1-1-2011

An experience report of the solo iterative process

Christopher Dorman
Wayne State University,

AN EXPERIENCE REPORT OF THE SOLO ITERATIVE PROCESS

by

CHRISTOPHER DORMAN

THESIS

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

2011

MAJOR: COMPUTER SCIENCE

Approved by:

____

Advisor Date

DEDICATION

To Andrea

ACKNOWLEDGEMENTS

I would like to thank Prof. Rajlich for his help and inspiration. His belief in this work is the motivation I required to finish. Laurentiu Radu Vanciu was also instrumental. He contributed major parts to this thesis, including selecting the project, many of the tools, proofreading and much more.

Atlassian Pty Ltd. supplied a complimentary copy of their Clover Java code coverage & test optimization for use in this project. This made the testing coverage numbers possible. Tasktop Technologies provided a complimentary copy of their Tasktop plugin for Mylyn and Eclipse. Mylyn and Tasktop together made it easy and unobtrusive to time the phases of this project, even correcting my mistakes at time.

JRipples made by a former Wayne State SEVERE group member Maksym Petrenko is a tool that was invaluable for this project. I'm still not sure how I could perform impact analysis without it.

Finally, I would like to thank the muCommander community for supplying the program for this project. Finding suitable open source projects for university projects is more difficult than one might think. The muCommander program is a very good candidate with easy to explain functionality and well maintained code.

This work was supported in part by grant from the National Science Foundation CCF-0820133. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1
Introduction

The field of software engineering is over 50 years old; in his in press manuscript, Rajlich gives a brief history [1]. Originally, mathematicians and engineers thought software development was more of an art form than a defined process. These first software engineers managed to produce a variety of complex, working software.

1.1 Waterfall Model

As time went on software engineers came to a point where it was necessary to move to a defined process modeled after processes in other engineering disciplines known today as the *waterfall* model. This model had four stages:

1. *Requirements*
2. *Design*
3. *Implementation*
4. *Maintenance*

In the waterfall model each stage must be completed before the next stage is started. To begin, the software engineers would collect requirements from the stakeholders. Then they would use the gathered requirements to design the entire system. Once they completed the design they would implement the program and release it to the users. When the users reported problems, the problems would be fixed during maintenance.

This model ran into significant complications because the requirements of software are volatile. In large programs, the requirements often change so drastically while the software engineers are performing the first three steps that programs delivered are completely different from the stakeholders' current requirements. This problem with the waterfall model was famously described by Brooks [2].

1.2 Agile Manifesto

Since Brooks published his book in 1975 software engineers developed new processes of software development. In 2001 a group of software engineers drafted the *Agile Manifesto* [3] that summarizes the foundations of these new processes:

> "We are uncovering better ways of developing software by doing it and helping others do it. We value:
>
> - Individuals and interactions over processes and tools.
> - Working software over comprehensive documentation.
> - Customer collaboration over contract negotiation.
> - Responding to change over following a plan."(p. 2)

The principles of the agile manifesto do not declare that processes, documentation or any other workproduct is unimportant, but rather just a reminder that the most import workproduct is the program along with the people who write it. The agile manifesto is popular, it has over 10 thousand signatories [4]. Many processes include the agile principles and research shows them to be successful; a selection is discussed in more depth in Chapter 2. Agile principles have become so widespread that processes in other engineering disciplines have defined their own, such as the *Integrated Project Delivery* for the construction industry [5].

1.3 Solo Iterative Process Experience Report

This thesis is an experience report of the *Solo Iterative Process (SIP)* as defined by Rajlich [1]. SIP describes a process of a programmer working alone on a software project and it belongs to the group of iterative evolutionary processes. It shares many characteristics with team iterative processes including repeated software change (SC), baseline build, elicitation and analysis of requirements for the product backlog, and so forth.

This thesis describes an implementation of a new feature by enacting SIP on a medium sized open source program. The feature is implemented in an iteration that consists of several software changes, each adding new functionality or fixing a bug. It also draws on the programmer's experience to present lessons learned about of the individual phases of SC after performing multiple changes.

Chapter 2 surveys the previous work and Chapter 3 describes the SIP process model. Chapter 4 describes the subject program, technologies involved, and a high level description of the feature to be implemented. Chapter 5 contains the description of the SIP enactment that implements the new feature. Chapter 6 contains the measurements and discussion of the experience and Chapter 7 contains conclusions and future work.

Chapter 2
Previous Work

Many different software processes are in use. Much research has been done and continues on these processes, their tasks and the tools used to implement them. This chapter details a current state of the art selection of these processes, tasks and tools.

2.1 Software Processes

The field of software engineering defines software processes for programmers to use to produce high quality programs. Research has defined many software processes and gathered data to show that these processes help programmers produce the intended high quality programs. This section briefly looks at why agile methods of software evolution are used; then looks more in depth at 2 solo processes and an assortment of team software processes based on software evolution.

2.1.1 Software Evolution

Even with the amount of research and industrial use of software evolution, there are still software engineers who use other methods of software development and question the need for software evolution. This is addressed by Lehman [6], who draws from personal experience and the wealth of research done on software evolution to argue software evolution is currently the most effective approach to develop software. He provides examples of different types of software that benefit from software evolution, but also presents a general argument that software evolution is necessary because the domain of software itself evolves, also called the volatility of requirements.

2.1.2 Solo Software Change

There are many well defined team based software evolution processes; however, a solo programmer can also use a process. Previous work in software processes for a single programmer has successfully show a solo programmer can produce high quality software; it includes work by Febbraro and Rajlich [7]. They did an initial design of a simple point of sale program and then used SC to add functionality. The results were compared to a version of the program created through object-oriented design and they conclude that SC produces a simpler design. They also discuss the important role of refactoring in SC. The point of sale program was made using the SC process presented by Rajlich and Gosavi [8]. They identify the best practices in a how to process for changing object-oriented software. It starts by identifying the concepts of the change, identifying the software modules to change, then preparing, changing and cleaning up the code after the change through refactoring. It also includes verifying the software during the change.

2.1.3 Personal Software Process

Another software process for a solo programmer is the Personal Software Process (PSP) [9]. This process builds on a programmer's preexisting abilities and is intended to prepare them for a team process. It is taught through a series of ten programming tasks, where the student keeps track of a battery of metrics [10]. During each task they learn from their mistakes to create higher quality software more efficiently. Various studies have shown PSP to improve performance in both university and industrial settings, such as one by Ferguson, Humphrey, Khajenoori, Macke and Matvya [11]. However, the metrics used many PSP case studies are mainly the data collected by the users of PSP. Johnson and Disney believe the PSP data is error prone and outside metrics would be a better indicator [12]. They do admit that outside metrics are difficult to obtain, even when simple, such as cost-effectiveness. Additionally, even after calling into question the data showing the effectiveness of PSP, they still believe in it, "… both of us consider it to be one of the most powerful software engineering practices we have adopted in our careers."(p. 343) Although, they rely on the data they believe erroneous and anecdotal evidence to support their opinions.

2.1.4 Team Software Processes

There are many team software processes; many of the challenges faced by a solo programmer are also faced by teams of programmers. The volatility of requirements is one notable shared challenge, where the team tasks may be applicable to a solo process. This section will look at a selection of team processes and their view on dealing with the challenges of software engineering.

One team software process is SCRUM as defined by Schwaber [13]. It accounts for difficulties of industrial software production; some of these are realities of any business, such as time pressure and competition, while others are more specific to software, such as the volatility of technology and how it reduces the availability of programmers. It has flexibility built in with the intent to allow programmers to account for the volatility of software development; planning is only done for short periods of time, known as sprints. At the end of a sprint the current state of the project is reassessed before the next sprint. Rising and Janoff [14] explain how SCRUM is suited to small teams of programmers. They present a picture of chaos for software development in small teams, because of requirement's volatility. They continue that small teams can limit the chaos by using SCRUM and support their contention with experience reports using SCRUM.

Test-Driven Development as presented by Martin [15] is an agile process that is based on writing tests, then production code that passes the tests. He lists the processes three laws:

- “You may not write production code unless you’ve first written a failing unit test.
- You may not write more of a unit test than is sufficient to fail.
- You may not write more production code than is sufficient to make the failing unit test pass.” (p. 32)

Although he admits the laws are more of guidelines, he does argue the tenets produce a structurally different code that is superior to code produced using other software processes. This is because the code will be error free, free of bloat and deadlines will be met. He also argues another advantage is that by definition, there will be a comprehensive regression test suite that will encourage refactoring.

Extreme Programming (XP) is another agile process that has a defined set of practices the agile team follows. Müller and Tichy study issues with a subset of the practices while introducing it to programmers who are accustomed to using other processes [16]. They find that some of the practices such as writing tests before writing production code and only designing a small part of a program at a time are difficult for some programmers to accept. Furthermore, while the programmers enjoy pair programming and believe it produces high quality code, both the programmers and authors are unsure of its value, especially when writing simple code. They conclude that its implementation requires the team to be tightly managed and there will be difficulty scaling XP to large teams.

Cockburn and Highsmith claim that the common factor in agile processes is the quality of the people implementing the process [17]. They present the argument that, "'people trump process'"(p. 131) in many of the common agile processes such as XP, SCRUM and others. The one factor they consider to be able to overtake quality people is organizational politics.

2.2 Software Tasks

Solo and team software processes are composed of tasks that programmers perform to write programs. Besides software process granularity, previous research in software evolution has also studies on the individual phases and tasks. Much of the research into this area explains a method any programmer can use to complete a task. This section looks at some of these tasks.

2.2.1 Concept Location & Impact Analysis

Concept location techniques in object-oriented software is studied by Marcus, Rajlich, Buchta, Petrenko and Sergeye [18]. They start by explaining a method to bridge the relationship between human concepts and code concepts then explain three concept location techniques for object-oriented code: text based searching (grep), dependency search and information retrieval techniques (IR). They give examples of how and when to use each technique to show some advantages and disadvantages of each, especially in respect to code concepts that are explicit and implicit.

Concept location was also studied by Chen and Rajlich [19]. They look in depth at dependency search and its requirements. The requirements focus is on what would be required for an automated tool to assist with concept location. They define a graph to with edges made up of function calls and data flows specifically for this purpose.

Ren, Chesley and Ryder look at impact analysis by presenting 2 tools that work together to find the impact of a SC [20]. They conclude their tool is effective because it is able to find the reason why the majority of regression tests fail after changing code they are unfamiliar with. Additional research into impact analysis and change propagation by Han [21] looked at how both could be expanded beyond software maintenance tasks and also be used during software design. This appears to be a precursor step in the acceptance of software evolution techniques. He also performs impact analysis and change propagation directly on the code.

2.2.2 Refactoring

Refactoring is well defined by Fowler [22], who explained basic refactorings such as extract class, inline class, move field and others. Refactoring is also regularly updated by Fowler and the software community through his website [23]; it has over 90 examples of refactoring currently. Mens and Tourwé [24] outline a process to that list steps the programmer should take for a successful refactoring. This provides programmers process for successful refactoring and includes the concept that the programmer should include all the artifacts in a refactoring.

2.3 Software Process Tools

The research into software evolution has not been restricted to abstract processes and tasks; but has also implemented and studied concrete tools to assist programmers with the processes and tasks. This section looks at one tool particularly suited to the SC process (section 2.1.2) and some well accepted software evolution tools.

2.3.1 JRipples

Buckner, Buchta, Petrenko and Rajlich present a tool to assist with the tasks of concept location, impact analysis and actualization during SC [25]. The tool provides different methods for concept analysis, such as grep and dependency search. It also identifies dependencies in a program and tracks a programmer's visits to them to assist with impact analysis and change propagation. The authors claim an automated tool is better at these tasks and frees the programmer to do steps better suited to humans.

2.3.2 Other Software Tools

Other tools that assist with the tasks of SC are JUnit presented by Gamma and Beck to assist with verification [26]. Another tool for verification is Abbot that adds functional test for GUI components to JUnit [27]. To assist the programmer with measuring verification coverage, Yang, Li and Weiss review a variety of different tools and conclude none of the coverage tools is superior to all others; a coverage tool should be selected based on the program and project [28].

Chapter 3
Solo Iterative Process

Agile methods of software evolution focus on programmers talents to produce quality software [17]. This experience report used one such process, the *Solo Iterative Process* (SIP) [1]. It is a process that a single programmer can use to create high quality software and meet time and resource constrains. SIP helps a solo programmer with technical goals, such as meeting the stakeholders' requirements and the business aspects such as paying bills. The term *iterative* in SIP is important to an agile method; it means that this is a process that is repeated to obtain a finished product. An iterative process is important so that it can adjust for the reality of volatility in software development.

At the core of SIP is the task of SC, which has been successfully used in research and university classrooms [29]. However, SIP is more than exclusively the task of changing software; it includes the following tasks and workproducts necessary for a programmer to meet the responsibilities of software engineering:

1. *Product Backlog* – add, organize and choose a user stories to implement
2. *Software Change* – implement a change request
3. *Iteration/release* – a special commit that can be distributed to users
4. *Measuring SIP* – logs the programmer keeps

SIP assists with planning by recording time spent of each task and using it to estimate future effort. This allows the programmer to use resources more wisely, especially his most important resource, time. If the programmer does not keep track of his time, it will be difficult for him to estimate the effort required for future projects, if a programmer cannot estimate time accurately, it will be challenging to meet users expectations and consequently to pay bills.

3.1 Product Backlog

The Product Backlog is a collection of *user stories* that need to be added to the software through change requests. User stories are simple explanations of a change a stakeholder would like implemented in the code. They are added to the backlog by any of the project's stakeholders, such as users and the programmer. This is the only task of SIP that includes stakeholders besides the programmer.

Four types of change requests are made from the user stories; they are categorized by their purpose. If a user asks for a bug in the program to be fixed it is a *corrective* change request. If the request is to add new functionality it is a *perfective* change request. If the programmer adds a change request to make the source code easier to change in the future it is a *protective* change request. If a change request asks for the software to be compatible with a version of a technology it is an *adaptive* change request.

The user stories are entered into a spreadsheet to limit the scope of change requests created from them and it also allows them to be prioritized by the programmer whenever necessary. Other mediums such as 3”x5” card can also be used to manage the user stories in the product backlog. Many different criteria can be used to prioritize the product backlog. To help keep it organized a programmer needs to have different levels of priority. Four levels of priority (1 for high priority, 4 for low priority) [1] help the programmer to quickly identify which user stories need to be addressed soon and which ones can be handled at a later date. While all user stories use the same priority levels, different descriptions are used to help the programmer properly categorize the user stories. For perfective change requests, the descriptions are based on the business value:

"1. An essential functionality without which the application is useless

2. An important functionality that users rely on

3. A functionality that users need but can be without

4. A minor enhancement" (chp. 5)

However, for corrective and adaptive change requests, the descriptions are based on severity:

"1. Fatal application error

2. Application is severely impaired (no workaround can be found)

3. Some functionality is impaired (but workaround can be found)

4. Minor problem not involving primary functionality" (chp. 5)

For protective change request, the descriptions are based on the threat:

"1. A serious threat, the so-called "showstoppers"; if unresolved, the project is in serious trouble

2. An important threat that cannot be ignored

3. A distant threat that still merits attention

4. A minor inconvenience" (chp. 5)

These priorities help a programmer to prioritize the product backlog, however, they are recommendations; not all priority 1 change requests will be done before priority 2 change requests. The programmer will use other factors to decide the actual order of the backlog. For example, the programmer may choose a priority 3 change request over a priority 2, if it requires significantly less time to implement. Likewise if users communicate dissatisfaction because of bugs, the program will choose to move corrective change requests forward in the backlog and other categories back. The product backlog is reshuffled in this manner as often as the volatility of the requirements demand.

3.1.1 Iteration Backlog

The iteration backlog is a subset of change requests of the product backlog. The programmer chooses the iteration backlog at the start of an iteration of SIP, once the iteration backlog is chosen and the iteration starts, no additions can be made to the iteration backlog. The goal of the iteration is to complete the tasks in iteration backlog, by performing the steps of SC on each change request in a pre-chosen amount of time. However, if setbacks occur, the SIP programmer can extend the time of an iteration or leave some change requests unfinished and return them to the product backlog. The SIP programmer will evaluate the length of time available then select a set of change requests he considers he can complete in the time frame The programmer needs to limit the size of the iteration, because the longer the iteration the more the volatility of requirements will set in, which means the more likely the programmer’s decisions will be off the mark.

3.2 Software Change (SC)

This section is a summary of the model of software change (SC) presented by Rajlich and Gosavi [8]. SC is the task inside the SIP process when the programmer changes the source code; it is repeated for change requests in the iteration backlog. The phases of SC along with a brief description are:

1. *Initialization* – chose a change request to implement in the code
2. *Concept Location* – find the place in the code that the ideas of the change request are implemented
3. *Impact Analysis* – examine the code neighboring the concept location to determine if it needs to be changed also
4. *Prefactoring* – prepare the code to make the change easier
5. *Actualization* – implement the change in the code
6. *Postfactoring* – rework the code to make future changes easier
7. *Verification* – confirm that the code is of high quality
8. *Conclusion* – commit updated code to the repository

The phases should be done in order with the exception of verification, which is done in concurrence with prefactoring, actualization and postfactoring. Also, the phases are a guideline for each change; individual phases such as concept location when the programmer is familiar with the location of concept extension or postfactoring during a trivial change request may be skipped if the programmer determines it is not necessary. The following sub sections describe each of these phases in more detail.

3.2.1 Initialization

Initialization is the start of a change request in SC. Since the SIP programmer already selected the iteration backlog, initialization is simply choosing one of the user stories from the iteration backlog to be implemented. However, some user stories may be too large to implement in one change request; in these cases the SIP programmer divides the change request into multiple change requests. Each of these change requests implement part of the functionality, for example, a change request could be divided into three change requests, one for the GUI, one to check the input and one with an algorithm that processes the data. The programmer then chooses to perform the GUI change request first and update the code by committing it to the repository. This helps the user to stay organized and measure progress.

