code.

The *Selection* of code (i.e., AST nodes) is only seen positively by our participants, despite a behavior that is different from parser-based editors: *"Selection of program fragments happens based on program structure, not on cursor positions."*

Finally, *Cloning* is perceived negatively, but the corresponding comments show that this shortcoming is related to the enforced AST conformance (discussed shortly in Sec. 1.3.4): *"Things look like text but are not [real] text (one cannot copy/paste some piece of code in another place where it would make sense)."* Here, the participant emphasizes that copied code cannot be pasted into all places where it would fit syntactically, but not based on the AST.

Other Basic-Editing-Related Aspects. Most users are satisfied with the modern editing facilities, such as automatic code formatting and intentions (little user-initiated in-place program transformations, also known as quick fixes in other IDEs). On the negative side, participants complain about unpredictable cursor positions and restrictions on arbitrary code commenting: *"The editing of text is not straightforward, but also not too difficult. The more this will be improved (delete operations, cursor positioning, comment editing), the less this point comes into picture. I feel that in future it should be only neces-*

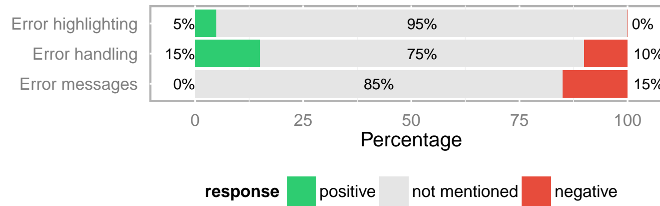


Figure 2: Perception of error-related aspects

sary to know about Ctrl+space and Alt+enter." This comment also emphasizes that code completion ('Ctrl+Space') and intentions ('Alt+Enter') could potentially be the main editing operations relevant to projectional editing.

Another shortcoming expressed by two participants is the import of code (i.e., copy/paste) from sources outside of MPS. Finally, two participants explicitly point out the problem of editing expressions, which is easier by rewriting the respective code: *"For changing existing model elements (e.g. putting parentheses around a part of an expression in mbeddr) [...] it is often less work to just remove the whole part you want to change and redo it, rather than trying to change it in place."*

1.3.3 Errors

This category comprises mistakes and unintended editing results that require correction.

In a Likert-scale question, we explicitly asked about participants' agreement with the statement "With MPS, I make less errors while programming." The median of responses agrees with this statement. Those who agree or strongly agree, state that the error prevention is related to the enforcement of valid ASTs: *"MPS catches or prevents most syntactical or spelling mistakes. A well designed language with its type definitions prevents or catches more complicated mistakes. Only mistakes at a really execution or algorithmic level slip through."*

One participant agrees, but emphasizes the dilemma of not having hard empirical data: *"I tend to agree [...] although I am not sure how to quantify this."* Interestingly, 42 % of our participants express only a neutral opinion about error reduction. Some explain that any such achieved via projectional editing is minor compared to semantic/logical errors in the code: *"The main type of errors [is] logical errors, which are not influenced by the IDE used."* Others point out that error reduction is tied closely to the language design, and is less a conceptional problem of projectional editing: *"Depends on the DSL and how good it is implemented."* Finally, one participant disagrees and another one even strongly, since they did not experience any error reduction at all: *"I make the same amount of errors."*

Analyzing all open-text responses led to three further aspects beyond error reduction, whose perception is shown in Fig. 2. Participants like how the projectional editor highlights and handles errors. Although hardly generalizable to projectional editing, some participants (15 %) state that error messages are hard to interpret and not very helpful when fixing mistakes. While most of these messages are language-specific (violation of metamodel constraints), MPS in fact has some difficult to understand built-in error messages that cannot be changed by the user. These messages reflect the implementation of MPS instead of end-user-relevant concerns. Examples include "Abstract concept instance detected" or "Not in search scope".

1.3.4 AST Conformance

This category includes all statements about the enforced conformance of programs to a correct AST at any time.

In a Likert-scale question, we asked participants about their agreement with the statement "I like that I can only produce structurally correct programs (valid ASTs)." The median of responses agrees. One

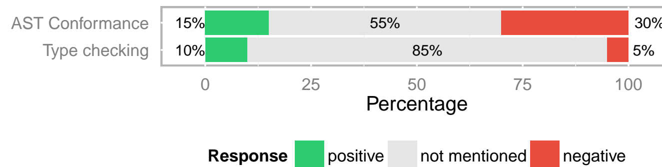


Figure 3: Perception of conformance-related aspects

strongly agreeing participant explains: *"This helps a lot. If you need [...] incomplete trees, you need to think about your language design and make it explicitly possible. [...] You can actually create incorrect trees with many red errors [...]."*

This comment also describes the problem of references between nodes (e.g., references to variables, methods or types) in the tree. Such references can be unbound—for instance, when the target does not (yet) exist. Such inconsistencies need to be resolved at least before persisting the program, except if the language explicitly supports dangling references, for instance with placeholders (node proxies). In other words, allowing structural inconsistencies is a responsibility of the language developer, not the user in projectional editing.

Participants' open-text responses reveal two further aspects they are concerned about: enforced AST conformance and type checking, as shown in Fig. 3.

Six participants elaborated explicitly on the aspect that AST conformance is always enforced. Most comments were negative; many respondents would like to—at least temporarily—violate AST rules during the development: *"Invalid ASTs are also necessary during development."* Yet, three participants (15 %) found the enforced AST conformance very helpful. For instance, two appreciate that the program is *"correct by construction"*, one likes the *"syntax-directed editing"*, and another one elaborates: *"At some point, the brain automatically switches itself off from ensuring all this and instead focuses more on logic or actual intent of the program."*

The error reduction discussed before is mainly attributed to the enforced AST conformance. Most importantly, respondents emphasize the absence of *"syntactical or spelling mistakes"* (typos), and that: *"Errors can be found at edit time instead of compile time or (worse) runtime."* Yet, two respondents do not see the permanent AST conformance as a benefit over parser-based editors. One explains that *"Syntactical correctness is easy anyway."* Another participant elaborates: *"If the compiler is incremental and fast enough (like Eclipse's Java compiler), you get immediate feedback on syntax errors, so this is not a big advantage."*

In summary, most participants consider the enforced AST conformance not as a benefit in itself. However, many are willing to accept it as a side-effect of projectional editing, which provides tangible advantages in other respects (notation, language extensibility). The main problem seems to be the fact that references can only be created after the reference target has been created, unless the language developer expressively creates means to allow otherwise—such as quick fixes to create targets on demand or proxies that support "dangling" references.

1.3.5 Refactoring

This category comprises aspects related to our participants' experiences with refactoring code.

When mentioning refactoring, our participants express a solely positive experience—for instance, confirming that *"Renaming/refactoring always work."* This is owed to the fact that references between program elements are immediately bound during editing (using code-completion) [5], and changes to the name of the refernce target are automatically and immediately propagated to references.

We also identified comments on moving code, which is often used for refactoring or extending code. Our participants expressed ambivalent experiences—for instance, that *"The lack of freedom to move*

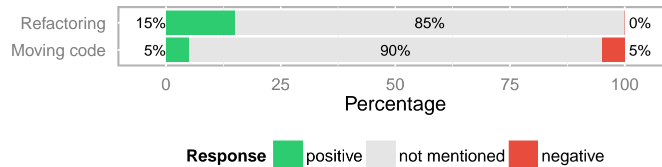


Figure 4: Perception of refactoring-related aspects

around is sometimes restricting" or that *"Moving code fragments with Ctrl+Shift arrow up/down and automatic code formatting is also very important/convenient."*

2 Experiment Material

This section contains a description of (and pointers to) the materials used for the experiment. Recall that we provided a training session, whose material is also provided below, before the actual experiment.

2.1 Repositories

The experiment groups had an installation of MPS and Eclipse CDT with fully setup projects: one for the training session ("Hello World") and one for the actual experiment session ("Problem"). The projects for MPS, used by the beginners (Proj) and experts (ProjE) with a projectional editor, are available in the ZIP file `repo-Proj,ProjE.zip`. Note that we did not provide a dedicated training session (project "Hello World") for the experts. The projects for Eclipse CDT are contained in the ZIP file `repo-Par.zip`. In both repos, note that the tasks were enumerated beginning with 0, i.e., task 1 in the paper is contained in the problem0 folder. Also note that problem4 and problem5 were solved by our participants, but did not become part of our analysis, since comparing graphical editing (e.g., decision table) in MPS with text editing in Eclipse (e.g., large switch statement) was beyond the scope of our present analysis. Since these two tasks appeared at the end of the experiment, they did not influence the previous tasks (internal validity).

2.2 Training Session Handout

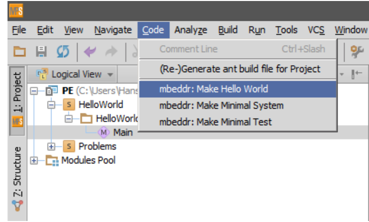
Before the experiment, we provided a 45-minute training session for the groups using a projectional editor (Proj, ProjE). The training session was described on the following handout.

Learning

In this phase we would like you to get familiar with the projectional editor.

We have created a small project called **PE**. Inside this project you should find two solutions: one called **HelloWorld** and one called **Problems**. In this phase you are only to work with the **HelloWorld** solution. Inside this solution we have placed a model called **Main**. Under this model you will be performing all of your tasks for the learning part.

Let us start with the usual HelloWorld program. The easiest way to do this is to use the included functionality provided by mbeddr. In the Logical View make sure you have selected the **Main** model. Now go to the code menu item and select the **mbeddr: Make Hello World**.



Under the **Main** model, you should now find three new items:

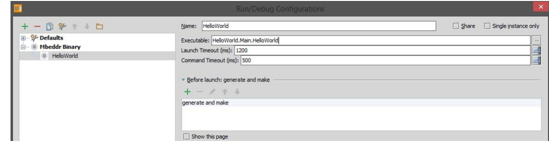
BuildConfiguration This specifies which modules should be compiled into an executable or library, as well as other aspects related to creating an executable. You will need to edit this later.

HelloWorld is an implementation module. Modules act as namespaces and as the unit of encapsulation. You are requested to insert C code and edit such modules as we go along.

TypeSizeConfiguration This specifies the sizes of the primitive types (such as int or long) for the particular target platform. The one you have created is the mbeddr C default configuration. If you have two *TypeSizeConfigurations* you can delete the bottom one, as we will only need one. You will not need to edit this during this session.

Open the HelloWorld module and confirm that a main function has been generated. Do not worry about the *messages*. It is a special language extension for logging.

Before you can run it, we need to create a **run configuration**. Go to *Run → Edit Configurations* found in the menu bar. In the *Run/Debug Configurations* window that appears, select the green plus-sign in the upper left corner. Be sure to select the option **Mbeddr Binary**.



In the options pane to the right, name it *HelloWorld*. In the **executable,** use the ellipses (...) to select the **HelloWorld** Implementation module. Use the default values for the remaining options. When you are done, click *OK*.

We are now ready to run our HelloWorld program. Use the shortcut **Shift + F10** to run it. Confirm that the console prints out *Hello, World!*.

Now that we have a working program, let us start with some basic expressions. In the logical view, select the *Expressions* module. To get a feel for the editor, let us create a basic expression, write a test, and run it as part of our Hello World program.

In the *Expression* module, place the cursor in the body of the module under the horizontal black line. Notice that you can move around in the module using the shortcuts **Tab** *or* **Shift + Tab** instead of using the mouse.

Note that in a projectional editor, code completion (**Ctrl + Space**) is heavily used. Be sure to make a note of this, as writing words out character by character might not produce a valid program in certain contexts. The word will be marked in a red color to signal that it is not yet valid. If you do not already use this shortcut (**Ctrl + Space**) extensively, be sure to try and do so as part of this experiment session, as it is a core action when working with a projectional editor such as MPS.

Now, try to write **int** and invoke code completion from there. You should be presented with a set of possible options. Choose the smallest integer representation **int8** (this is mapped to a char in the *TypeSizeConfiguration*), which is fine in our small example. Press Tab or Right Arrow to navigate to the variable name and name it *i*. Now, in turn, press: **=**, **1**, **+** and **2**. You should now have an expression that looks like this:

```
int8 i = 1 + 2;
```

Now, try to place the cursor directly after 2, but before the semicolon. Press **Backspace**. Notice only the *2* is deleted. Now, place your cursor after the semicolon and press **Backspace**. Notice how the *entire* declaration is deleted (The reason for this is that you are deleting the function node, and thus the entire subtree under it is deleted). Thus, be sure to pay attention to the cursor position when doing delete actions. Undo the deletion by using the undo action (**Ctrl + Z**). In fact, you can always use undo action (**Ctrl + Z)** if you get in trouble.

Let us define a function that returns the result from this declaration.

Start by writing out the return type, which is **int8**. Then write the name of the function, in this case *Add,* and finally type an opening parenthesis. This should give you a function. Afterwards, **Tab** or use **Arrow Keys** to the body of the function and add the *return* statement, which should be **1 + 2**.

To make sure that the function really does return *3* as we expect, let us create a testcase for it. For this we will use the provided language extension in mbeddr for testing. Place your cursor in the expression module after the *add*() function. Write **test** and do code completion from there. You should get a testcase constructed for you.

Now, in turn, do the following:

1. Name it **testAdd,**
2. **Tab** to the body,
3. Write **assert** and invoke code completion
4. Select the option that is just called **assert**
5. In the *<no expr>* tag, type **add** and do code completion, if necessary.
6. Press **Right Arrow** to move behind the right-parenthesis.
7. Type **==**.
8. Type **3**.