3.2.2 Concept Location

Concept location begins with the programmer reading the change request and separating out the concepts that need to be found in the code, which is called extraction of significant concepts. For example, a program that explorers an operating system's file system receives the change request, "Add a basic search function. The search should allow a user to search in the current directory for all or part of the title of a folder or file and return a list of the matching files and directories." The relevant concepts are:

- search
- current directory
- search term
- matching files and directories

Words such as "add" and "should" are instructions to the programmer and are discarded. The programmer then determines if the concepts are likely to appear directly in the code, which is an explicit concept and often easier to find. For example, "current directory" is a concept that is likely to appear directly in the code and is therefore, an *explicit* concept. A concept that is unlikely to appear directly in the code is an *implicit* concept and generally more difficult to find. An example is "search", since the change request requests search functionality added to the program, it is unlikely that the code contains search directly.

The programmer also adds intensions or synonyms and connotations of the concept. In the change request the programmer adds a simple synonym of directory, folder and determines “matching files and directories” includes the file or directory’s name. Intensions can be very complicated, in Linux the data structure used to store directory information is called an “inode” [30], another possibility might be to group directories with other files, such as archive files and call the group “browsable”.

One technique used to find an intension in source code is to do a simple text search. This is commonly known as “grep”, from the UNIX search, but modern development tools have many different variations. In the example above, the programmer might choose to search for “directory” or “folder” at the same time. If the search returns a reasonable number of results, the programmer will visit the classes to determine if they contain the concept extension. If the programmer cannot find the concept extension, the added knowledge obtained from unsuccessful searches helps him create new searches. If the search returns no results or too many results for the programmer to visit, he can revise his search to include more terms, fewer terms or combinations of terms. These grep searches are not always successful, if the programmer is unfamiliar with the code, he may not be able to guess the intensions of the extensions implemented in the code.

Another concept location technique is called a dependency search. The programmer begins the search in top level class, in many programming languages the class with the *main()* method. The programmer then visits the classes that handle parts of the top level class’s responsibilities, known as *suppliers* and if necessary the programmer visits the suppliers of the suppliers recursively until the concept extension is located. If the programmer takes the wrong path, he backtracks to a higher level class and takes a new path to find the concept extension.

The programmer chooses the appropriate search strategy based on knowledge of the source code. If the programmer has very limited knowledge at the outset of concept location, he may start with a grep search. If he gains the knowledge that the code has poorly named identifiers, he may decide to switch to a dependency search. Likewise, he may use a combination of strategies, such as visiting a class that is a grep search result, then switch to a dependency search and visit its suppliers to locate the concept extension. Ultimately, the programmer creates the *initial impact set,* which contains all the classes with a concept extension.

3.2.3 Impact Analysis

After the programmer locates the main concepts in the code, he needs to account for the effect of changing the classes of the initial impact set. The programmer does this by visiting the classes that have *dependencies* of the classes in the initial impact set, if these classes also need changes; they are added to the *estimated impact set*. Dependencies are relationships where one class allows another class to handle some of its responsibility. If a class handles a responsibility for another class, it is a supplier, which was previously defined (section 3.2.2) and if the class depends on a class for part of its responsibility it is called a *client*. There can be a class that is not impacted by the change request, but communicates between 2 classes that are dependent on each other. These intermediary classes *propagate* the dependency and are not added to the estimated impact set. However, the classes that have dependencies with the propagating class should be visited to ensure they are not also impacted.

A simple example of impact analysis is a change request that requires a method's return type to change from a type of `int` to a type of `long`. The programmer must visit all the classes that include a call to this method because they are clients of the method. The programmer then must determine if these classes must be changed to match the new method return type. If the method is the parameter for an overridden method that also has a version that accepts a `long`, such as the Java `System.out.print()` method, the class is not added to the estimated impact set. However, if the client stores the impacted method's return value in a field of type `int`, the client field's type also needs change to a type `long` and the class is added to the estimated impact set.

3.2.4 Prefactoring

Prefactoring is *refactoring* done mainly to make it easier to actualize a change. Refactoring is rewriting source code without changing its functionality, such as dividing a large class into 2 classes by extracting a class. An example of prefactoring is extracting a super class from a class. The programmer can then actualize the change by incorporating another class that inherits from the base class. This way the functionality in the super class does not have to be duplicated and classes are not impacted when they switch between the implementations of the super class using polymorphism.

3.2.5 Actualization

Actualization is the procedure of changing the existing code or adding new classes to add new functionality. The programmer changes the code of the classes in the estimated impact set and adds new classes to the code if necessary. The programmer may realize that some classes were missed during impact analysis and need to be changed or that they do not actually need to be modified. The classes that are changed during actualization or prefactoring are the *changed set.*

Actualization can be as simple as modifying a single line of code (LOC) or as complex as changing and adding large numbers of classes. An example of a small change is fixing a bug by changing the limit condition of a loop to prevent an array out of bounds condition. This is a very simple actualization, but it is the entire actualization of a corrective change request.

Larger changes require new classes to be incorporated into the code. The classes may be incorporated through different techniques, four used in this experience report are: polymorphism, replacement, as a new supplier or as a new component. Polymorphism can be the easiest method; the programmer creates a new class that inherits from a super class. This is easy because classes that are clients of the super class can use the new class without being impacted.

Replacement is used when a basic class is removed from the code and a more complex class is put in its place. An example of replacement is replacing a class that finds words in a text document with one that not only finds the exact word, but also synonyms of the word. The basic class just did a simple text match; while the new class needs to access a database to get synonyms and then it must find any of the words from the set of synonyms. The new class is much more complex; it requires much more than just changing or adding a few methods and is therefore done by writing a new class and then replacing the basic class.

Incorporation of a new supplier is used to expand existing functionality. A new class is added to the source code and an object of it is added to an existing class. The new supplier takes on responsibility for the existing class. One example of incorporation of a supplier is a change request to add persistent data storage; a new supplier is added to store the existing data in a database, text file or other technology.

Incorporation of a component is similar to replacement, except that nothing is removed. This is generally done when new functionality is added. An example of incorporation of a component is a class that saves the history of user input. Before the incorporation of the component, the source code takes user input from a supplier class and performs a task with it and sends it to a client. The new component class will also get the user input from the supplier class, store it and provide it to the same client as the other component upon request.

3.2.6 Postfactoring

Postfactoring is refactoring done after actualization and is very similar to prefactoring. The difference is that it does not add value to the current change request; rather its purpose is to make future changes easier in general. Some programmers may not see the value in postfactoring, but it is important. It is an investment in the code; without it code decay can become very severe making future change requests difficult if not impossible.

A simple but effective example of postfactoring is changing the name of an identifier. For instance, a programmer may use the name `i` for an iterator in a loop that iterates through the rows of table. If the programmer changes the name `i` to `row` it will be easier during future change requests for programmers to know what the loop does. Individually, small changes like this may not seem significant but collectively they can make change requests significantly easier.

3.2.7 Verification

Verification is different from the other phases of SC because it is integrated with the phases of prefactoring, actualization and postfactoring. Its purpose is to reassure the stakeholders that the code meets the requirements placed upon it and is of high quality. However, because of the essential difficulties of software, no amount of verification can guarantee its quality. Some may consider it a synonym for the various forms of testing, such as unit and functional, but it also includes other types such as code inspections.

Unit tests are named such because they each test one unit of the source code. One unit may be a single method; however, it can be larger, if a method has suppliers the unit could be the method and its suppliers. Unit testing is white box testing meaning that the programmer can see the source code when writing and running test. A test can test multiple conditions of a unit of code or can have multiple tests directed at it, for example, a programmer could write 2 tests for the following method:

```
public void addToList(String stringToAdd){
     if(stringToAdd == null)
          throw new NullPointerException();

     listOfStrings.add(stringToAdd);
}
```

One test calls the method with a `null` value and one with a `String` value or both conditions could be in a single test. Multiple tests are preferable because it makes the test's goal very clear; if a test fails, it is very easy for the programmer to identify the reason often just by the name of the test.

Another type of verification is functional testing. It tests the functionality of a program; it is not concerned with the structure of the code, but rather if it performs as desired. Functional testing can be either white box testing, like unit testing or black box testing, where the programmer does not have access to the code. It is especially useful to test GUI components that require user input.

Verification can also include code inspection. It is not an automated test like unit or functional tests; but rather is the programmer reading the code. It has advantages over automated test, because programmers are inclined to see a bug that is dependent on a particular value, such as a divide by zero condition. Automated tests are written to test a set of values, if the set does not include the value that creates the defective condition, the automated test will not detect the bug. However, programmers are prone to miss errors such as misspellings that automated test can easily detect. Therefore, a comprehensive verification plan will include multiple types of verification.

The code implementing the tests and only code that is only necessary to support the tests is known as *harness code*. While the code tested that implements the features of the program is *production code*. Whatever types of tests the programmer chooses it is important that a large percentage of the production code is verified. The metric of verification is called *coverage*. Test coverage can be measured in many different granularities; one is the statement level. In the unit test example method, there are three statements, one on each LOC inside the method. However, in general, not every LOC is a statement. Statements are executable LOC, such as ifs, switches and returns. Variable declarations, package imports and such are not statements. A comprehensive verification strategy includes unit tests that execute a high percentage of statements. However, even if every statement is covered, bugs can still be present. There are multiple reasons for this, some rooted in the core principles of computer science, such as the halting problem, but in other cases the code may be correct, the bug is because the programmer did not understand the requirements of the user. Additionally, obtaining complete statement coverage can be very time consuming for some code, such as exception handling. In this case the programmer's time is better spent on other tasks. SC does not define a level of code coverage; the stakeholders must determine the proper level of coverage to make good use of resources and meet their quality requirements.

3.2.8 Conclusion

The phase of conclusion ends each SC. The programmer updates the source code in the repository with the changed code files. This saves the change as part of the code base and incorporates it into the code.

3.3 SIP Workproducts

The programmer produces specific workproducts to keep track of his progress. They provide an outline of SIP programmer's activities, so that he can make decisions that use his resources more effectively.

3.3.1 Basline

A baseline is a special code update that is well verified and does not contain any partially implemented functionality; therefore it is a good point to return to if a defect is found later. However, not all change requests leave the code in a good state for a baseline. For example, if a GUI is implemented during a change request, but requires more change requests to complete its functionality, the other 2 change requests would need to be redone. Therefore, the programmer would wait until the functionality is completed to create the baseline. At that point, the program is stable and no change requests would need to be redone if the programmer returned to it because of partial functionality.

A SIP programmer does not need to worry about conflicts with other programmers because he is working alone. However, baselines are still important; because the code is not seen by other programmers a SIP programmer is especially prone to habitualization or seeing an erroneous code as correct. The more often baselines are made the less work the programmer will lose, if it is necessary to return to a previous baseline.

3.3.2 Iteration/release

The iteration and release phase of SIP is a special baseline. It marks the end of an iteration of the SIP process. The iteration ends either because the programmer completes all of the change requests in the iteration backlog or because the programmer decides to end the iteration before the iteration backlog is empty. At the end of an iteration the source code should be in a complete and high quality state, but the programmer still must decide whether or not to release the program to the users or to do more iterations. The programmer makes this decision mainly based on the current business environment. If the SIP programmer believes the program is ready to be released to users, he will release it. However, if a competitor has released a program with functionality that the current iteration cannot compete with, the programmer will choose to wait for a subsequent iteration to release. Additionally, other business realities may override technical issues; if the programmer is running low on resources, he may choose to release it. In either case the next step is to return to the product backlog and start the next iteration.

3.3.3 Time Log

The most important one is a time log, which is a record of the amount of time the programmer spends on each task. For tasks that include changing the code the programmer also tracks the number of LOC added. This data helps the programmer estimate the effort of future tasks; the programmer can use the data from a previous change request that is similar to a current change request as an estimate so he can plan his time accordingly. This helps the programmer to manage his time and meet the stakeholders' requirements.

3.3.4 Defect Log

The programmer also keeps a defect log; a record of all defects in the program. It includes the date the defect was found, the task performed when the defect was found, its location, its origin and when it was fixed. This helps the programmer track the time it takes to fix defects and the tasks that most often introduce them.

3.3.5 Iteration Backlog Table

When the programmer chooses the iteration backlog, he will also create an iteration backlog table. In this table the programmer will estimate the time required for each change request using historical data from the time log. As the programmer completes change requests, he will update the table with the actual time required. If the programmer stays on schedule he will complete all the change requests in the iteration backlog. If he falls behind schedule he can still complete the all the change requests in the iteration, however, other requirements may force him to complete the iteration and return the unfinished change requests to the product backlog for a future iteration.

Chapter 4
Solo Iterative Process Experience Report

This chapter presents the source code project used in this experience report and the technologies the programmer depended on.

4.1 muCommander

The program muCommander is an open source, cross platform, advanced file manager program [31]. It expands upon an operating systems native file manager, by offering an expanded, customizable view. Additionally, it supports advanced features such as browsing file systems over FTP and other connections and can browse in archive files.

The code of muCommander is 76 KLOC and has 1,070 code files. It is written entirely in Java. It has a JUnit [32] test suite that includes 441 tests covering 18.1 percent of the statements. Its GUI components use the Swing Java Foundation Classes [33] and the unit tests are dependent upon JUnit.

4.2 Eclipse Technologies

The Eclipse IDE [34] is a popular Java development environment. The programmer chose it because of the wide variety of plugins available for it. Each of the plugins used and the reasons for choosing them is discussed in the next sections.

4.2.1 JRipples

JRipples is an Eclipse plugin that assists programmers with the tasks of incremental change [35]. It has three different phases concept location, impact analysis and change propagation. It assists programmers by displaying dependencies of Java classes. It was extensively used during this project.

4.2.2 Clover Java Code Coverage & Test Optimization

The programmer used the Clover Java Code Coverage & Test Optimization tool to measure test coverage [36]. Clover has many metrics, including statement coverage, which was used as the test coverage metric. Clover has many nice features, such as the ability to create custom metrics. All metrics collected through Clover use the "Application classes" setting which is equivalent to the production code file definition in this project. This means that the metrics do not include the statements or methods in the harness.

4.2.3 Mylyn & TaskTop

Mylyn is included with Eclipse [37]; it assists users in managing and measuring the effort of tasks. The programmer used Mylyn for its timing tools. To record and export timing data in the minute granularity requires an additional plugin called Tasktop [38].

4.3 Other Technologies

4.3.1 Abbot Java GUI Test Framework

muCommander had no functional tests, which should be included in a complete verification strategy. The Abbot Java GUI Test Framework is a technology that helps build functional test [39]. It is based on the JUnit test framework and the Java Virtual Machine automated robot classes. It has classes added to help a programmer test many types of Swing components, including JButton, JCheckBox and JTextBox. The programmer used Abbot to write functional tests that test the GUI components of the change requests.

4.3.2 Subversion & TortoiseSVN

The project required a copy of muCommander to be stored on a version control system (VCS). The programmer downloaded a copy of muCommander from its public VCS and created a separate VCS for this experience report. He chose to use the Subversion (SVN) VCS [40]. To download from, commit to and manage this VCS, the programmer used TortoiseSVN [41]. It is an open source, easy to use VCS client; that includes a diff tool.

4.3.3 DiffStats

DiffStats is a tool that extracts the number of LOCs added, deleted and moved in a diff file created by TortoiseSVN. A moved line is a LOC that was deleted in one part of the change request, but then added to another part of the program during the same task. An example of moved code is a method extracted from one class to another during postfactoring. It ignores blank and comment lines. It was developed by the programmer specifically for this project.

Chapter 5
Solo Iterative Process: Experience Report

This chapter summarizes the 9 change requests the programmer implemented for this experience report. While researching muCommander to find a needed feature the programmer found the second question from the Frequently Asked Questions (FAQ) on the muCommander website that reads:

"How can I search for a specific file?

At the time of writing, you can't.

This is an often requested feature, one that we're thinking about and have a few ideas on how to implement, but it is not there yet." [31] (p. FAQ q. 5)

The programmer decided to use this as the user story for the iteration described in this experience report. The programmer then familiarized himself with the subject program before starting the iteration. He investigated the capability of the program through experimentation and visiting the website. He then used the program as his file explorer for 2 days. This time was not accounted for in the timing logs nor is there a phase of the process that includes this. It is something that the programmer often does before attempting to perform changes on a program. The time was not recorded in the time logs.

Implementing a full-fledged search feature is too large for one change request. Therefore, it was divided into multiple change requests. The programmer created the product backlog in Table 5.1.

Table 5.1 Original Product backlog

#	Title	User Story
1	Basic Search	Add a basic search function that allows a user to search in the current directory for all or part of the title of a folder or file, and return a list of the matching files and directories.
2	Recursive Search	Add the ability to search inside all directories.
3	Advanced Output	Change the output to a table similar to the main muCommander window.
4	Date Search	Allow the user search by a date of file's modification.
5	Case Sensitive Search	Add capability to search by case sensitive search terms.
6	Extension Search	Add the ability to search for files with specific extensions.
7	Properties Search	Add options to search for files based on their properties.
8	Size Search	Add the ability to search for a file by its size.
9	Regular Expression Search	Add capability to search by a regular expression.
10	Lucene Search	Incorporate the Apache Lucene search.

During the iteration, the programmer added 2 change requests to address bugs and did not finish all the change requests in the product backlog. Table 5.2 shows the change requests completed during this experience report.

Table 5.2 Product Backlog Completed

#	Title	User Story
1	Basic Search	Add a basic search function that allows a user to search in the current directory for all or part of the title of a folder or file, and return a list of the matching files and directories.
2	Recursive Search	Add the ability to search inside all directories.
3	Advanced Output	Change the output to a table similar to the main muCommander window.
4	Date Search	Allow the user search by a date of file's modification.
5	Case Sensitive Search	Add capability to search by case sensitive search terms.
6	Extension Search	Add the ability to search for files with specific extensions.
7	Properties Search	Add options to search for files based on their properties.
8	Directory Chooser Bug	Choosing a directory with the file chooser doesn't update the search directory.
9	Date Bug	DateOption is not removed when disabled.

5.1 Change Request 1 Basic Search

5.1.1 Initialization

This change request is: "Add a basic search function. The search should allow a user to search in the current directory for all or part of the title of a directory or file and return a list of the matching files and directories."

To help understand the change request, the programmer envisioned the following functionality for the change:

1. Add options to activate a search in three different ways:
    a. the "Go" menu
    b. the quick launch toolbar
    c. a hot or virtual key combination

2. Create a search window where the user can enter a search term, start a search and see the results.
3. Write a search algorithm that uses a simple loop to match the search term with files in the current directory.

5.1.2 Concept Location

The programmer extracted the following significant concepts for the change request:

- activate the search
- current directory
- search term
- matching files and directories
- "Go" menu
- toolbar
- search window
- search algorithm

The first part of the change, activating the search functionality, will need to conform to the methods and patterns of the current code and therefore is also the concept to look for during concept location. The second part of the change, a search window, the programmer planned to create as a separate class and incorporate as a component during actualization. The programmer decided to address the third part of the change in impact analysis, as it will probably require minor changes, if any.

The programmer started a dependency search for the concept of activating the search feature, by marking the `Launcher` class, which contains the program's main method as propagating. JRipples added neighbors of `Launcher` to the set of Next code files. Since the programmer had very limited knowledge of the program, he decided to visit the 43 neighbors alphabetically. `AbstractFile`, `AbstractNotifier` and `ActionKeymapIO` were visited and marked Unchanged. The programmer then visited

ActionManager; this file contains a library of all the possible actions in the program. It is used as a central location to keep all the possible actions of the program organized. Upon inspection, the programmer realized that this is where the search functionality would be added, activating the search functionality will be a new action of muCommander. This completed concept location. Figure 5.1 is a UML diagram of the code files visited during concept location.

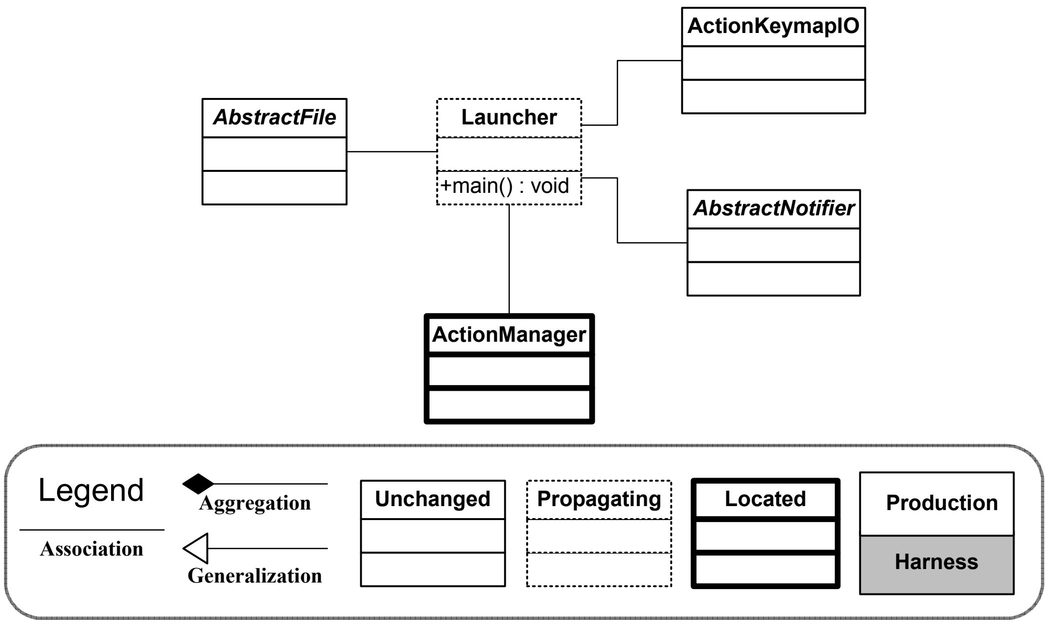


Figure 5.1 Change 1 Concept location

5.1.3 Impact Analysis

To start impact analysis the programmer switched JRipples from concept location phase to impact analysis phase. This changed ActionManager's mark from Located to Impacted and created a new Next set of code files composed of 172 of ActionManager's neighbors. The programmer visited 16 code files and marked 3 as Impacted, 1 Propagating and 13 Unchanged, see Figure 5.2. The impacted classes are:

- ActionManager, the class containing the concept extension
- MainMenuBar, the class that is responsible for the "Go" menu

- `ToolBarAttributes`, the class that defines the toolbar options

The change propagated from `ActionManager` to `ToolBarAttributes` through `ToolBar`. `Toolbar` is responsible for creating the toolbar, but delegates the responsibility of defining the buttons on the Toolbar to `ToolBarAttributes`.

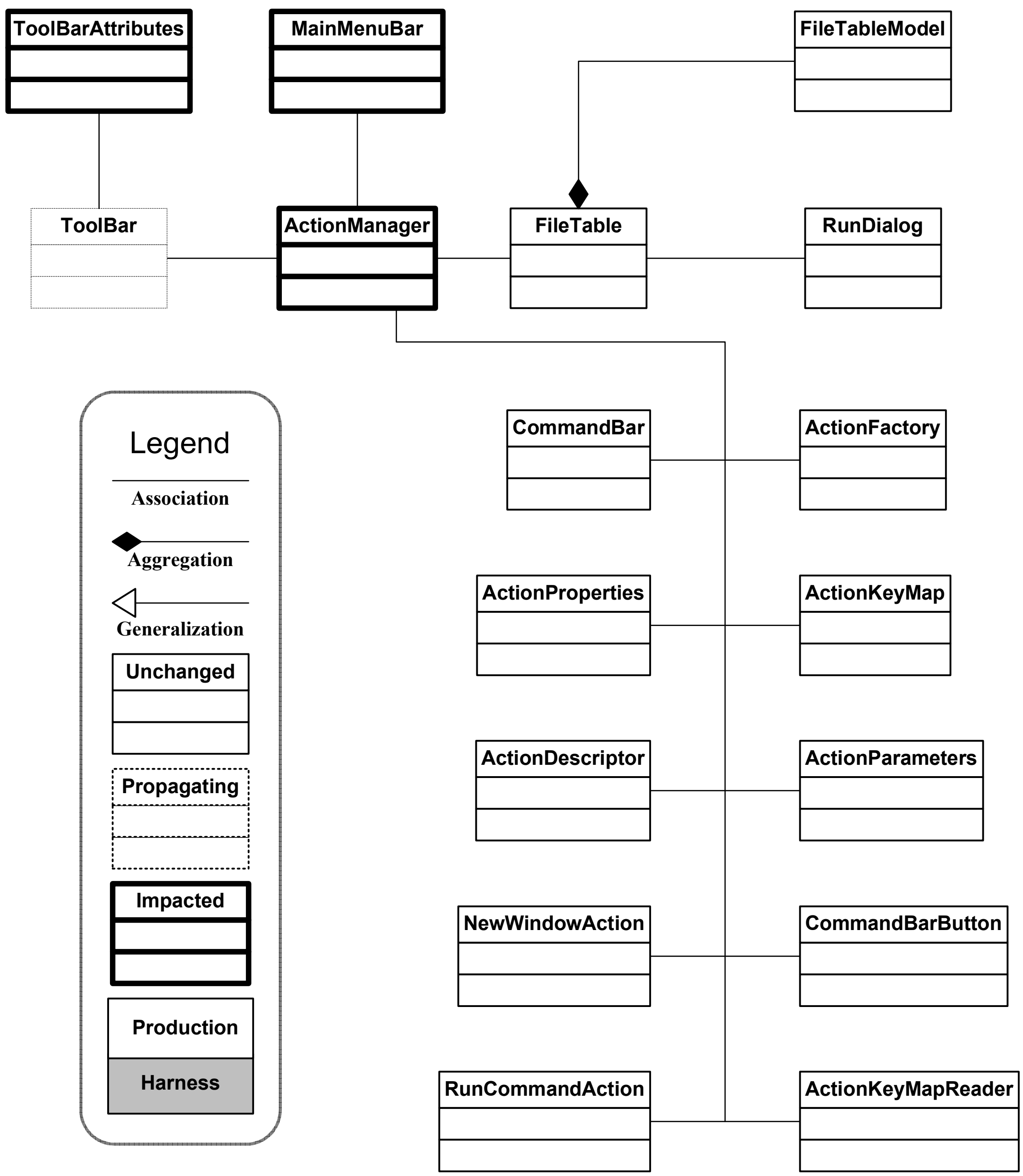


Figure 5.2 Change 1 Impact Analysis

5.1.4 Prefactoring

There was no prefactoring done in this change request.

5.1.5 Actualization

To actualize the change request, the programmer added 2 code files. The first, `SearchAction` was incorporated as a supplier of `ActionManager`. The existing code uses a factory design pattern [42], which the programmer followed when implementing `SearchAction` by modeling it after an existing code file that implements the pattern named `RunCommandAction`. The factory design pattern allows the incorporation of new suppliers that handle user events. The advantage to using this pattern is that change requests that incorporate a new supplier of `ActionManager` are unlikely to propagate beyond `ActionManager`.

The second code file contains the class `SearchDialog`, which creates the search window and implements the search algorithm. It is a component of `SearchAction`. To create the class, the programmer copied the existing muCommander class `RunDialog`, which also creates a dialog and changed it to the current change requests requirements. The programmer did this to help match the coding conventions of the existing code. The fields and methods of `SearchDialog` are:

Fields

- `MainFrame mainFrame`
- `ShellComboBox inputCombo`
- `JTextField inputBox`
- `JButton runStopButton`
- `JButton searchButton`
- `JButton cancelButton`

Methods

- `createOutputArea()`
- `createInputArea()`
- `createButtonsArea()`
- `keyPressed()`
- `actionPerformed()`
- `switchToSearchState()`

- `JButton clearButton`
- `JTextArea outputTextArea`
- `SpinningDial dial`
- `PrintStream processInput`
- `AbstractProces currentProcess`
- `Dimension MINIMUM_DIALOG_DIMENSION`
- `FileSet searchResults`

- `searchCommand()`
- `addToTextArea()`

Once these 2 were incorporated, the search window was now a registered action of muCommander. This allowed the programmer to implement the activation functionality described in concept location, by adding the action to `MainMenuBar` and `ToolBarAttributes`.

Two additional code files were added for the purpose of verification; 1 class for unit testing, `BasicSearchUnitTest` and 1 for functional testing, `BasicSearchFuncTest`. The addition of these test classes propagated to the class `Translator` that was not discovered during impact analysis. `Translator` is a supplier to `SearchDialog`; it has a sequential coupling anti-pattern because its method `loadDictionaryFile()` must be called to initialize `Translator`, otherwise calls to `Translator`'s other methods will throw an exception. However, if `loadDictionaryFile()` is called a second time, it also throws an exception. This *false multiplicity* anti-pattern preexisted in the code and meant that the new test classes could not be run together. The programmer added a `boolean` getter to `Translator` to check if the dictionary is loaded, but this does not address the sequential coupling anti-pattern, so the programmer also added a protective change request to the product backlog to change the `Translator` class to a singleton design pattern [42]. Since the change propagated to the `Translator` class solely because of a harness class requirement, it is considered part of the the harness for this change. The harness classes will be described in verification (section 5.1.7). Figure 5.3 is a UML diagram of the classes added and visited during actualization.

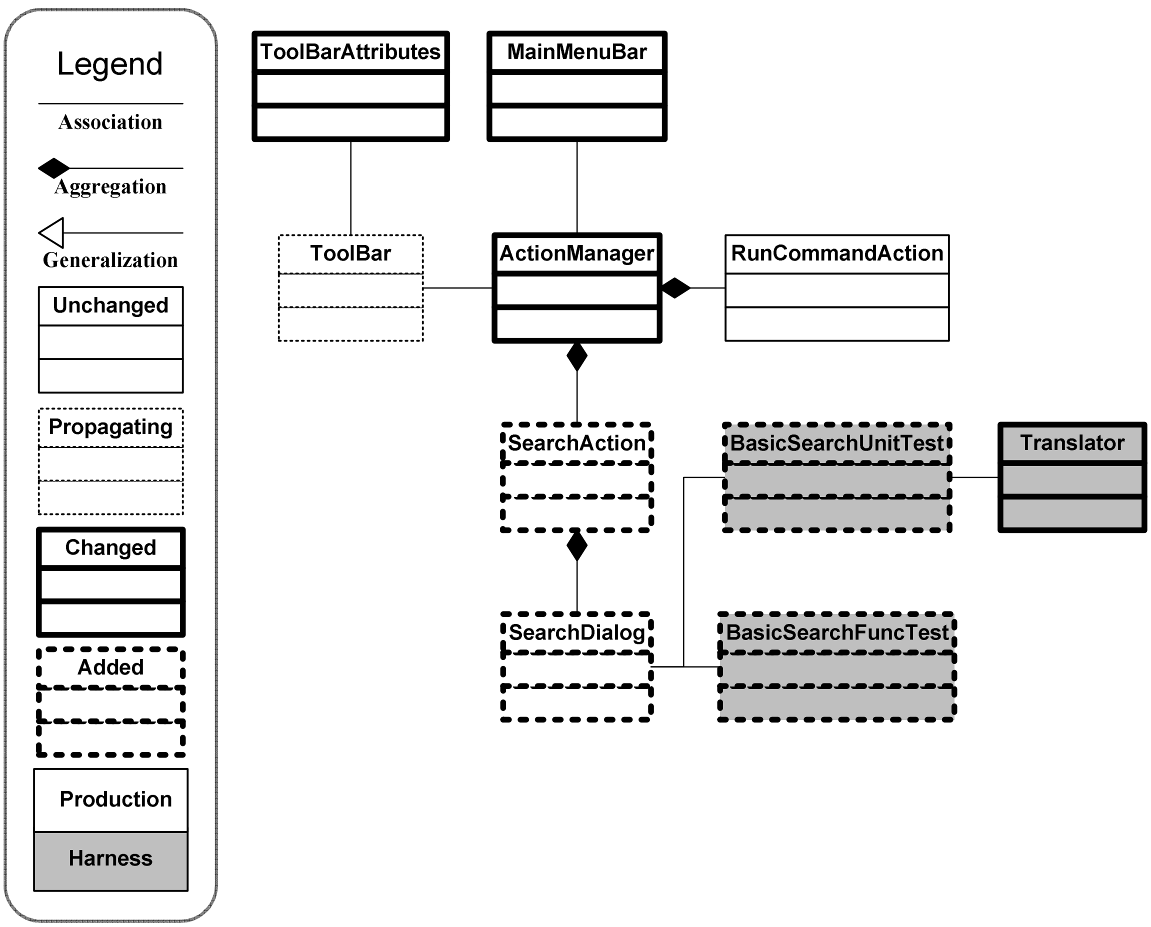


Figure 5.3 Change 1 Actualization

5.1.6 Postfactoring

During postfactoring, old comments were deleted and new comments added. Additionally, the following fields in `SearchDialog` were copied from `RunDialog`, but were not used in the class so they were deleted:

- `ShellComboBox inputCombo`
- `JButton runStopButton`
- `JButton clearButton`
- `PrintStream processInput`
- `AbstractProces currentProcess`

5.1.7 Verification

Functional and Unit testing was added for the `SearchDialog` class. During verification no bugs were found. This is most likely due to the simple nature of the request. There was an issue with the single functional test in `BasicSearchFuncTest`. It runs and passes its assertions but ends displaying a gray result, instead of the green for pass or red for fail. This occurred because a `java.lang.System.exit()` call was made by a class in the preexisting muCommander code before JUnit could make its own call to the method . This causes the Java Virtual Machine to close JUnit before it can finish running and display green or red. It also meant that only 1 functional test would run, if a second test was added, it would be skipped. The programmer did not know the cause of the problem during the change request; he researched the issue and fixed it during change request 2 (section 5.2.4). Table 5.3 shows the statement level coverage of the test harness for the production code files added during this change request.

Table 5.3 Change 1 Statement verification coverage of production code files

#	Code File	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	`SearchAction`	7	7	100.0	0	0
2	`SearchDialog`	100	87	87.0	0	0

5.1.8 Conclusion

The programmer committed the updated code to the repository as a new baseline. For the summary of the code files visited added and changed during change request 1 see Table 5.4.

Table 5.4 Change 1 Summary

Number in Code files						
Inspected Concept Location	Estimated Impact Set	Changed Set	Added during			Total Project
			Pre	Act	Post	
5	3	4	0	4	0	1,074

5.2 Change Request 2 Recursive search

5.2.1 Initialization

This change request is: “Add the ability to search inside all the directories.”

To help understand the change request, the programmer envisioned the following functionality for the change:

1. Enhance the search algorithm to:
   a. recursively search in directories it encounters

b. start a search in a specified directory

2. Add GUI components

   a. a checkbox to enable recursive searching

   b. a text field to enter directories

   c. a file chooser to use a GUI to select a directory

   d. display the path of results, in addition to the name

   e. an error message if an invalid directory is chosen

3. Add ability to stop a search before it completes

5.2.2 Concept Location

The programmer gained significant knowledge from change 1; this enabled him to extract relevant concepts from the change request and using their intensions he converted them to following significant concepts:

- search inside → recursively search
- any directory
- file system
- search algorithm
- search window
- interrupt search

After extracting the concepts and understanding the change request, the programmer decided to search for the first concept, the search algorithm, because it will have to change to implement recursive searches. This made concept location unnecessary because the programmer just implemented the search algorithm in change 1 so he knew the concept location was `SearchDialog`.