You should now have a test case that looks like this:

```
exported testcase testAdd {
  assert(0) add() == 3;
} testAdd(test case)
```

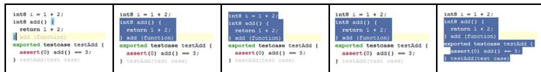
In order to run this test case, we need to add it to the Hello World program we created earlier. Start by navigating to the HelloWorld module. Inside the top of this module, locate the **import** *nothing* keywords and use code completion(you may have to invoke twice) to replace *nothing* by *Expressions*. Now, go to the return statement in the main function and delete the *0*; instead, write **test**. If needed, invoke code completion and select the *test* in the options available.

Selection is different in a projectional editor. For instance, you cannot select blocks of code using the mouse. Instead, you have several options available to perform selections.

You can select along the tree using the **Ctrl + Up/Dow**n shortcut. The use of this shortcut expands/shrinks block selection region. The example below illustrates continuous presses of the **Ctrl + Up** shortcut (be sure to notice the initial cursor placement)**.**

A similar shortcut is **Ctrl + W**, which successively selects increasing code blocks.

You can also select siblings using the **Shift + Up/Down** shortcut. Using this shortcut you can extend the selected region to siblings. The example below illustrates two **Shift + Up** followed by two **Shift + Down** presses(again, notice the initial cursor placement).



Another core feature in the editor is the *Show Intention* action (**Alt + Enter**). In some editors this is also known as *Quick Fix*. In contrast to code completion, intentions can perform transformations on the program (e.g., to add additional code to your program), whereas code completion can only present the options currently available. The type of intention that can be invoked is context-specific.

Let us use the *Show Intention* action (**Alt + Enter**) to add all the visible test cases that is available from the main function. Make sure your cursor is placed on the test keyword that we wrote as the return value before. Now, use the *Show Intention* action (**Alt + Enter**) and select the option called *Include all visible Test Cases*.

As the last thing, go the the **BuildConfiguration**. Place your cursor inside the **Binaries** section on the red error line and show intentions (**Alt + Enter**). Select the **Add Missing modules** option.

When you are done, run your HelloWorld Program again (**Shift + F10**). In the console verify that Hello World is still being printing, but also that our testcase is run as well.

We have now introduced a set of common operations and a set of Shortcuts to help you in your work with the editor. All of the mentioned shortcuts is available from the cheat sheet provided.

Refactoring

We have now gone through a few simple tasks when working with editors. Now, let us proceed to some common refactorings. Recall that refactorings are changes to the code that do not alter its behaviour.

Start by navigating to the HelloWorld module. Inside the top of this module, locate the **imports** keyword and add all the following modules: *ExtractVariable, Inline, ModuleA, ModuleB,* and *RenameMe*. You add a new line by pressing **Enter** and then either type out the module name or use code completion to fill in a module.

Furthermore, use the *Show Intention* action (**Alt + Enter**) to add all the now visible test cases. Select the option called *Include all visible Test Cases*. Recall your cursor should be placed on the **test** keyword

Then, as we did before, go the the **BuildConfiguration**. Place your cursor inside the **Binaries** section and show intentions (**Alt + Enter**). Select the **Add Missing modules** option. Confirm that all the five modules mentioned before are added.

Rename

Perhaps the most common refactoring is a rename refactoring. **Rename** refactorings allow you to rename symbols, automatically correcting all references in the code. In the logical view, select the Refactoring folder and open the *RenameMe* module.

First of all, notice that the name of the module is spelled incorrectly. Try correcting it directly without using the **Rename** refactoring. Notice that this works without introducing errors.

Now consider the table below. Refactor the left code fragment (Pre-refactor column) to the right code fragment (Post-refactor column).

Pre-refactor	Post-refactor

`string x = "ACKS-RAY";`
 	
`string y() {`
`  return "YANG-KEY";`
`} y (function)`
 	
`exported testcase testRename {`
`  assert(0) x == "ACKS-RAY";`
`  assert(1) y() == "YANG-KEY";`
`} testRename(test case)`
	`string xray = "ACKS-RAY";`
	
	`string yankee() {`
	`  return "YANG-KEY";`
	`} yankee (function)`
	
	`exported testcase testRename {`
	`  assert(0) xray == "ACKS-RAY";`
	`  assert(1) yankee() == "YANG-KEY";`
	`} testRename(test case)`

Observe how the name changes instantly across its uses. Confirm that your changes did not break anything by running the test case in the module.

Extract Variable

Another very common refactoring is to extract a variable. The **Extract Variable** refactoring puts the result of the selected expression into a variable. It declares a new variable and uses the expression as an initializer. The original expression is replaced with the new variable. Our example is of the form **Extract Local Variable**. In the logical view, open the *ExtractVariable* module.

Now consider the table below. Refactor the three left code fragment (Pre-refactor column) to the right code fragment (Post-refactor column).

You can find this refactoring in the context menu under refactorings. The menu is context sensitive, so make sure that you place the cursor at the correct position.

Pre-refactor	Post-refactor

`int8 preExpression1() {`
`  return 42;`
`} preExpression1 (function)`
	`int8 postExpression1() {`
	`  int8 MeaningfullConstant = 42;`
	`  return MeaningfullConstant;`
	`} postExpression1 (function)`

`int8 preExpression3() {`
`  int8 c = 3 + 2 + 1;`
`  return c;`
`} preExpression3 (function)`
	`int8 postExpressions23() {`
	`  int8 a = 1;`
	`  int8 b = 2 + a;`
	`  int8 c = 3 + b;`
	`  return c;`
	`} postExpressions23 (function)`

Confirm that your changes did not break anything by running the test case (**Shift + F10**) in the module.

Inline

The **Inline Variable** refactoring replaces redundant variable usage with its initializer. This refactoring is opposite to the **Extract Variable** from the previous section. In the logical view, open the *ExtractVariable* module.

Now, again, consider the table below. Refactor the left code fragment (Pre-refactor column) to the right code fragment (Post-refactor column).

Hint: use the Inline refactoring command from either the context menu or shortcut **Ctrl + Alt + I**

Pre-refactor	Post-refactor

`void PreInlineVariable() {`
`  int16 number = 1000;`
`  int16 b = number + 10;`
`} PreInlineVariable (function)`
	`void postInlineVariable() {`
	`  int16 b = 1000 + 10;`
	`} postInlineVariable (function)`

It should be clear that the refactoring did not alter any behaviour, but to be sure, run the provided test case (**Shift + F10**).

Move

Move refactorings allow to move nodes around within a project. For instance, a function or a variable declaration. Our example is of the form **Move to Imported Module**. In the logical view, open the *ModuleA* module.

Now consider the table below. Refactor the left code fragment (Pre-refactor column) to the right code fragment (Post-refactor column).

Pre-refactor	Post-refactor
Module A (empty) **Module B** (empty) **Module C**: *functionA2() :void*, *functionA1() :void*, *functionB() :void*, *functionC() :void*	**Module A**: *functionA1() :void*, *functionA2() :void* **Module B**: *functionB() :void* **Module C**: *functionC() :void*

For this exercise we would like you to move some of the functions; functions that contain *A* should be moved to *ModuleA* and functions that contain *B* should be move to *ModuleB*. You can use either copy pasting or the **Move to Imported Module** refactoring (**Ctrl + Alt + M**).

The steps are outlined table below

Step #	Action
Step 1	**Module C**: *functionA2() :void*, *functionA1() :void*, ***functionB() :void***, *functionC() :void* « Move » **Module B**: ***functionB() :void***
Step 2	**Module C**: ***functionA2() :void***, *functionA1() :void*, *functionC() :void* « Move » **Module A**: ***functionA2() :void***
Step 3	**Module C**: ***functionA1() :void***, *functionC() :void* « Move » **Module A**: *functionA2() :void*, ***functionA1() :void***
Step 4	**Module A**: ***functionA1() :void***, ***functionA2() :void*** « Move »

Make sure that the function A1 is placed above function A2. A handy shortcut to do this is to place the cursor behind the function or selecting the node and pressing **Ctrl + Shift + Up/Down**. The selection will be moved in the direction of choice.

```
void functionA1() {

} functionA1 (function)

void functionA2() {

} functionA2 (function)
```

9

2.3 Task Descriptions Handout for Groups Proj, ProjE

The beginners with a projectional editor and the industrial participants who were experienced with projectional editing received the following handout describing their tasks.

Problem Phase

Now that you have completed the learning phase you should be better able to work with a PE. We will now present a set of problems that we ask you to solve.

In this phase you are only to work with the **Problems** solution. Inside this solution we have placed a model called **Main**. It is subdivided into folders; one for each problem that you should attempt to solve.

In the root of the **Main** model we have placed a **Test** implementation module. This module contains a test for each problem. The criteria for solving the problems is having all the test run without errors.

We have provided a run configuration called **Problems** that runs this module. At any time you wish to test your solution you can run this and have your solution verified.

Problem 0 - Logic

Recall the propositional equivalences known as tautologies(perhaps from a Discrete Math course you have previously taken). If you do not recall them they will be presented here and as we go along.

Double negation law	$\neg(\neg p) \equiv p$
Commutative Law	$p \vee q \equiv q \vee p$
Associative Law	$(p \vee q) \vee r \equiv p \vee (q \vee r)$
Distributive law	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
De Morgan's Law	$\neg(p \wedge q) \equiv \neg p \vee \neg q$
Absorption Law	$p \vee (p \wedge q) \equiv p$

Your task is to complete all the laws in the provided code. That is, you should correct the expression so that it is identical to the tautology. For instance, by moving parenthesis, adding an expression or fixing a defect in the statement. We have already implemented the Double negation law to get you started, so you will know how we expect you to solve the task.

Now go to the **Logic** Module in the **problem0** folder and start the task presented in the next pages.

When you are done run the test(**Shift + F10**) and make sure that this problem does not fail.

Double negation law

The Double negation law is defined as

$$\neg(\neg p) \equiv p$$

Your task is now to implement the Double negation law given the **Before** implementation, so that is identical to **After** implementation.

<table>
<tr><th>Before</th><th>After</th></tr>
<tr><td><code>boolean doubleNegationLaw = !(!p))</code></td><td><code>boolean doubleNegationLaw = !(!p)) == p;</code></td></tr>
</table>

Commutative Law

The Commutative law is defined as

$$p \vee q \equiv q \vee p$$

Your task is now to implement the Commutative law given the **Before** implementation, so that is identical to **After** implementation.

<table>
<tr><th>Before</th><th>After</th></tr>
<tr><td><code>boolean commutativeLaw :</code></td><td><code>boolean commutativeLaw = p || q == q || p;</code></td></tr>
</table>

Associative Law

The Associative law is defined as

$$(p \lor q) \lor r \equiv p \lor (q \lor r)$$

Your task is now to implement the Associative law given the **Before** implementation, so that is identical to **After** implementation.

<table>
<tr><th>Before</th><th>After</th></tr>
<tr><td><code>boolean associativeLaw = (p || q) || r == (p || q) || r;</code></td><td><code>boolean associativeLaw = (p || q) || r == p || (q || r);</code></td></tr>
</table>

Distributive law

The Distributive law is defined as

$$p \lor (q \land r) \equiv (p \lor q) \land (p \lor r)$$

Your task is now to implement the Distributive law given the **Before** implementation, so that is identical to **After** implementation.

<table>
<tr><th>Before</th><th>After</th></tr>
<tr><td><code>boolean distributeLaw</code></td><td><code>boolean distributeLaw = p || (q && r) == (p || q) && (p || r);</code></td></tr>
</table>

De Morgan's Law

The De Morgan's law is defined as

$$\neg(p \land q) \equiv \neg p \lor \neg q$$

Your task is now to implement the De Morgan's law given the **Before** implementation, so that is identical to **After** implementation.

<table>
<tr><th>Before</th><th>After</th></tr>
<tr><td><code>boolean deMorgansLaw = p && q == p || q;</code></td><td><code>boolean deMorgansLaw = !(p && q)) == !p || !q;</code></td></tr>
</table>

Absorption Law

The Absorption law is defined as

$$p \lor (p \land q) \equiv p$$

Your task is now to implement the Absorption law given the **Before** implementation, so that is identical to **After** implementation.

<table>
<tr><th>Before</th><th>After</th></tr>
<tr><td><code>boolean absorbtionLaw = r && (q && false) != p;</code></td><td><code>boolean absorbtionLaw = p || (p && q) == p;</code></td></tr>
</table>

Problem 1 - BubbleSort

BubbleSort is a sorting algorithm that works by repeatedly stepping through lists that need to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. This passing procedure is repeated until no swaps are required, indicating that the list is sorted.

Now go to the **Bubblesort** Module in the **problem1** folder. There you should implement the *bubbleSort* function with the bubblesort algorithm.

Before

```
exported testcase testBubbleSort {
  int8[5] input = {1, 4, 5, 2, 3};
  int8[5] output = {1, 2, 3, 4, 5};
  // bubbleSort(input, 5);

  for (int8 i = 0; i < 5; i++) {
    if (input[i] != output[i]) {
      fail(0);
    } if
  } for
} testBubbleSort(test case)
```

After

```
exported void bubbleSort(int8[] list, int8 n) {
  int8 c;
  int8 d;
  int8 t;
  for (c = 0; c < (n - 1); c++) {
    for (d = 0; d < n - c - 1; d++) {
      if (list[d] > list[d + 1]) {
        t = list[d];
        list[d] = list[d + 1];
        list[d + 1] = t;
      } if
    } for
  } for
} bubbleSort (function)

exported testcase testBubbleSort {
  int8[5] input = {1, 4, 5, 2, 3};
  int8[5] output = {1, 2, 3, 4, 5};
  bubbleSort(input, 5);

  for (int8 i = 0; i < 5; i++) {
    if (input[i] != output[i]) {
      fail(0);
    } if
  } for
} testBubbleSort(test case)
```

When you are done run the tests(**Shift + F10**) and make sure that this problem does not fail

Problem 2 - Method Signatures

For this task we will look at a simple library. It contains functions, such as: searching for, reserving and borrowing books.