5.2.3 Impact Analysis

The concept extension was in `SearchDialog`; to start impact analysis the programmer labeled it Impacted JRipples. The programmer visited all of the 16 production neighbors of `SearchDialog`, identified by JRipples and marked them Unchanged, see Figure 5.4. The programmer visited and marked following harness code file Impacted: `BasicSearchUnitTest` and `BasicSearchFuncTest`. This resulted in an estimated impact set of 3 code files.

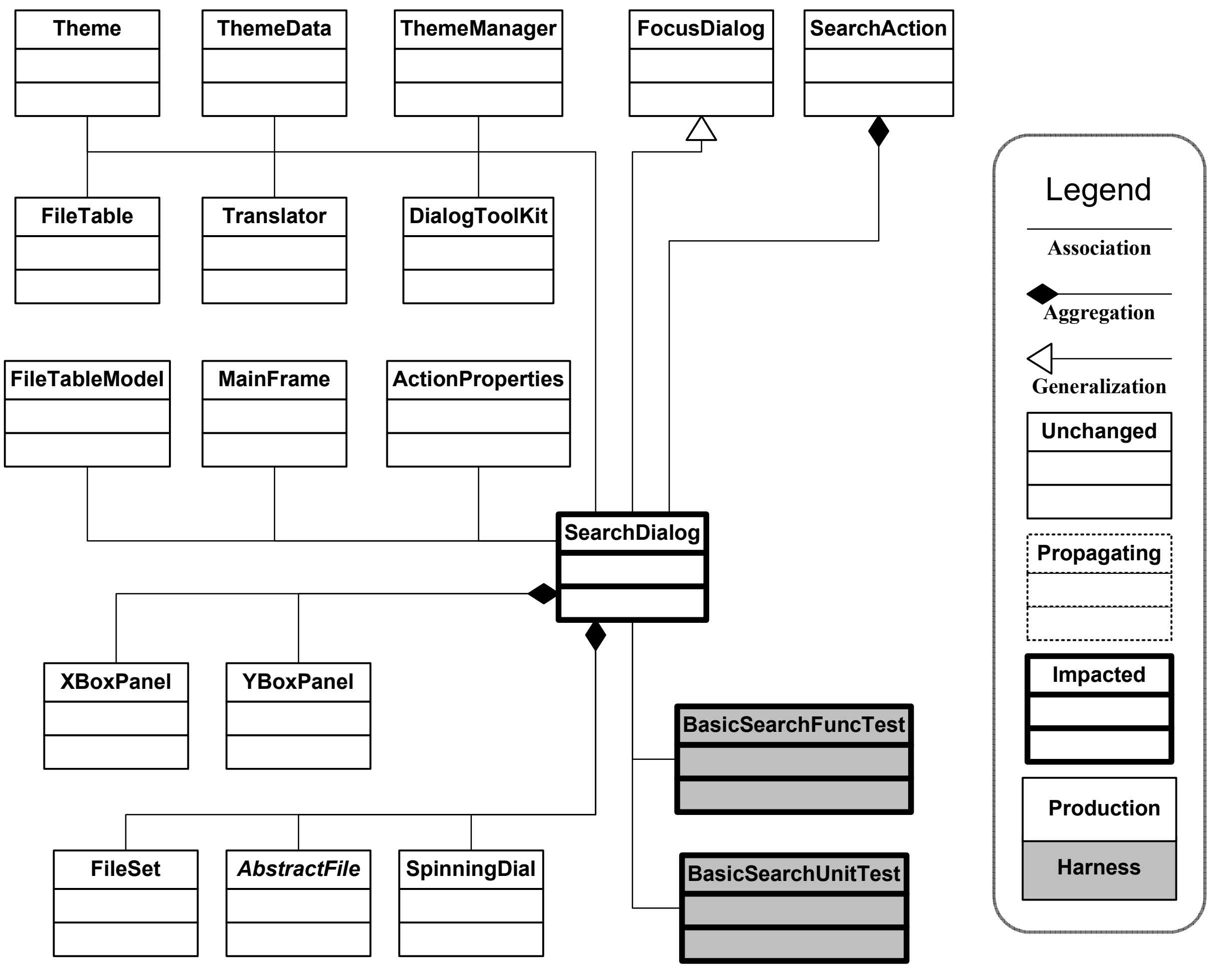


Figure 5.4 Change 2 Impact Analysis

5.2.4 Prefactoring

In preparation for the actualization of this change request, the programmer extracted 2 classes from `SearchDialog`. `SearchDialog` contained both the search algorithm and the GUI components; if the programmer added the new responsibilities of this change request to `SearchDialog`, it would have become large and difficult to understand. The first class extracted from `SearchDialog`, `SearchThread`, was given the responsibility of the search algorithm and the other, `InputPanel`, was extracted to remove the GUI features displayed in the top half of the dialog that are responsible for the user input. By separating the search logic from the GUI components, it was easier to create a separate thread for the search algorithm to run in. This way the GUI can still respond to user input while the search is executing.

The programmer also extracted 2 test classes from `BasicSearchUnitTest`. The first, `SearchThreadTest` contains the tests for `SearchThread` and the second `InputPanelTest` contains the tests for `InputPanel`. The classes extracted are shown in a UML diagram in Figure 5.5.

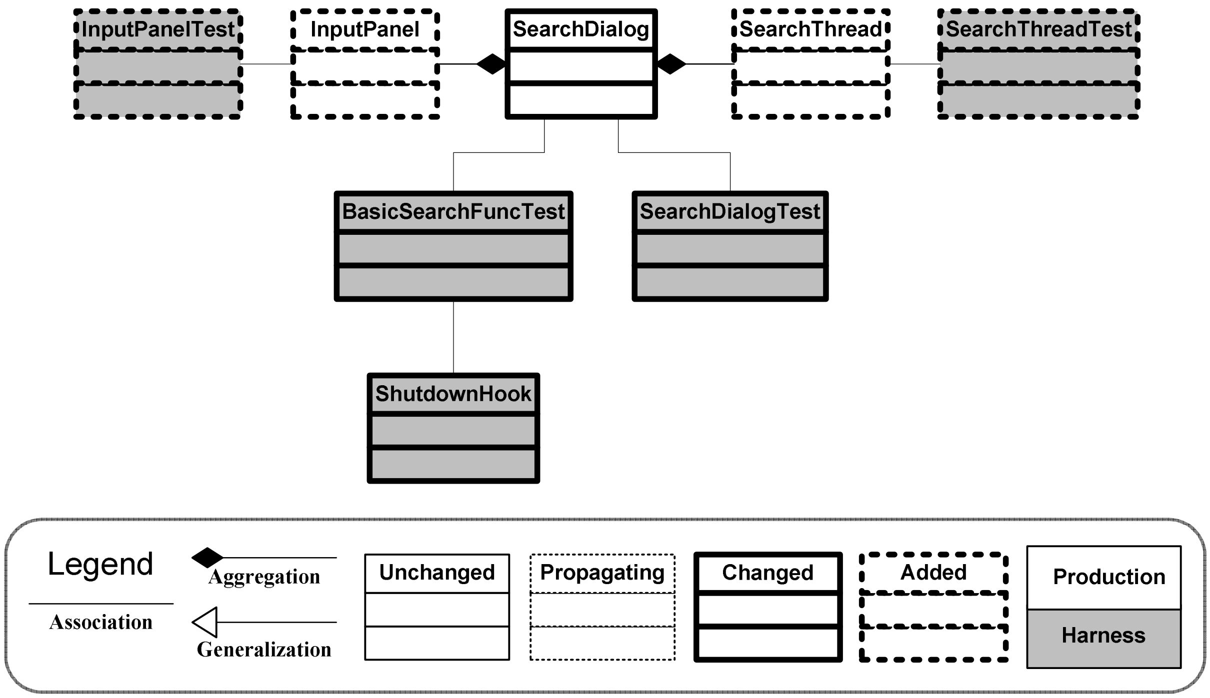


Figure 5.5 Change 2 Prefactoring

The programmer planned to add additional functional tests during this change request. To prepare for the new functional tests the programmer addressed the issue discussed previously (section 5.1.7), which is it would pass its assertions, but display a gray instead of green color, by modifying the `ShutdownHook` class. This class was not identified during impact analysis. The programmer did a grep search and determined that `ShutdownHook` contained the `java.lang.System.exit()` that was preventing JUnit from completing; he added a `boolean` field and setter method to `ShutdownHook` to allow the program to be shut down without calling `java.lang.System.exit()`. The functional test then passed, this resolved the issue and it increased the change set from 3 code files to 4. Since the change propagated to the `ShutdownHook` class solely because of a harness class requirement, it is considered part of the the harness for this change.

5.2.5 Actualization

To add the recursive search capabilities, no new code files were added to the project after prefactoring and the change did not propagate to any other code files. However, the responsibility of the `SearchThread` class was expanded by incorporation through replacement. The programmer wrote a new class that creates a new thread that recursively iterates through the file system checking the files to see if their name contains a search term and replaced the `SearchThread` code file in the project with this new code file. The replacement `SearchThread` contained the following fields and methods:

Fields

- `SearchDialog parent`
- `AbstractFile searchDirectory`
- `String searchTerm`
- `boolean recursiveSearch`

Methods

- `main()`
- `run()`
- `searchCommand()`
- `searchCommand(AbstractFile, String)`

In `SearchDialog` the programmer changed the added a new `boolean` that the `SearchThread` object checks to determine if it should continue to iterate through the file system. Then changed and added the following methods:

Changed

- `actionPerformed()`
- `switchToSearchState()`
- `runCommand()`

Added

- `notifyEnd()`
- `addSearchResult()`
- `setError()`

- `addTextToArea(FileSet)`
- `addTextToArea(String)`
- `getKeepSearching()`

The Programmer added the following 11 fields and 10 methods to InputPanel:

Fields

- `JPanel directoryPanel`
- `JTextField inputDirectoryBox`
- `JButton browseButton`
- `JLabel invalidDirectoryError`
- `File file`
- `JCheckBox recursiveBox`
- `boolean alternate`
- `Timer blinkingTimer`
- `int blinks`
- `static final int TOTALBLINKS`
- `static final int BLINK_LENGTH`

Methods

- `createDirectoryArea()`
- `chooseFile()`
- `isValidDirectory()`
- `getDirectory()`
- `flashError()`
- `isErrorEnabled()`
- `isRecursive()`
- `actionPerformed()`
- `focusLost()`
- `keyReleased()`

5.2.6 Postfactoring

After finishing the actualization phase and the change request was up and running, the code needed to be refactored because of code decay introduced during actualization. The InputPanel class had grown too large and had too much responsibility. Two classes DirectoryPanel and FlashLabel classes were extracted from it into new code files, see Figure 5.6. Both of these classes could have been incorporated as suppliers to InputPanel during actualization.

To keep the test suite organized the tests in InputPanelTest that test methods extracted to the new classes, DirectoryPanel and FlashLabel were moved into new test classes, DirectoryPanelTest and FlashLabelTest. In SearchDialogTest and SearchThreadTest the 4 methods that setup and teardown for the tests were very similar; the programmer extracted them to a new abstract class SearchDialogTestSetUp.

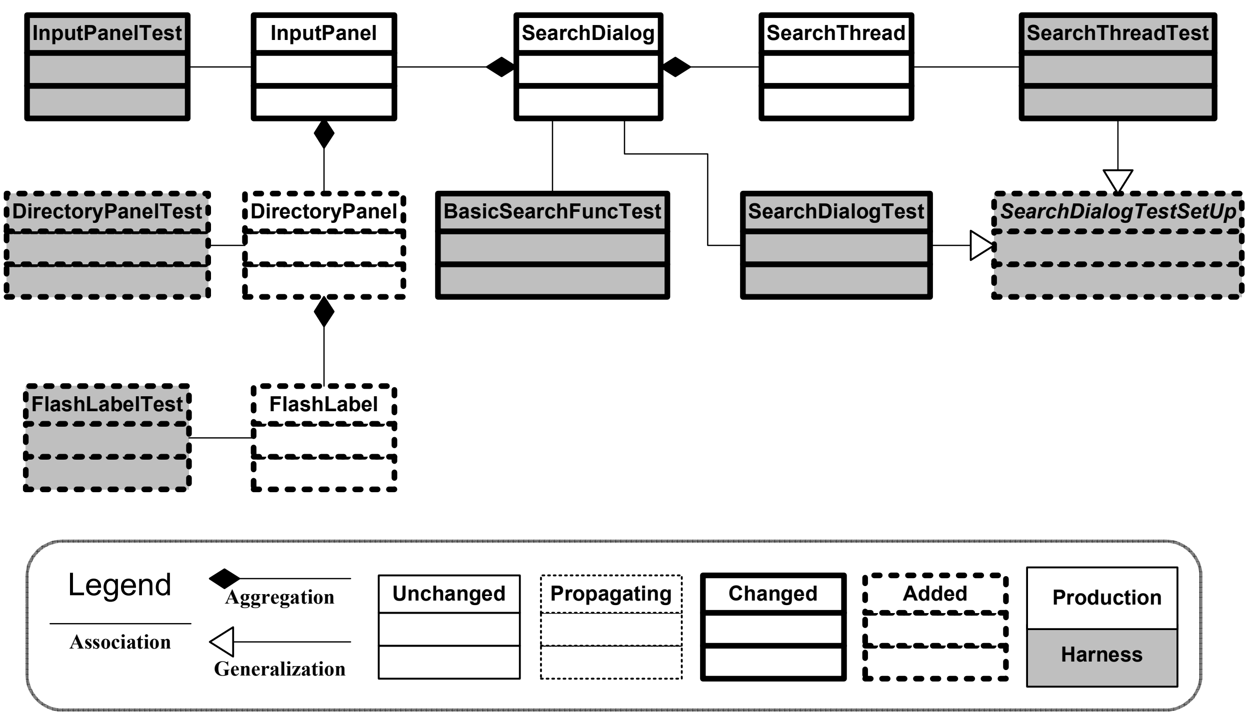


Figure 5.6 Change 2 Postfactoring

Finally, to better organize the project, the programmer created 3 new packages: org.severe.ui.dialog.search.panels, org.severe.ui.dialog.search.tests and `org.severe.ui.dialog.search.panels.tests`. Then the appropriate classes were placed into each package.

5.2.7 Verification

Unit tests expanded from 1 class to 5 plus a super class as described in the postfactoring (section 5.2.6). This included adding a total of 42 new tests to test the new functionality, 15 were deleted and 23 changed. The functional tests were also expanded, from 1 to 4 tests but remained in 1 class. During verification three bugs were found.

Two bugs were found by 2 of the new functional tests. First, when a user inputs a blank value for the directory an error message would appear, but when the test tried to type in a valid directory it would be redirected to another input location before it could complete. This was caused because an exception was thrown before text could be entered when the directory input box was selected; the catch statement was resetting the interface as if the user had finished entering a directory, even though they had not had a chance to yet. The catch statement was rewritten to do nothing, there is another catch statement to handle invalid directories after the user is finished entering.

The second bug discovered, is that a search prematurely stops if it encounters a directory that the file system marks as readable, but is set as read-only through a different mechanism. An example of this is a quarantine directory used by an antivirus program. This bug was also caused by a catch statement; when this type of exception the catch block was stopping the search, now it adds an error message, but allows the search to continue.

When modifying the tests from change request 1 the programmer realized a message displayed to the user that there were no search results found, was no longer functioning. Previously, the results were returned all at once as a set, if the set was empty a message was displayed to the user. Now the files are returned individually, so there was no set to check. The programmer added a check to the method `notifyEnd()` that is called when the search algorithm completes; if the output area is empty the no search results message is added to the output area.

All of the bugs were fixed during this change request. Table 5.5 shows the statement level coverage of the test harness for the code files added or changed during this change request.

Table 5.5 Change 2 Statement verification coverage of production code files

#	Code File	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	`DirectoryPanel`	52	41	78.8	0	1
2	`FlashLabel`	14	14	100.0	0	0
3	`InputPanel`	29	29	100.0	0	0
4	`SearchDialog`	81	76	93.8	0	1
5	`SearchThread`	19	19	100.0	0	1
6	`ShutdownHook`	41	4	9.8	0	0

5.2.8 Conclusion

The programmer committed this change request to the repository as a new baseline. During this change request, the programmer added a class to the changed set during prefactoring, see Table 5.6.

Table 5.6 Change 2 Summary

Number in Code files						
Inspected Concept Location	Estimated Impact Set	Changed Set	Added during			Total Project
			Pre	Act	Post	
0	3	4	4	0	5	1,083

5.3 Change Request 3 Advanced Output

5.3.1 Initialization

This change request is: “Change the output to a table similar to the main muCommander window.”

To help understand the change request, the programmer envisioned the following functionality for the change:

1. Change the search results display to the muCommander table file display
2. Add a results total
3. Enable the click to navigate option on the results

5.3.2 Concept Location

The programmer extracted relevant concepts from the change request and using their intensions he converted them to following significant concepts:

- muCommander window → table file display
- output → search window output area

The programmer realized there are 2 concepts in the first functionality to add, the current search results display and the muCommander table file display. For the first concept, no concept location was necessary; the programmer knew it is located in the `SearchDialog` code file from the previous changes. The second and third functionality was part of impact analysis.