The function signatures of the library are given in the interface below:

```
exported cs interface library {
  Book searchBook(string title)
  Reservation reserrveBook(Book book, Loaner loaner)
  Receipt cancelReservationion(Book book, Loaner loaner)
  Receipt borrowBook(Book book, Loaner loaner)
  Receipt returnBook(Book book)
}
```

Now go to the **Library** Module in the **problem2** folder. There you should create the interface with the functions signature of all the functions as specified in the code fragment. You can represent the domain object, e.g book, by an empty struct.

When you are confident that you have created the functions and have a program that compiles you can continue to the next problem.

Problem 3 - Refactor

Introduction

This problem is about using refactoring. You decide how the refactoring is carried out. However, all changes you do should be kept in the same module. You do not need to move any code fragment to any other module.

Now go to the next page of this document. There are three subtasks to be solved. When you are done run the tests(**Shift + F10**) and make sure that this problem does not fail.

Subtask 1

At the **OrderProcess** Module in the **problem3** folder, consider the *CheckInventoryAndValidateCustomer* function. It has too much responsibility. Refactor it into two different functions. The *CheckInventoryAndValidateCustomer* should no longer exist when you are done; only *CheckInventory* and *ValidateCustomer* functions.

Be sure to fix calls to this method, so that the program is still valid (no errors.)

Before

```
boolean checkInventoryAndValidateCustomer(Item[] items, Customer customer) {
  boolean isItemInStock = false;
  int8 counter = 0;
  for (int8 i = 0; i < ORDER_SIZE; i++) {
    for (int8 j = 0; j < INVENTORY_SIZE; j++) {
      if (items[i].itemNo == items[j].itemNo) {
        counter++;
      } if
    } for
  } for
  (counter == ORDER_SIZE)?(isItemInStock = true):(isItemInStock = false);

  boolean isCustomerValid = false;
  for (int8 i = 0; i < CUSTOMER_REGISTRY_SIZE; i++) {
    if (customers[i].name == customer.name) {
      if (customers[i].status == GOOD) {
        isCustomerValid = true;
      } if
    } if
  } for
  return isItemInStock && isCustomerValid;
} checkInventoryAndValidateCustomer (function)
```

After

```
boolean checkInventory(Item[] items) {
  boolean isItemInStock = false;
  int8 counter = 0;
  for (int8 i = 0; i < ORDER_SIZE; i++) {
    for (int8 j = 0; j < INVENTORY_SIZE; j++) {
      if (items[i].itemNo == items[j].itemNo) {
        counter++;
      } if
    } for
  } for

  return counter == ORDER_SIZE;
} checkInventory (function)

boolean validateCustomer(Customer customer) {
  boolean isCustomerValid = false;
  for (int8 i = 0; i < CUSTOMER_REGISTRY_SIZE; i++) {
    if (customers[i].name == customer.name) {
      if (customers[i].status == GOOD) {
        isCustomerValid = true;
      } if
    } if
  } for
  return isCustomerValid;
} validateCustomer (function)
```

Subtask 2

Consider the *calculatePriceAndApplyDiscount* function in the module. Do this by applying the following refactorings to the code fragment:

1. Refactor it into to two different functions; one to calculate the price and one to apply discounts.
2. When calculating the price replace the nested conditionals with a sequence of **if** clauses; that is, remove all **else** parts.
3. When applying the discount, inline all the function calls so that the logic is placed inside the **if** statements. Afterwards, you can delete the no longer needed discount functions (*give25PercentDiscount*, *give50PercentDiscount* and *give75PercentDiscount*).
4. Replace all magic number with meaningful constants.

Be sure to fix all errors so that the program is still valid.

Before

```
float calculatePriceAndApplyDiscount(Item[] items) {
  float runningTotal = 0.0;
  for (int8 i = 0; i < ORDER_SIZE; i++) {
    if (items[i].type == AMETHYST) {
      runningTotal += items[i].carat * 12000;
    } else {
      if (items[i].type == DIAMOND) {
        runningTotal += items[i].carat * 16000;
      } else {
        if (items[i].type == EMERALD) {
          runningTotal += items[i].carat * 13000;
        } else {
          if (items[i].type == RUBY) {
            runningTotal += items[i].carat * 13200;
          } else {
            if (items[i].type == SAPPHIRE) {
              runningTotal += items[i].carat * 10100;
            } if
          } if
        } if
      } if
    } if
  } for

  if (runningTotal > 20000) {
    runningTotal = addSeventyFivePercentDiscount(runningTotal);
  } else if (runningTotal > 10000) {
    runningTotal = addFiftyPercentDiscount(runningTotal);
  } else if (runningTotal > 5000) {
    runningTotal = addTwentyFivePercentDiscount(runningTotal);
  }
  return runningTotal;
} calculatePriceAndApplyDiscount (function)
```

After

```
float calculatePrice(Item[] items) {
  float runningtotal = 0.0;
  for (int8 i = 0; i < ORDER_SIZE; i++) {
    if (items[i].type == AMETHYST) {
      runningtotal += items[i].carat * AMETHYST_PRICE;
    } if
    if (items[i].type == DIAMOND) {
      runningtotal += items[i].carat * DIAMOND_PRICE;
    } if
    if (items[i].type == EMERALD) {
      runningtotal += items[i].carat * EMERALD_PRICE;
    } if
    if (items[i].type == RUBY) {
      runningtotal += items[i].carat * RUBY_PRICE;
    } if
    if (items[i].type == SAPPHIRE) {
      runningtotal += items[i].carat * SAPHHIRE_PRICE;
    } if
  } for
  return runningtotal;
} calculatePrice (function)

float applyDiscount(float total) {
  if (total > MAJOR_DISCOUNT_LIMIT) {
    total = total * MAJOR_DISCOUNT;
  } else if (total > MEDIUM_DISCOUNT_LIMIT) {
    total = total * MEDIUM_DISCOUNT;
  } else if (total > MINOR_DISCOUNT_LIMIT) {
    total = total * MINOR_DISCOUNT;
  }
  return total;
} applyDiscount (function)
```

13

Subtask 3

As the last part of the refactoring consider the *processOrder* function in the module.

In this function apply the following refactorings:

1. Surround the checks to the inventory and the customerRegister with an **if** statement. If both checks are true, then the price can be calculated and returned, if not, then you should return **-1**.
2. Inline the *ShippingFee* variable.
3. Extract the *ShippingFee* to a constant.