To find the second concept, the table file display in the main muCommander window, the programmer did a dependency search starting in the `Launcher` code file by marking it Propagating in JRipples. One of the JRipples' Next set of code files, `WindowManager` contained a field of type `MainFrame`, which because of its name sounded very promising; he marked it Propagating in JRipples, because it has a field of type `MainFrame`.

`MainFrame` contains 2 fields of type `FolderPanel` and 2 of type `FileTable`; both of these code files sounded promising, because of their names. `MainFrame` was marked as Propagating. One of the Next code files in JRipples' set was `FolderPanel`, which the programmer also saw in his `MainFrame` visit; therefore he visited it first. It has a `boolean` variable `treeVisible`, which he changed from false to true. The programmer rebuilt and ran the program; the tree view was now visible at startup, which confirmed that the second concept location had been found. During concept location the only code file visited and marked Unchanged was `FocusDialog`.

5.3.3 Impact Analysis

For the first step of impact analysis the programmer marked the code file `SearchDialog` containing the first concept extension, the current search results display, as Impacted in JRipples. Then the programmer visited and marked the following code files Impacted:

- `SearchThread`, performs the search
- `InputPanel`, gets the user search criteria
- `FlashLabel`, displays an error to the user

- DirectoryPanel, gets the search directory
- SearchDialogTest
- SearchDialogTestSetUp, Impacted test classes inherits from
- SearchThreadTest
- BasicSearchFuncTest
- InputPanelTest
- FlashLabelTest
- DirectoryPanelTest

At this point, FolderPanel, the code file that contains the second concept extension, the muCommander table display, was included in the JRipples Next set. The programmer visited it and marked it as Impacted. The programmer visited FileTable because it is a neighbor of both FolderPanel and MainFrame. Upon reading its Javadoc description that it, “displays a folder’s contents”; the programmer marked it Impacted. JRipples added code files that the programmer suspected to be suppliers of FileTable because their names started with FileTable; he marked the following Impacted:

- FileTableModel
- FileTableHeaderRenderer
- FileTableHeader
- FileTableConfiguration
- FileTableColumnModel
- FileTableCellRenderer

Finally, `MainFrame` was marked as Impacted because it had a private method that created a `FileTableConfiguration` class need to create a `FileTable` that would be impacted. At this point 328 code files were in JRipples' Next set. The programmer marked all of these code files as Unchanged. The estimated impact set contained 21 code files at the end of impact analysis is in Figure 5.7, the Unchanged code files were left off for clarity.

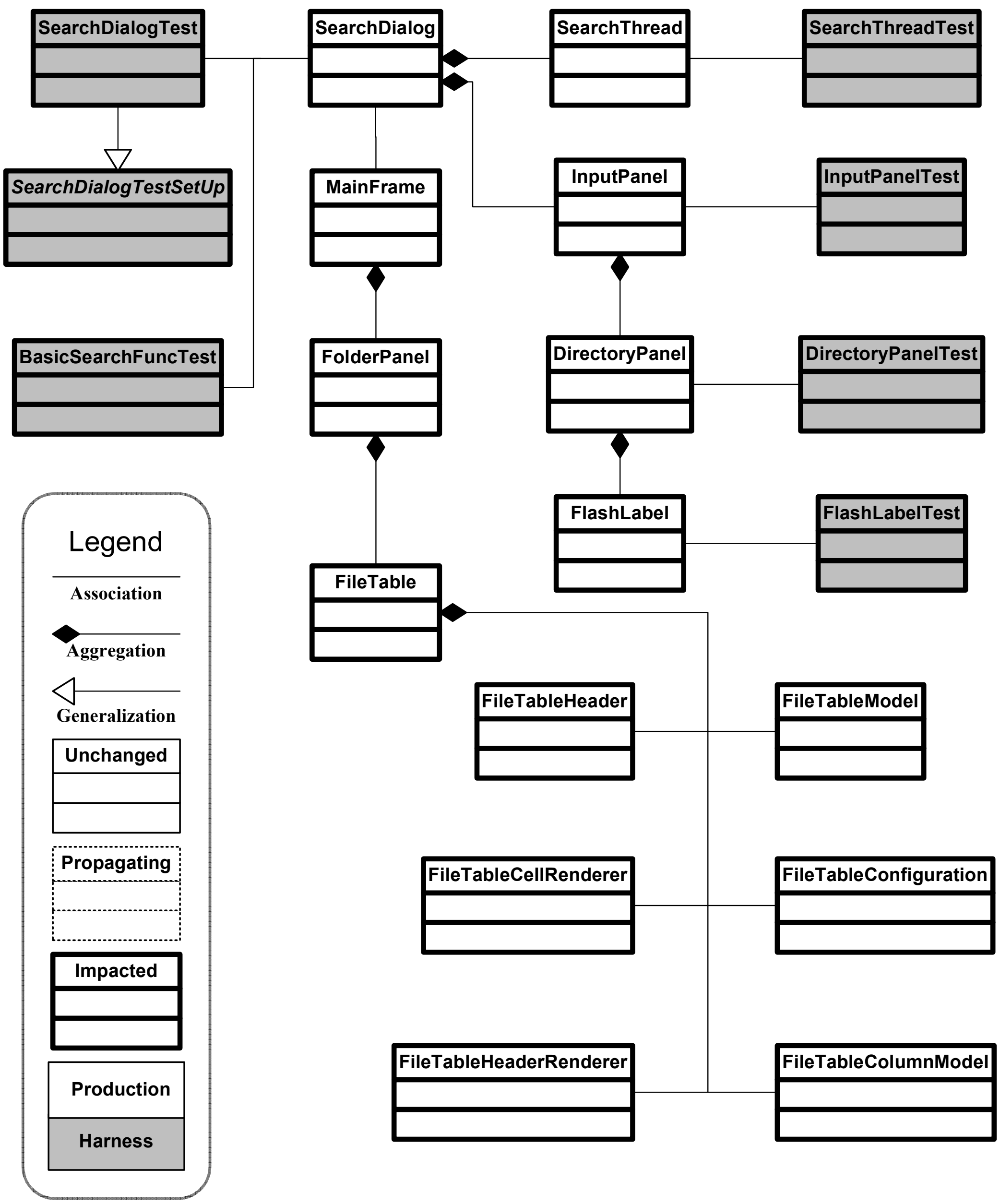


Figure 5.7 Change 3 Impact Analysis

5.3.4 Prefactoring

To prepare for this change, 2 super classes `AbstractFileTable` and `AbstractFolderPanel` were extracted from `FileTable` and `FolderPanel` respectively. The programmer extracted these classes because objects of type `FileTable` and `FolderPanel` classes can only be instantiated in an object of type `MainFrame`. This extraction allows the file table display to be contained in other types of objects. These were very large class extractions the original code files were 2069 and 1478 LOC respectively. Because of the size of the class extractions the task was not broken up into smaller tasks, such as extracting methods in the current class then moving them to the new abstract class. While that strategy may be a safe strategy, because of the size of the class extraction, the programmer perceived to be overly burdensome.

The strategy used was to move universal functionality to the abstract class and leave the rest. For example, the `FolderPanel` class has a field, `currentFolder`, of type `AbstractFile`, which is the directory displayed in muCommander. Since search results do not necessarily have a common parent directory, this field was left in `FolderPanel`. However, since all types of displays can have more files to display then their size allows, the field `scrollPane` of type `JScrollPane` was moved to the abstract class. This will allow all `AbstractFolderPanels` to have the capability to scroll through the displayed files when necessary.

Additionally, 2 suppliers of `FileTable`, `FileTableHeader` and `FileTableCellRenderer` had attributes of their parent type `FileTable` this had to be changed to type AbstractFileTable. A UML diagram showing the changed and extracted classes is in Figure 5.8.

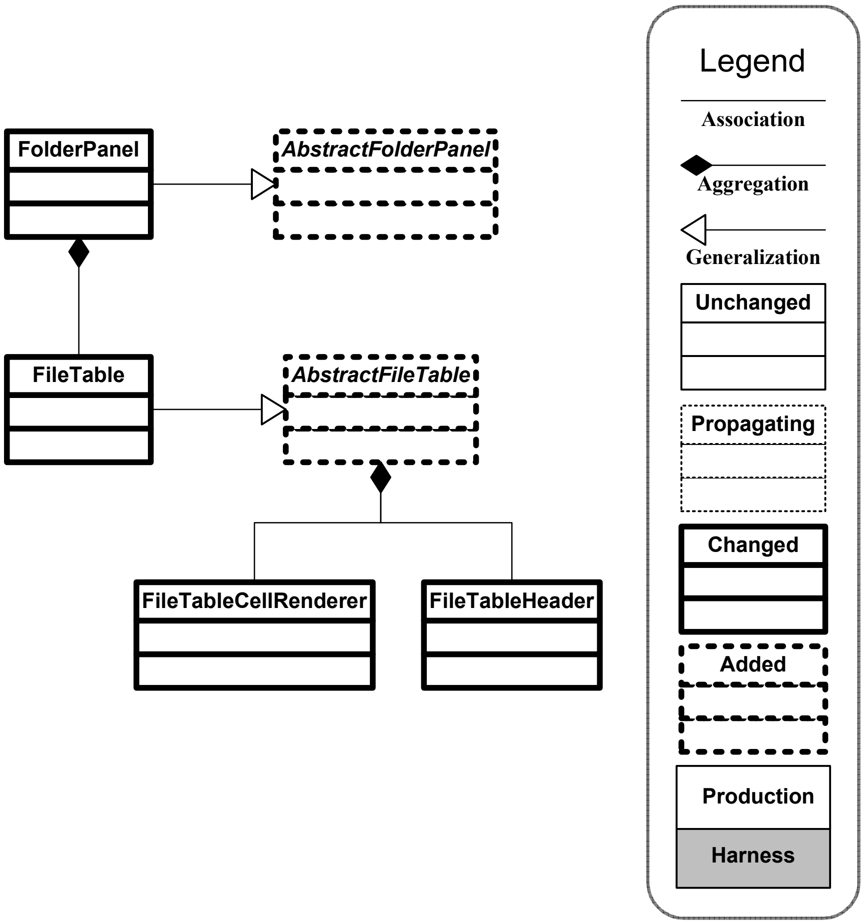


Figure 5.8 Change 3 Prefactoring

5.3.5 Actualization

To actualize the change, 2 new classes were created and added to the program through incorporation, SearchFolderPanel and SearchTable. These classes inherit from the classes extracted during prefactoring AbstractFolderPanel and AbstractFileTable. Parts of the change propagated through these new classes to their suppliers. Then an object of type SearchFolderPanel was created in SearchDialog and an object of SearchTable in SearchFolderPanel.

`SearchFolderPanel` Methods

- `clearOutput()`
- `setSearchResults()`

`SearchTable` Methods

- `doubleClick()`
- `setSearchResults()`
- `isColumnDisplayable()`
- `keyReleased()`

The overall flow to display the results starts in `SearchThread`, which contains the search algorithm; it finds the files that match the search term in the file system. It then calls methods in `SearchDialog` to display the results. Then `SearchDialog` sends the results to `SearchFolderPanel`, which sends them to `SearchTable`. `SearchTable` sends the results to the class that manages its data structure, `FileTableModel` and `FileTableCellRenderer` actually displays them to the user. Five suppliers of `SearchTable`'s needed to change, they are:

- `AbstractFileTable`, method added to show that the table is unsorted
- `FileTableModel`, method added that displayed an array of `AbstractFile` objects
- `FileTableCellRenderer`, method changed to display entire path of file, if parent is a `SearchTable` object
- `FileTableHeader`, method changed to create content menu, if parent is a `SearchTable` object
- `FileTableHeaderRenderer`, changed field from type `FileTable` to `AbstractFileTable`

Three existing test classes changed and 2 new test classes were added:

Changed

- SearchDialogTest
- SearchThreadTest
- BasicSearchFuncTest

Added

- SearchFolderPanelTest
- SearchTableTest

A UML diagram showing the code files visited during actualization is in Figure 5.9.

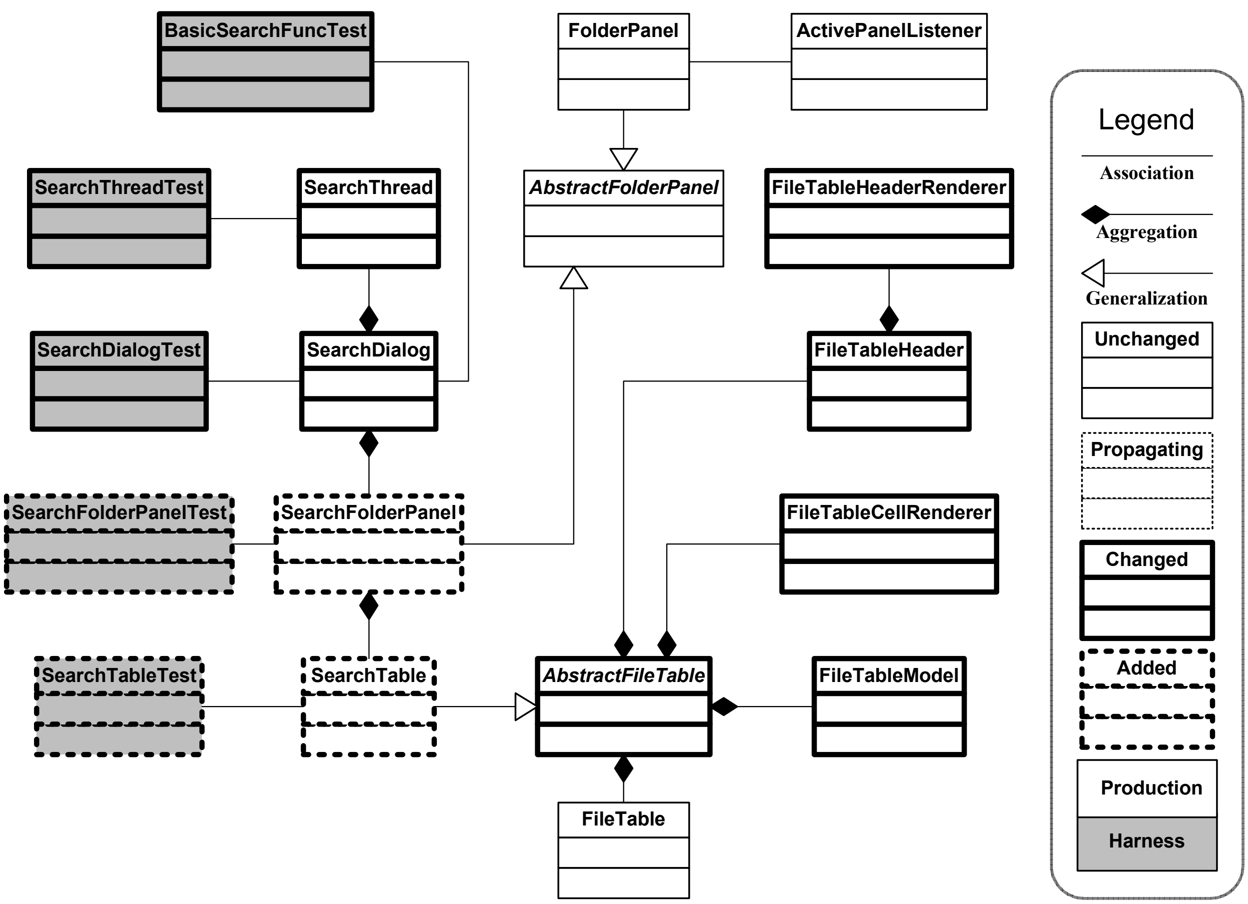


Figure 5.9 Change 3 Actualization

5.3.6 Postfactoring

Many code smells developed during actualization. The programmer added too much responsibility to the `SearchDialog` class. Therefore, he moved responsibility to a newly extracted class, `ButtonPanel` and to 3 other classes, `SearchThread`, `SearchFolderPanel` and `MainFrame`. The responsibilities moved included:

Method extracted from	Class extract to
• `createOutputArea()`	`SearchFolderPanel`
• `createButtonArea()`	`ButtonPanel.ButtonPanel()`
• `actionPerformed()`	`ButtonPanel.actionPerfomed()`
• `getKeepSearching()`	`SearchThread.getKeepSearching()`
• `getFileTableConfiguration()`	`MainFrame .getFileTableConfiguration()`

Another code smell created during actualization was that the suppliers of `AbstractFileTable` now had 2 sets of responsibilities, one set if called by an object of `FileTable` and another if called by and object of `SearchTable`, in hindsight, this could have been addressed during prefactoring. To resolve the situation the programmer extracted a super class, `AbstractFileTableModel` from `FileTableModel` and also extracted the `SearchModel` class from it. `FileTableModel` and `SearchTableModel` both inherit from `AbstractFileTableModel` and the code applicable to objects of `FileTable` use `FileTableModel` and objects of `SearchTable` use `SearchTableModel`.

The same code smell was present in the case of `FileTableCellRenderer` and `FileTableHeader`, however, the differences were smaller so the programmer extracted 2 classes, `SearchTableCellRenderer` and `SearchTableHeader` that inherit from `FileTableCellRenderer` and `FileTableHeader` respectively; they override a subset of their super class's methods. Once all these extra classes were extracted the `org.severe.ui.dialog.search.panels` package had too many classes, many of which were not panels, so a new package `org.severe.ui.dialog.search.table` was created for them. The package `org.severe.ui.dialog.search.components` was also created for `FlashLabel`.

The class extraction of `AbstractFileTableModel` propagated to 7 classes not in the estimated impact set that depended on `FileTableModel` as a supplier. Six of the classes required a field or temporary variable type to be changed to `AbstractFileTableModel` from `FileTable` and 1 required a getter call to be cast to a `FileTable`. The getter is inherited from `AbstractFileTable`; it was determined that the best solution was to change these classes. By using a generic type future changes should be easier.

Many of the harness classes were creating the same `AbstractFile` objects or using instances created in the `SearchDialogTest` class. These were all extracted to a new harness class `TestConstants`. Some of the code files added during this change request were changed during postfactoring resulting in a postfactoring change set of 32 code files, see Figure 5.10.

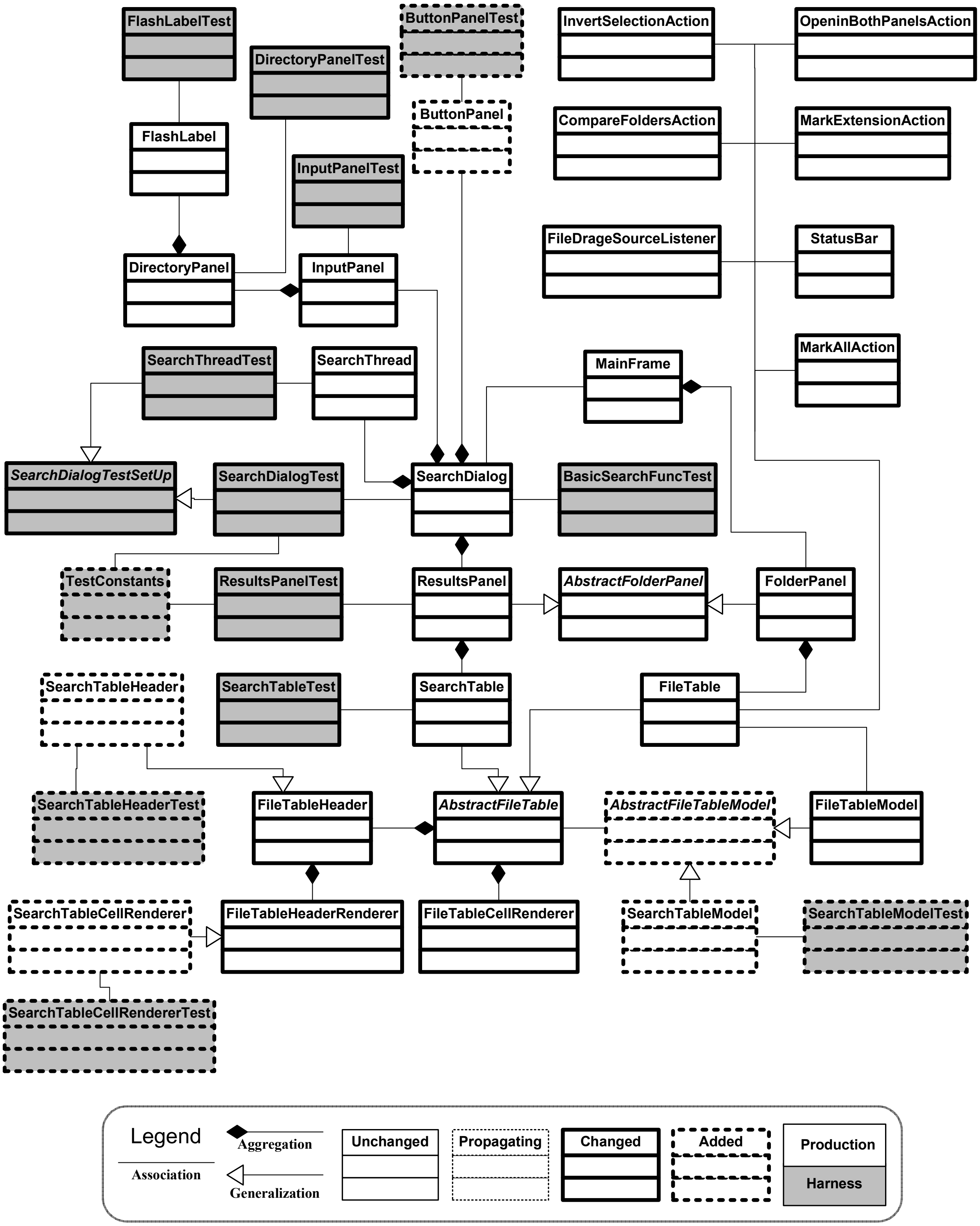


Figure 5.10 Change 3 Postfactoring

5.3.7 Verification

All the regression tests passed; no new regression tests were added for the classes impacted by refactoring. The statement level coverage for `FolderPanel`, `FileTable` and its suppliers was low; `FileTableHeader` has only 14% coverage. Therefore a protective change request with a priority 4, minor problem not involving primary functionality, was added to the backlog to improve the test suite of these classes. The programmer added a similar change request for the 7 action code files added to the impact set for the same reason; for example, `FileDragSourceListener` has only 11% statement coverage, see Table 5.7

The classes in the `org.severe.ui.dialog` packages now each have their own unit test class. All harness code files are in their own package, which has the same name as the package containing the class being tested plus *tests*. There is 1 functional test class, `BasicSearchFuncTest`. During verification 2 bugs were found, both in the new classes extracted during postfactoring.

The first bug was in `SearchTableModel`; it was getting the path of the parent folder of the search result instead of the path of the search result in the `fillCellCacheAtRow()` method. The second bug was in `SearchTable`, in the `addSearchResultMethod()`. It needs to call `resizeAndRepaint()`, an inherited method after adding the first result, to allow the table to resize the columns to the Objects in them. Both of these bugs were fixed when they were found.

Table 5.7 Change 3 Statement verification coverage of production code files

#	Code File	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	AbstractFileTable	274	195	71.2	0	0
2	AbstractFileTableMo	37	21	56.8	0	0
3	AbstractFolderPanel	60	35	58.3	0	0
4	ButtonPanel	23	23	100.0	0	0
5	CompareFoldersActio	43	6	14	0	0
6	DirectoryPanel	51	42	82.4	0	0
7	FileDragSourceListe	27	3	11.1	0	0
8	FileTable	331	89	26.9	0	0
9	FileTableCellRender	95	84	88.4	0	0
10	FileTableHeader	28	4	14.3	0	0
11	FileTableHeaderRend	18	18	100.0	0	0
12	FileTableModel	163	120	73.6	0	0
13	FlashLabel	14	14	100.0	0	0
14	FolderPanel	328	144	43.9	0	0
15	InputPanel	29	29	100.0	0	0
16	InvertSelectionActi	16	6	37.5	0	0
17	MainFrame	210	122	58.1	0	0
18	MarkAllAction	15	8	53.3	0	0
19	MarkExtensionAction	45	6	13.3	0	0
20	OpenInBothPanelsAct	34	9	26.5	0	0
21	ResultsPanel	26	25	96.2	0	0
22	SearchDialog	42	43	97.7	0	0
23	SearchTable	34	33	97.1	0	1
24	SearchTableHeader	38	38	100.0	0	0
25	SearchTableModel	65	65	100.0	0	1
26	SearchThread	27	25	92.6	0	0
27	SearchTableCellRend	10	10	100.0	0	0
28	StatusBar	207	151	72.9	0	0

5.3.8 Conclusion

The programmer committed this change request to the repository as a new baseline. The changed set was 11 code files, while the estimated impact set was 21, see Table 5.8. Two of the code files in the estimated impact set, but in the changed set are `FileTableConfiguration` and `FileTableColumnModel`; they are suppliers to `FileTable`. During impact analysis the programmer thought the changes to `FileTable` were so significant that these suppliers would also have to change; however the change never propagated to them. The other estimated impact set code files not in the changed set were changed during postfactoring. The change was more difficult than the programmer originally thought he simplified actualization by making the changed set smaller. This resulted in more code smells that he addressed during postfactoring. The programmer also changed 7 code files during postfactoring that were not part of the estimated impact set (section 5.3.6).

Table 5.8 Change 3 Summary

Number in Code files						
Inspected Concept Location	Estimated Impact Set	Changed Set	Added during			Total Project
			Pre	Act	Post	
6	21	11	2	4	10	1,099

5.4 Change Request 4 Date Search

5.4.1 Initialization

This change request is: "Allow the user search by a date of file's modification"

To help understand the change request, the programmer envisioned the following functionality for the change:

1. Add date criteria to the search algorithm

2. Add a check box to turn date searching on and off

3. Add text boxes to enter before and after dates

4. Add calendars to click on before and after dates

5.4.2 Concept Location

The programmer extracted relevant concepts from the change request and using their intensions he converted them to following significant concepts:

- file created/modified date
- a specific date
- search
- file → file name
- calendars → Java file chooser
- search algorithm

The programmer determined the concept to locate is the search algorithm. No concept location was needed for this change request. Based on experience obtained during previous change requests the programmer knew the search is located in the `SearchThread` class which was created during change 2. Functionalities 2 to 4 were added during actualization through incorporation of new classes.

5.4.3 Impact Analysis

The programmer started a dependency search by marking the code file containing the concept extension, `SearchThread` Impacted in JRipples. The programmer then visited and marked the following code files from JRipples' Next set Impacted:

- `SearchDialog`, has an object of `SearchThread` whose constructor will change
- `InputPanel`, date range GUI component added here

- `BasicSearchFuncTest`
- `InputPanelTest`
- `SearchDialogTest`
- `SearchThreadTest`
- `ButtonPanel`, will be responsible for checking to make sure there are no errors in the search criteria, before a search starts
- `DirectoryPanel`, the error it displays will move to a central management location for errors
- `DirectoryPanelTest`
- `ButtonPanelTest`
- `TestConstants`

The programmer visited `AbstractFile`; it has a method, `getDate()`, that can be used to compare an `AbstractFile`'s date to a date range; since this is all the search algorithm requires for this change request, it was marked Unchanged. This change request will require a date to be formatted; the programmer knew `AbstractFileTable` formatted a date from change request 3. `AbstractFileTable` was already in JRipples' Next set, the programmer visited it and found it calls a static method in the class `CustomDateFormat`; therefore, `AbstractFileTable` was marked as Propagating. JRipples added `CustomDateFormat` to the Next set and the programmer visited it. It has a method, `getDateFormatString()` that returns a `String` containing the date format based on setting in the preference file. It would work, but it included the time, the programmer marked it Impacted; it will need a new method that returns a date format without the time. The estimated impact set of 13 code files is shown in Table 5.13.

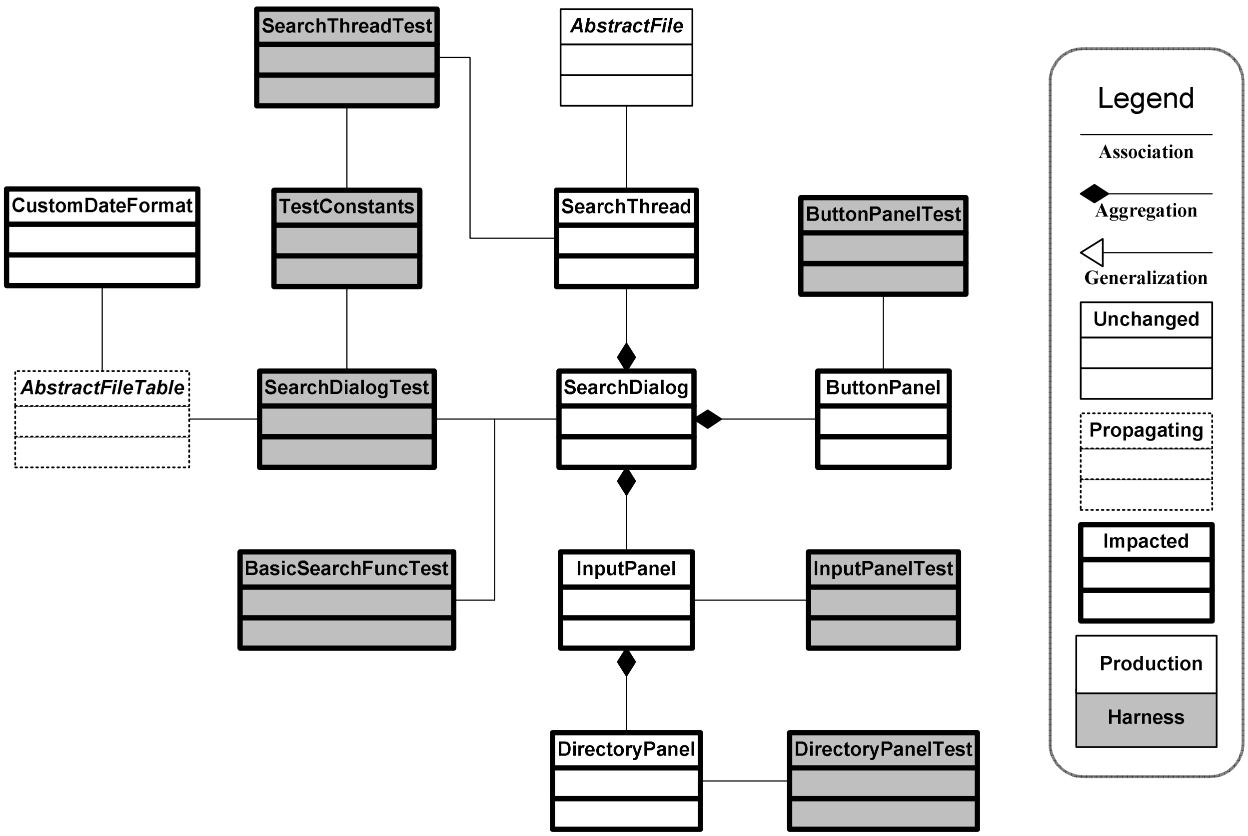


Figure 5.11 Change 4 Impact Analysis

At this point JRipples had 112 code files in the Next set. These code files were visited in a similar manner as in change 3. Code files such as `MarkForwardAction` were just marked as Unchanged based on their names. But, other code files, such as `ResultsPanel` that is part of the search dialog, were inspected more closely. Ultimately, all of these code files were marked as Unchanged.

5.4.4 Prefactoring

To prepare for this change request the programmer extracted the class `ErrorManager` from `DirectoryPanel`. The programmer did this because the program will handle multiple types of errors; instead of having `SearchDialog` check each error to see if it is enabled before a search, it will just check with this extracted class. The following `DirectoryPanel` fields and responsibility was extracted from these methods:

DirectoryPanel	ErrorManager
• `flashError()`	`flashErrors()`
• `isErrorEnabled()`	`isErrorEnabled()`
• `actionPerformed()`	`disableError()`
• `focusLost()`	`enableError()`
• `keyReleased()`	`disableError()`

This extracted class will also flash all the enabled errors if the user tries to start a search with an error enabled. This refactoring was done to make the change request easier, not because of existing code smells. A matching harness class, `ErrorManagerTest` was extracted from `DirectoryPanelTest` and the class extractions propagated to 3 more production and 3 harness code files see Figure 5.12. This is because the object of `ErrorManager` was created in `SearchDialog` and it replaced dependency these code files had with `DirectoryPanel`.

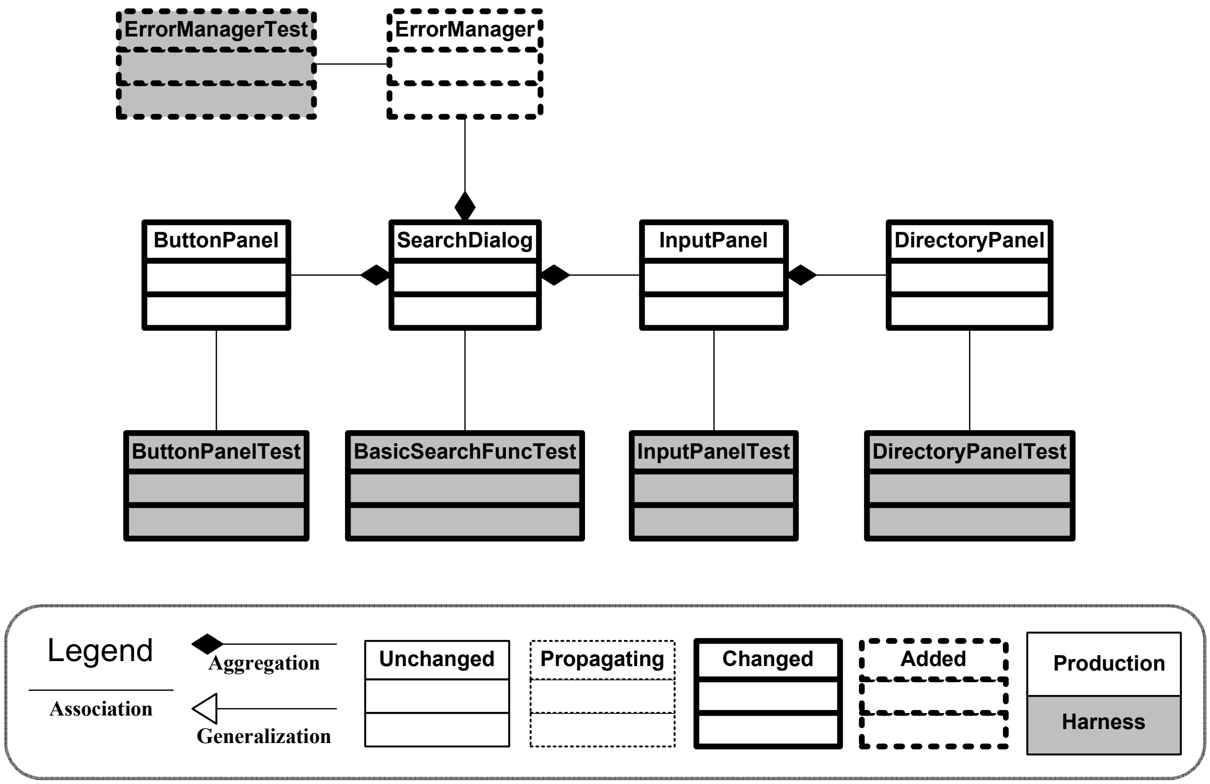


Figure 5.12 Change 4 Prefactoring

5.4.5 Actualization

To actualize this change request, the programmer incorporated a new supplier of `InputPanel` called `DatePanel` that extends `JPanel`. This class contains all the GUI components of the change request description. This class gets dates from the user as text and creates `Date` objects from the text. It performs error checking to make sure that the user entered a valid date and checks to make sure that the minimum date is less than the maximum date.