Before

```
float processOrder(Item[] items, Customer customer) {
  //[ Step 1 ]
  checkInventoryAndValidateCustomer(items, customer);

  //[ Step 2 ]
  float total = calculatePriceAndApplyDiscount(items);

  //[ Step 3 ]
  if (total < FREE_SHIPPING_LIMIT) {
    float shippingFee = 75;
    total = total + shippingFee;
  } if
  return total;
} processOrder (function)
```

After

```
float processOrder(Item[] items, Customer customer) {
  //[ Step 1 ]
  if (checkInventory(items) && validateCustomer(customer)) {
    //[ Step 2 ]
    float total = calculatePrice(items);
    total = applyDiscount(total);

    //[ Step 3 ]
    if (total < FREE_SHIPPING_LIMIT) {
      float shippingFee = SHIPPING_FEE;
      total = total + shippingFee;
    } if
    return total;

  } if
  return -1;
} processOrder (function)
```

14

2.4 Task Descriptions Handout for Group Par

Our students who used the parser-based editor (Eclipse CDT) received the following handout.

Problem Solving

We will now present a set of problems that we ask you to solve.

We have created a project called **TE**(short for *Text Editor*) that you will work in. Inside this we have placed a folder called **problems**. It is subdivided into additional folders; one for each problem that you should attempt to solve.

In the root of the **problems** folder we have also placed a **RunAllTests.c** file. This file contains a test for each problem. The criteria for solving the problems is having all the tests run without errors.

Problem 0 - Logic

Recall the propositional equivalences known as tautologies(perhaps from a Discrete Math course you have previously taken). If you do not recall them they will be presented here and as we go along.

Double negation law	$\neg(\neg p) \equiv p$
Commutative Law	$p \vee q \equiv q \vee p$
Associative Law	$(p \vee q) \vee r \equiv p \vee (q \vee r)$
Distributive law	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
De Morgan's Law	$\neg(p \wedge q) \equiv \neg p \vee \neg q$
Absorption Law	$p \vee (p \wedge q) \equiv p$

Your task is to complete all the laws in the provided code. That is, you should correct the expression so that it is identical to the tautology. For instance, by moving parenthesis, adding an expression or fixing a defect in the statement. We have already implemented the Double negation law to get you started, so you will know how we expect you to solve the task.

Now go to the **Logic.c** source file in the **problem0** folder and start the task presented in the next pages.

When you are done run the test(**Ctrl + F11**) and make sure that this problem does not fail.

Double negation law

The Double negation law is defined as

$$\neg(\neg p) \equiv p$$

Your task is now to implement the Double negation law given the **Before** implementation, so that is identical to **After** implementation.

Before	After
`bool doubleNegationLaw = !(!p)`	`bool doubleNegationLaw = !(!p) == p;`

Commutative Law

The Commutative law is defined as

$$p \vee q \equiv q \vee p$$

Your task is now to implement the Commutative law given the **Before** implementation, so that is identical to **After** implementation.

Before	After
`bool commutativeLaw`	`bool commutativeLaw = p || q == q || p;`

Associative Law

The Associative law is defined as

$$(p \lor q) \lor r \equiv p \lor (q \lor r)$$

Your task is now to implement the Associative law given the **Before** implementation, so that is identical to **After** implementation.

<table>
<tr><th>Before</th><th>After</th></tr>
<tr><td><code>bool associativeLaw = (p || q) || r == (p || q) || r;</code></td><td><code>bool associativeLaw = (p || q) || r == p || (q || r);</code></td></tr>
</table>

Distributive law

The Distributive law is defined as

$$p \lor (q \land r) \equiv (p \lor q) \land (p \lor r)$$

Your task is now to implement the Distributive law given the **Before** implementation, so that is identical to **After** implementation.

<table>
<tr><th>Before</th><th>After</th></tr>
<tr><td><code>bool distributeLaw</code></td><td><code>bool distributeLaw = p || (q && r) == (p || q) && (p || r);</code></td></tr>
</table>

De Morgan's Law

The De Morgan's law is defined as

$$\neg(p \land q) \equiv \neg p \lor \neg q$$

Your task is now to implement the De Morgan's law given the **Before** implementation, so that is identical to **After** implementation.

<table>
<tr><th>Before</th><th>After</th></tr>
<tr><td><code>bool deMorgansLaw = p && q == p || q;</code></td><td><code>bool deMorgansLaw = !(p && q) == !p || !q;</code></td></tr>
</table>

Absorption Law

The Absorption law is defined as

$$p \lor (p \land q) \equiv p$$

Your task is now to implement the Absorption law given the **Before** implementation, so that is identical to **After** implementation.

<table>
<tr><th>Before</th><th>After</th></tr>
<tr><td><code>bool absorbtionLaw = r || (q && false) != p;</code></td><td><code>bool absorbtionLaw = p || (p && q) == p;</code></td></tr>
</table>

Problem 1 - BubbleSort

BubbleSort is a sorting algorithm that works by repeatedly stepping through lists that need to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. This passing procedure is repeated until no swaps are required, indicating that the list is sorted.

Now go to the **BubbleSort.c** source file in the **problem1** folder. There you should implement the *bubbleSort* function with the bubblesort algorithm.

```
void bubbleSort(int list[], int n) {
  int c;
  int d;
  int t;
  for (c = 0; c < (n - 1); c++) {
    for (d = 0; d < n - c - 1; d++) {
      if (list[d] > list[d + 1]) {
        t = list[d];
        list[d] = list[d + 1];
        list[d + 1] = t;
      }
    }
  }
}
```

When you are done run the tests(**Ctrl + F11**) and make sure that this problem does not fail. You have to uncomment the call to the bubblesort function in the test case in order to make it pass the test.

Problem 2 - Method Signatures

For this task we will look at a simple library. It contains functions, such as: searching for, reserving and borrowing books.

The function signatures of the library are given below:

```
Book searchBook(char *title);
Reservation reserveBook(Book book, Loaner loaner);
Receipt cancelReservation(Book book, Loaner loaner);
Receipt borrowBook(Book book, Loaner loaner);
Receipt returnBook(Book book);
```

Now go to the **Library** Module in the **problem2** folder. There you should create the functions signature of all the functions as specified in the code fragment. You can represent the domain object, e.g book, by an empty struct.

When you are confident that you have created the functions and have a program that compiles you can continue to the next problem.

Problem 3 - Refactor

This problem is about using refactoring. You decide how the refactoring is carried out. However, all changes you do should be kept in the same module. You do not need to move any code fragment to any other module.

Now go to the **OrderProcess.c** source file in the **problem3** folder.

You can always run the provided tests(**Ctrl + F11**) and make sure that this problem does not fail.

Subtask 1

Consider the *CheckInventoryAndValidateCustomer* function. It has too much responsibility. Refactor it to two different functions. The *CheckInventoryAndValidateCustomer* should no longer exists when you are done; only *CheckInventory* and *ValidateCustomer* functions.

Be sure to fix calls to this method, so that the program is still valid (no errors.)