Fields	Methods
• `JCheckBox dateBox`	• `createDateTextBox()`
• `JLabel dateLabelBefore`	• `createCalendarButton()`
• `JLabel dateLabelAfter`	• `setEnabled()`

- JTextField minDateTextBox
- JTextField maxDateTextBox
- JButton minCalButton
- JButton maxCalButton
- DateFormat dateFormat
- FlashLabel dateError
- Date minDate
- Date maxDate
- ErrorManager errorManager
- boolean minError
- boolean maxError
- boolean minGreaterError
- datePanelSetEnabled()
- actionPerformed()
- focusLost()
- getErrorMessage()
- isError()
- dateTextBoxCheck()
- checkMinLessThan()
- getMinDate()
- getMaxDate()
- isDateSearch()
- keyReleased()
- checkYear()

To create a border for the class that has a `JCheckBox` in it the programmer incorporated a supplier that was provided by Kumar under a GNU License called `ComponentTitledBorder` [43]. A harness class to test it was also added.

To add GUI calendars for the user to select a date, new classes were incorporated by the programmer. These classes were taken from a program called `JCalendar` written by Toedter and available online under the GNU Lesser General Public License [44]. The program contained more functionality then needed so specific classes were chosen. These classes are:

- JCalendar
- JDayChooser

- JMonthChooser
- JYearChooser
- JSpinField

These classes used together made up a very feature rich GUI calendar with a month drop down box and a year text box, both of which have buttons to increment or decrement their values. They were placed in a new package called org.severe.ui.dialog.calendar. The programmer added a unit test class for each class and a functional test class that tests the functionality of all the classes together. These harness code files were all added to a new package, org.severe.ui.dialog.calendar.tests.

The programmer added a static method, getDateNoTimeFormatString(), to CustomDateFormat that returns a DateFormat String that is the same as the date format specified in the program's preferences file, but without the time. This allows the user to choose a date in the same format as the application display, but without the time.

The SearchThread class is responsible for the search algorithm; the algorithm is in a method recursiveSearch(). The programmer added a new method, isInDateRange() that recursiveSearch() calls, if the user enables a date search. A boolean parameter was added to the SearchThread constructor that is set to true if the date search is enabled; because of this SearchDialog, which creates the SearchThread object, was also changed. A UML diagram showing the changed and added classes is in Figure 5.13.

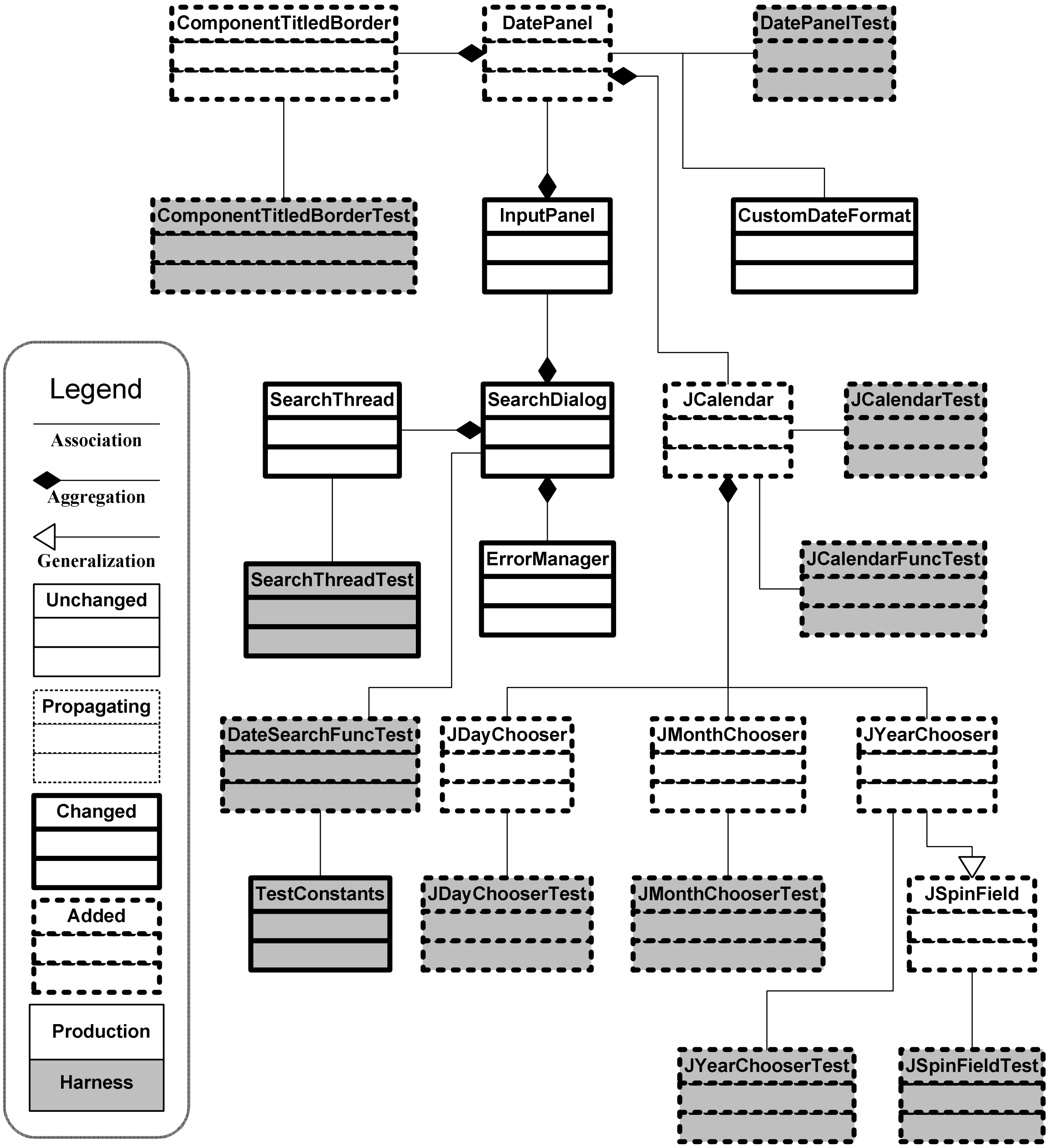


Figure 5.13 Change 4 Actualization

5.4.6 Postfactoring

The `DatePanel` class that the programmer incorporated during actualization was too large and had too much responsibility. The class `DateField` was extracted from it. It extends the `JTextField` class, see Figure 5.14. It adds methods to customize the class to only accept objects of type Date; by parsing the text entered into `Date` objects

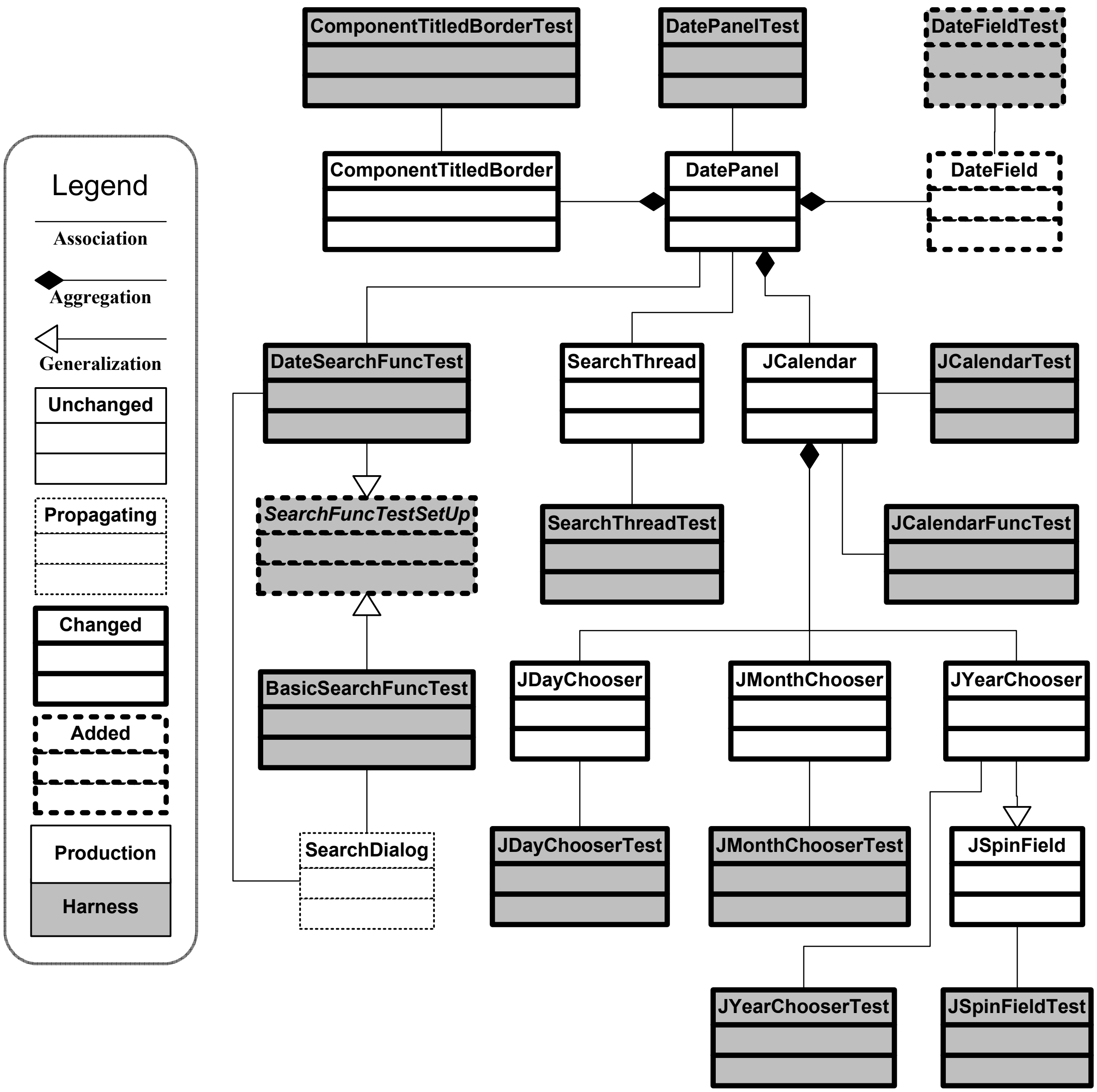


Figure 5.14 Change 4 Postfactoring

In the classes added from `JCalendar`, each class had a `main()` method and methods to set the locale to a different value than the operating system. These methods were removed because they are not needed. The programmer also performed other tasks, such as moving the fields from the end of the code file to the beginning to match the style of muCommander. `ComponentTitledBorder` had no Javadoc comments so the programmer added them to make future changes easier.

Postfactoring propagated from `DatePanel` to `InputPanel` and `SearchDialog` to `SearchThread`, which needed Javadoc added to the new method added during actualization. In the case of existing classes such as `SearchThread`, the cleanup was made necessary because of actualization.

The programmer visited the `DateSearchFuncTest` harness class and realized much of the `setUp()` and `tearDown()` methods were the same as the `BasicSearchFuncTest` class. The 2 classes are not neighbors, but propagate through `SearchDialog`. To remove the duplicated code the programmer extracted a super abstract class, `SearchFuncTestSetUp` from `BasicSearchFuncTest` and `DateSearchFuncTest` that has `setUp()` and `tearDown()` methods. It is similar to the abstract class `SearchDialogTestSetUp` that was extracted during change request 2. All 3 of these harness code files were put in a new package `org.severe.ui.dialog.search.functional.tests`. These functional tests take significantly longer to run than unit tests; having them in their own package makes it easier to run them separately.

5.4.7 Verification

After the change request all the regression tests passed. There was a unit test class added for each class added during the change; in addition, an abstract class was extracted during postfactoring to make future test easier to add. A class of constant objects, `TestConstants`, was also extracted, that can be used across the test suite.

Finally, the programmer added 2 new functional test classes, `DateSearchFuncTest` and `JCalendarFuncTest`; for a total of 3 functional test classes, see Table 5.9.

Table 5.9 Change 4 Statement verification coverage of production code files

#	Code file	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	`ButtonPanel`	26	26	100.0	0	0
2	`ComponentTitledBorder`	35	35	100.0	0	0
3	`CustomDateFormat`	22	13	59.1	0	0
4	`DateField`	55	54	98.2	0	0
5	`DatePanel`	89	86	96.6	0	2
6	`DirectoryPanel`	50	41	82.0	0	0
7	`ErrorManager`	13	13	100.0	0	0
8	`InputPanel`	36	36	100.0	0	0
9	`JCalendar`	75	60	80.0	0	0
10	`JDayChooser`	142	133	93.7	0	0
11	`JMonthChooser`	76	63	82.9	0	0
12	`JSpinField`	64	54	84.4	0	0
13	`JYearChooser`	15	15	100.0	0	0
14	`SearchDialog`	43	42	97.7	0	0
15	`SearchThread`	40	38	95.0	0	0

During verification 2 bugs were found, both in the new classes created during actualization. The first bug was in `DatePanel`; if the user types a date with a 2 digit year, such as 99 or 03, the `Date` object created by parsing had a 1st century year. The programmer added a new method to parse the date into a user expected date, such as 1999 or 2003. The second bug was that the `FocusLost` event that should trigger the creation of `Date` objects to use as search criteria would be scheduled after the `ActionListener` event that started the search. This would cause a search without a date, even though a date was displayed to the user. The programmer added a `KeyListener` event to parse the date after each keystroke to solve the problem.

5.4.8 Conclusion

The programmer committed this change request to the repository as a new baseline. The changed set had 1 less code file than the estimated impact set, see Table 5.10. During impact analysis, the programmer thought the change would propagate to the harness code file `SearchDialogTest` because `SearchDialog` was impacted. However, the change to `SearchDialog` affected 1 LOC in 1 method. This did not change the contract of the method with any client or supplier so the harness class was not impacted.

Table 5.10 Change 4 Summary

Number in Code files						
Inspected Concept Location	Estimated Impact Set	Changed Set	Added during			Total Project
			Pre	Act	Post	
0	13	12	2	16	3	1,120

5.5 Change Request 5 Case Sensitive Search

5.5.1 Initialization

This change request is: “Add capability to search by case sensitive search terms.”

To help understand the change request, the programmer envisioned the following functionality for the change:

1. Add case sensitive criteria to the search algorithm
2. Add a check box to turn case sensitive searching on and off

5.5.2 Concept Location

The programmer extracted relevant concepts from the change request and using their intensions he converted them to following significant concepts:

- case sensitive
- enable/disable
- file → file name
- search
- search algorithm

No concept location was needed for this change. The concept to location, the search algorithm, was the same as change request 4, the `SearchThread` class. Functionality number 2 was identified during impact analysis.

5.5.3 Impact Analysis

To start impact analysis the programmer marked `SearchThread` as Impacted in JRipples. The programmer visited and marked Impacted the following code files from JRipples' Next set:

- `InputPanel`, will add the case sensitive `JCheckBox`
- `SearchDialog`, will add an object of a class extracted from `SearchThread`
- `DatePanel`, extract fields from it `DateField`
- `DateField`, receive extracted fields from `DatePanel`
- `DirectoryPanel`, gets the user input directory

The programmer visited the harness code files in JRipples' Next set and marked 10 Impacted; these are the test classes for classes in the Impact set already, except for `ButtonPanelTest`. It is the test for, `ButtonPanel`, which is not in the impact set. It is impacted, because one of its tests calls a method, `searchCommand()` in

`SearchDialog` whose definition will change. The programmer marked 41 code files Unchanged, see Figure 5.15.