Before

```
bool checkInventoryAndValidateCustomer(Item items[], Customer customer) {
    bool isItemInStock = false;

    int counter = 0;
    for (int i = 0; i < ORDER_SIZE; i++) {
        for (int j = 0; j < INVENTORY_SIZE; j++) {
            if (inventory[j].itemNo == items[i].itemNo) {
                counter++;
            }
        }
    }
    isItemInStock = counter == ORDER_SIZE;

    bool isCustomerValid = false;
    for (int i = 0; i < CUSTOMERREGISTER_SIZE; i++) {
        if (customerRegister[i].custNo == customer.custNo) {
            if (customerRegister[i].status == GOOD) {
                isCustomerValid = true;
            }
        }
    }

    return isItemInStock && isCustomerValid;
}
```

After

```
bool checkInventory(Item items[]) {
    bool isItemInStock = false;

    int counter = 0;
    for (int i = 0; i < ORDER_SIZE; i++) {
        for (int j = 0; j < INVENTORY_SIZE; j++) {
            if (inventory[j].itemNo == items[i].itemNo) {
                counter++;
            }
        }
    }
    return isItemInStock = counter == ORDER_SIZE;
}

bool validateCustomer(Customer customer) {
    bool isCustomerValid = false;
    for (int i = 0; i < CUSTOMERREGISTER_SIZE; i++) {
        if (customerRegister[i].custNo == customer.custNo) {
            if (customerRegister[i].status == GOOD) {
                isCustomerValid = true;
            }
        }
    }
    return isCustomerValid;
}
```

Subtask 2

Consider the *calculatePriceAndApplyDiscount* function in the module. Do this by applying the following refactorings to the code fragment:

1. Refactor it into to two different functions; one to calculate the price and one to apply discounts.
2. When calculating the price replace the nested conditionals with a sequence of **if** clauses; in other words, remove all **else** parts.
3. When applying the discount, inline all the function calls so that the logic is placed inside the **if** statements. Afterwards, you can delete the no longer needed discount functions (*give25PercentDiscount*, *give50PercentDiscount* and *give75PercentDiscount*).
4. Replace all magic numbers with meaningful constants.

Be sure to fix all errors so that the program is still valid.

Before

```
float calculatePriceAndApplyDiscounts(Item items[]) {
    float runningTotal = 0.0;

    for (int i = 0; i < ORDER_SIZE; ++i) {
        if (items[i].type == AMETHYST) {
            runningTotal += items[i].carat * 12000;
        } else {
            if (items[i].type == DIAMOND) {
                runningTotal += items[i].carat * 16000;
            } else {
                if (items[i].type == EMERALD) {
                    runningTotal += items[i].carat * 13000;
                } else {
                    if (items[i].type == RUBY) {
                        runningTotal += items[i].carat * 13200;
                    } else {
                        if (items[i].type == SAPPHIRE) {
                            runningTotal += items[i].carat * 10100;
                        }
                    }
                }
            }
        }
    }

    if (runningTotal > 20000) {
        runningTotal = give75PercentDiscount(runningTotal);
    } else if (runningTotal > 10000) {
        runningTotal = give50PercentDiscount(runningTotal);
    } else if (runningTotal > 5000) {
        runningTotal = give25PercentDiscount(runningTotal);
    }

    return runningTotal;
}
```

After

```
float calculatePrice(Item items[]) {
    float runningTotal = 0.0;

    for (int i = 0; i < ORDER_SIZE; ++i) {
        if (items[i].type == AMETHYST) {
            runningTotal += items[i].carat * AMETHYST_PRICE;
        }
        if (items[i].type == DIAMOND) {
            runningTotal += items[i].carat * DIAMOND_PRICE;
        }
        if (items[i].type == EMERALD) {
            runningTotal += items[i].carat * EMERALD_PRICE;
        }
        if (items[i].type == RUBY) {
            runningTotal += items[i].carat * RUBY_PRICE;
        }
        if (items[i].type == SAPPHIRE) {
            runningTotal += items[i].carat * SAPPHIRE_PRICE;
        }
    }
    return runningTotal;
}

float applyDiscount(float runningTotal) {
    if (runningTotal > MAJOR_DISCOUNT_LIMIT) {
        runningTotal = runningTotal * MAJOR_DISCOUNT;
    } else if (runningTotal > MEDIUM_DISCOUNT_LIMIT) {
        runningTotal = runningTotal * MEDIUM_DISCOUNT;
    } else if (runningTotal > MINOR_DISCOUNT_LIMIT) {
        runningTotal = runningTotal * MINOR_DISCOUNT;
    }
    return runningTotal;
}
```

Subtask 3

As the last part of the refactoring consider the *processOrder* function in the module.

In this function apply the following refactorings:

1. Surround the checks to the inventory and the customerRegister with an **if** statement. If both checks are true, then the price can be calculated and returned, if not, then you should return **-1**.
2. Inline the *ShippingFee* variable.
3. Extract the *ShippingFee* to a constant.

Before

```
float processOrder(Item items[], Customer customer) {
    //1. step - validate
    checkInventoryAndValidateCustomer(items, customer);

    //2. step - calculate Price and discount
    float total = calculatePriceAndApplyDiscounts(items);

    //3. step - apply shipping charge
    if (total < FREE_SHIPPING_LIMIT) {
        float shippingFee = 70;
        total = total + shippingFee;
    }
    return total;
}
```

After

```
float processOrder(Item items[], Customer customer) {
    //1. step - validate
    if (checkInventory(items) && validateCustomer(customer)) {

        //2. step - calculate Price and discount
        float total = calculatePrice(items);
        total = applyDiscount(total);

        //3. step - apply shipping charge
        if (total < FREE_SHIPPING_LIMIT) {
            total = total + SHIPPING_FEE;
        }
        return total;
    }
    return -1;
}
```

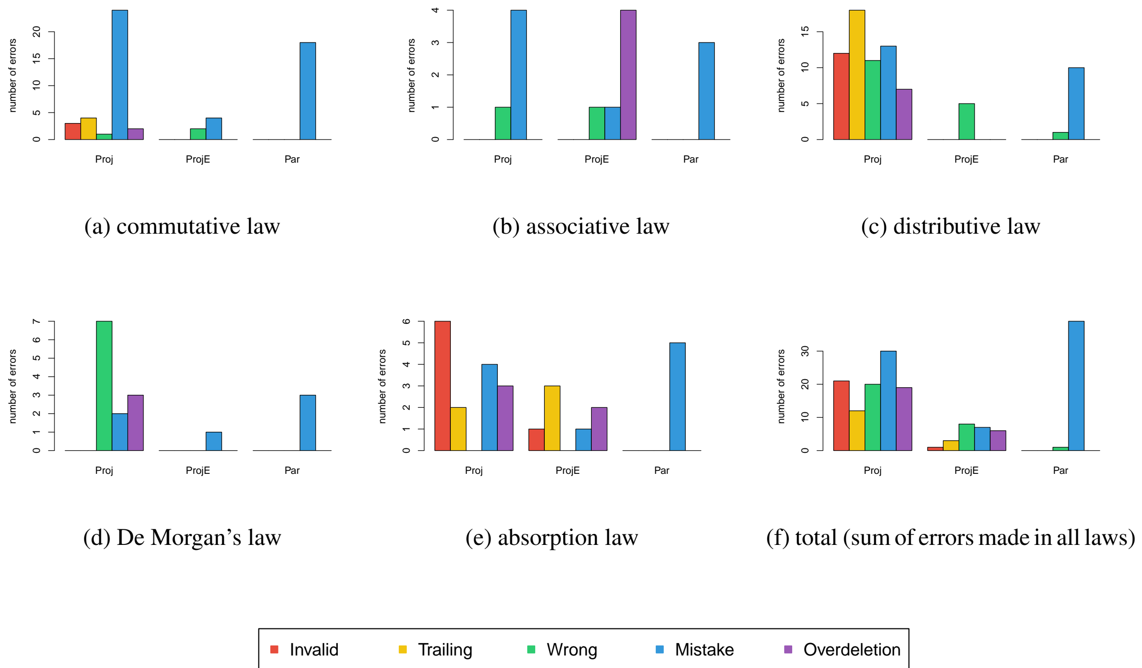
3 Detailed Additional Statistics

In addition to the statistics and diagrams provided in the paper, the following information might be useful, comprising a more exact breakdown of errors (per sub-task) for the first two tasks, and a more exact breakdown of modification and refactoring operations (per operation) in Task 4.

3.1 Basic Editing and Errors

3.1.1 Editing Errors

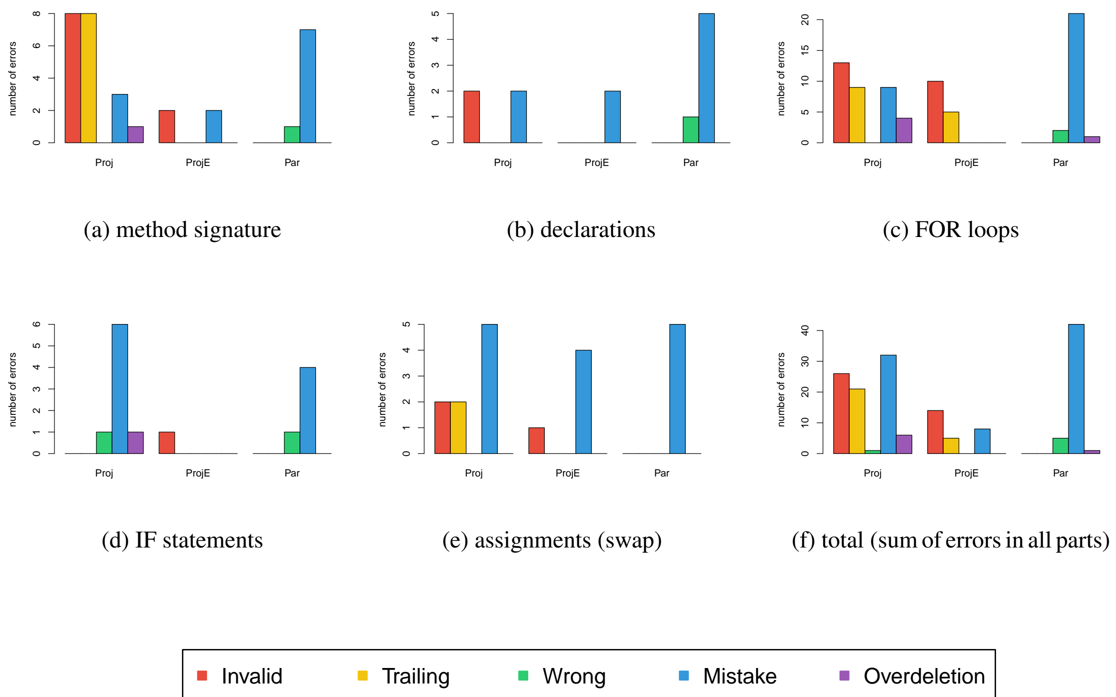
In addition to the relative numbers of errors made in the basic-editing tasks, which are shown in the paper, we provide the absolute numbers in Fig. 5 and Fig. 6.



(a) commutative law (b) associative law (c) distributive law

(d) De Morgan's law (e) absorption law (f) total (sum of errors made in all laws)

Figure 5: Absolute numbers of errors made in task 1 (edit expressions)



(a) method signature (b) declarations (c) FOR loops

(d) IF statements (e) assignments (swap) (f) total (sum of errors in all parts)

Figure 6: Absolute numbers of errors made in task 2 (bubblesort algorithm)

3.2 Modification and Refactoring Operations

Recall that we count, for each group and each of the three subtasks, how many participants have used a specific operation. This measurement simplified the counting effort, since we did not need to count all usages, but rather see wether one participant used it in a subtask, and if so, we stopped counting this participants usage of the operation in the specific subtask. In other words, we “observed” whether an operation was used, and then we aggregate the number of “observations” across all subtasks and participants.

Table 1: *Move* operation

refactoring subtask	Proj	ProjE	Par
subtask 1	2	3	0
subtask 2	4	4	0
subtask 3	1	2	1
total relative to all “observations”	35%	60%	6%

Table 2: *CopyPaste* operation

refactoring subtask	Proj	ProjE	Par
subtask 1	1	0	3
subtask 2	4	1	5
subtask 3	0	2	6
total relative to all “observations”	25%	20%	78%

Table 3: *CutPaste* operation

refactoring subtask	Proj	ProjE	Par
subtask 1	7	3	4
subtask 2	3	3	3
subtask 3	5	2	3
total relative to all “observations”	75%	53%	56%

Table 4: *ManualInsertion* operation

refactoring subtask	Proj	ProjE	Par
subtask 1	0	0	0
subtask 2	4	2	5
subtask 3	5	3	5
total relative to all “observations”	45%	33%	56%

Table 5: *RefactoringAction* operation

refactoring subtask	Proj	ProjE	Par
subtask 1	1	0	1
subtask 2	5	5	1
subtask 3	0	5	0
total relative to all “observations”	30%	53%	11%

Table 6: *SurroundWithAction* operation

Refactoring Sub-task	Proj	ProjE	Par
Sub-task 1	0	0	0
Sub-task 2	0	0	0
Sub-task 3	0	3	0
Total relative to observations	0%	20%	0%

References

[1] R. Likert, “A technique for the measurement of attitudes,” *Archives of Psychology*, 1931.

[2] A. Strauss and J. Corbin, “Grounded Theory Methodology,” in *Handbook of Qualitative Research*, 1994, pp. 273–285.

[3] R. Goede and C. de Villiers, “The Applicability of Grounded Theory As Research Methodology in Studies on the Use of Methodologies in IS Practices,” in *Proc. SAICSIT*, 2003.

[4] A. Strauss and J. Corbin, “Open Coding,” *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*, vol. 2, pp. 101–121, 1990.

[5] M. Voelter, J. Siegmund, T. Berger, and B. Kolb, “Towards User-Friendly Projectional Editors,” in *Proc. SLE*, 2014.