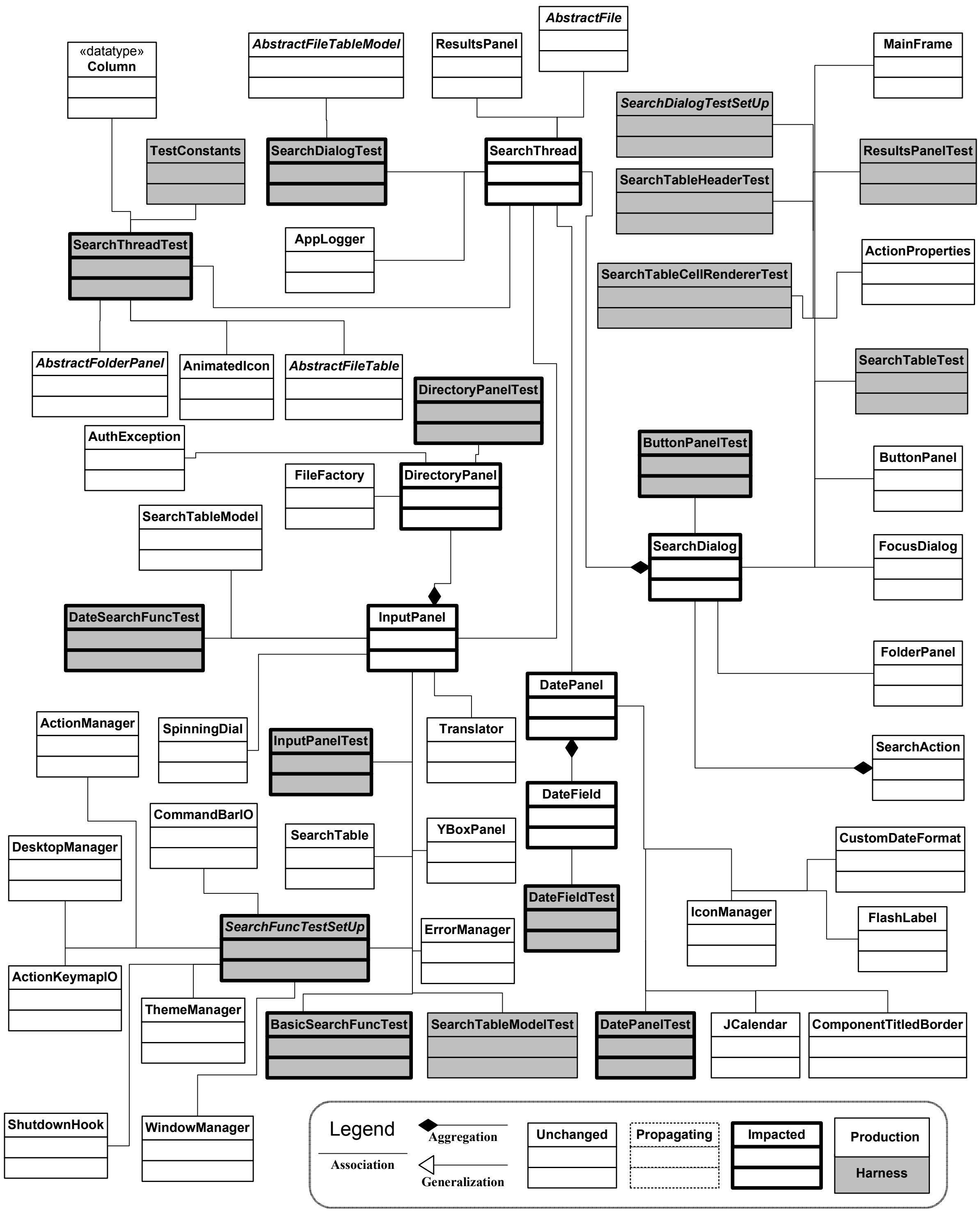


Figure 5.15 Change 5 Impact Analysis

5.5.4 Prefactoring

During impact analysis the programmer visited and realized that `SearchThread` had 2 responsibilities, one to create a separate thread that iterates through the files of the file system and 2 to check if each file met the search criteria. This made sense when `SearchThread` was extracted from `SearchDialog`, because there was only one search criterion, the file name. However, a second, date search criteria was added during change request 4 and a third criteria was going to be added during the current change request. The programmer decided to refactor this responsibility from `SearchThread` during prefactoring to make it easier to add a separate the search algorithm to run in during actualization.

During the last change a method was added to `SearchThread` to checks if a file's modified date is within a user specified date. The current structure encourages any new change request that adds a search criterion to add a new method with logic that checks the specific criteria. Then the `recursiveSearch()` method, will call this method to see if a file meets the criteria. This will make `SearchThread` a very large class, with a wide variety of responsibilities. To stop this from occurring, a strategy design pattern [42] was implemented. This will allow any new search functionality to create a class that decides if a file meets its criteria; the `SearchThread` class will not need to know anything about the algorithm that the new search option classes implement. This means adding new search options will be unlikely to propagate to `SearchThread`.

The programmer extracted a new class from `SearchThread` to manage the search criteria responsibility called `SearchManager` and created an interface, SearchOption. Classes that implement the SearchOption interface can be added to a list of criteria in SearchManager dynamically. These classes contain their own algorithms to decide if a file meets their responsibility of the search criteria. When a search is executed, SearchManager will check with all the classes on its list to decide if a file meets all the search criteria. The class SearchThread had the responsibility to check the date of a file extracted from it to a new class, DateOption that implements SearchOption; SearchThread then had just its original responsibility, of recursively stepping through the files in the file system.

This prefactoring moved the concept location from SearchThread to SearchManager. It also meant that the class that contains the concept location, SearchManager, would not need to be changed during actualization.

After, the new SearchManager and DateOption classes were extracted, it became apparent that some of the responsibility left in DatePanel during the last change, should be moved to DateField; namely the JButton that opens a dialog that allows the user to select a date from a calendar. Even though the programmer extracted DateField from DatePanel during the last change request, it was apparent that code smell were still present that needed to be addressed. There were still 2 objects of type JButton in DatePanel that should be in DateField. Additional fields moved and methods changed from DatePanel to DateField are:

Fields

- JCheckBox dateBox
- JButton minCalButton

Methods

- createDateTextBox()
- createCalendarButton()

- JButton maxCalButton
- DateFormat dateFormat

- actionPerformed()
- propertyChange()
- getMinDate()
- getMaxDate()
- isDateSearch()

The other classes that have responsibility to match the search criteria were also changed. The responsibility for matching the search term to the file's name was moved from the InputPanel class to a new class SearchTermOption, which implements SearchOption.

The recursive search and start directory responsibility were extracted to SearchManager from SearchThread. A UML diagram showing the changed and added classes is in Figure 5.16.

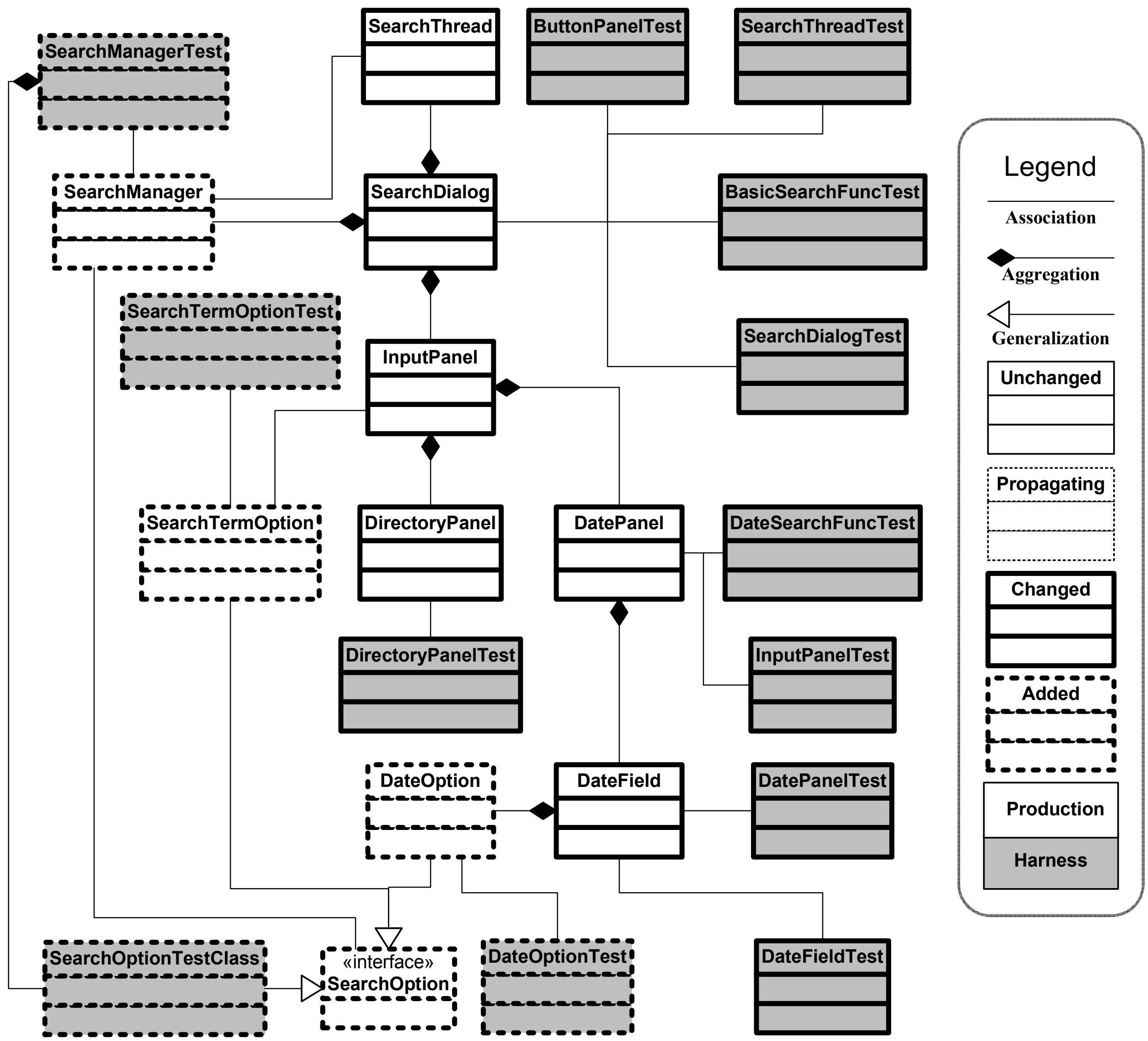


Figure 5.16 Change 5 Prefactoring

5.5.5 Actualization

The prefactoring prepared the code for the change very well. To actualize the change request, the programmer changed the `InputPanel` class and incorporated a new class, `CaseSensitiveOption` that implements the `SearchOption` interface through polymorphism. InputPanel added a check box to turn case sensitive searching on and off. It does this by swapping its `SearchTermOption` field for the

CaseSensitiveOption field. It also added a border around the recursive check box and the case sensitive check box in the GUI to organize it.

The added CaseSensitiveOption class is very similar to the SearchTermOption class, but it uses logic that includes the case of the search term and the file's name. A UML diagram showing the changed and added classes is in Figure 5.17.

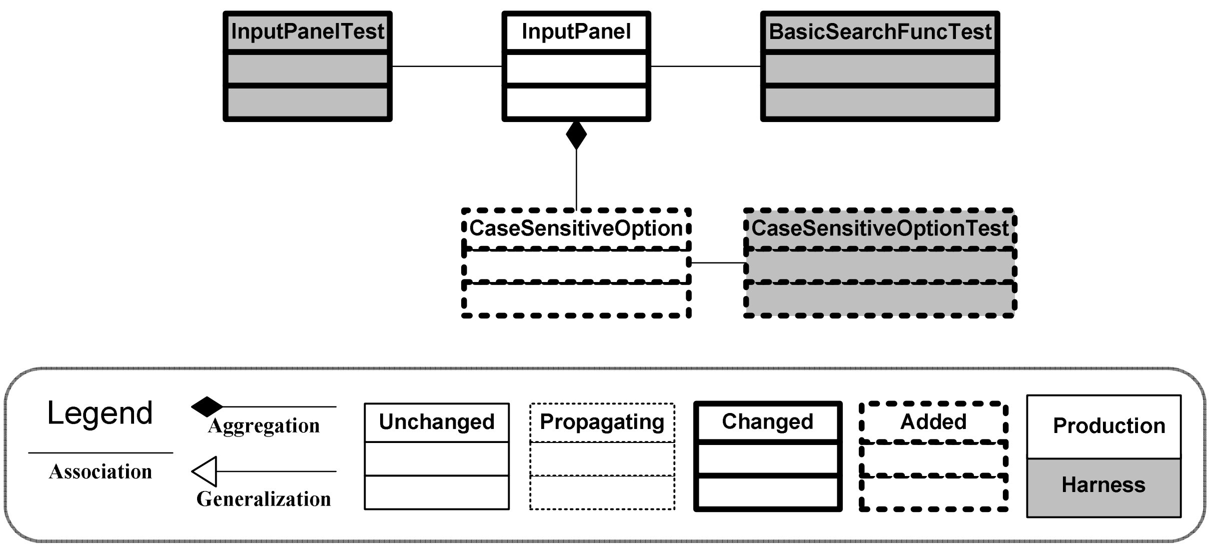


Figure 5.17 Change 5 Actualization

5.5.6 Postfactoring

The programmer addressed code smells that had developed over time during previous change requests. It is difficult to pinpoint exactly when these smells should have been addressed, but it is clear they need to be addressed now. For example, when the class InputPanel was extracted from SearchDialog during change request 2, it held all the input fields. During the change requests since then, DirectoryPanel was extracted and DatePanel was incorporated as a component; it now both holds other panels and instantiates objects of panels. To alleviate these code smells during this postfactoring and clarify its responsibility, BasicOptionsPanels was extracted from `InputPanel`; the fields moved and methods moved or impacted are:

Fields

- `JTextField inputBox`
- `JCheckBox recursiveBox`
- `JCheckBox caseSensitiveBox`
- `SearchManager searchManager`
- `SearchTermOption searchTerm`
- `CaseSensitiveOption caseSensitiveOption`

Methods

- `createInputBox()`
- `createOptionsPanel()`
- `switchToSearchState()`
- `getInputBox()`
- `actionPerformed()`

The classes `SearchTermOption` and `CaseSensitiveOption` had the same methods, but all 3 used different logic. A super class was extracted from them; this also allowed them to be swapped more easily by `BasicOptionsPanels` using their abstract class type. This super class extraction was necessary because of the change; it could have been done during prefactoring to prepare for the change. The field and methods moved to the `AbstractTermOption` are:

Field

- `String SearchTerm`

Methods

- `abstract setSearchTerm()`
- `insertUpdate()`
- `removeUpdate()`

A new test class for `BasicOptionsPanels` was extracted from `InputPanel` test. In addition the class extractions impacted 6 more harness code files see Figure

5.18. The class SearchFuncTestSetUp is part of the estimated impact set. It was not added to the changed set but was impacted during postfactoring.

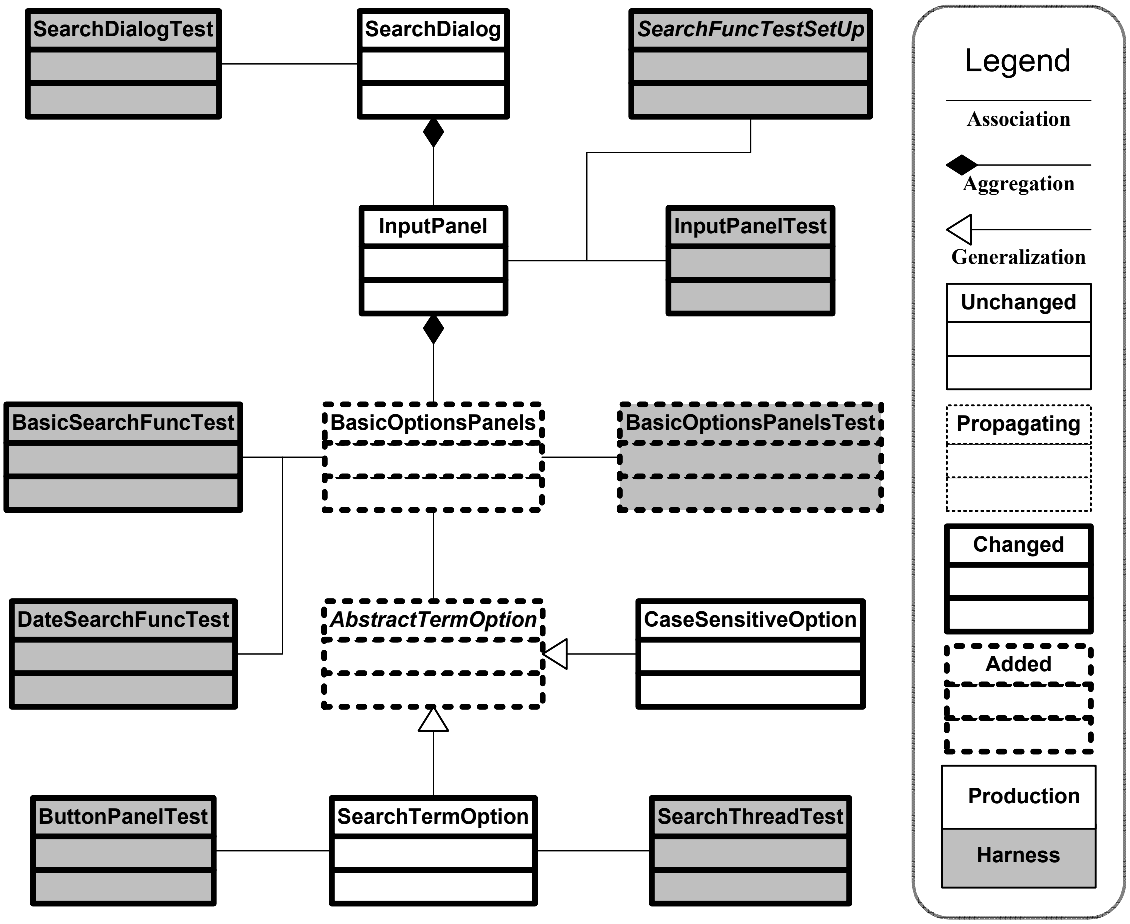


Figure 5.18 Change 5 Postfactoring

5.5.7 Verification

At the end of the change request all regression tests passed. The programmer followed the format of the previous change request and added a unit test for each added class. To test the SearchManager class the programmer also created a stub class SearchOptionTestClass and added it to the harness; it is a concrete implementation of the SearchOption interface. No unit test class was added for the abstract class AbstractTermOption; but both of the concrete implementations, SearchTermOption and CaseSensitiveOption have unit test classes. All new tests passed; no bugs were identified in this change. Table 5.11 shows the statement level coverage of the test harness for the code files added during this iteration.

Table 5.11 Change 5 Statement verification coverage of production code files

#	Code File	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	`AbstractTermOption`	7	6	85.7	0	0
2	`BasicOptionsPanels`	45	45	100.0	0	0
3	`CaseSensitiveOption`	4	4	100.0	0	0
4	`DateField`	69	64	92.8	0	0
5	`DateOption`	20	20	100.0	0	0
6	`DatePanel`	58	57	98.3	0	0
7	`DirectoryPanel`	53	44	83.0	0	0
8	`InputPanel`	36	36	100.0	0	0
9	`SearchDialog`	44	43	97.7	0	0
10	`SearchManager`	17	17	100.0	0	0
11	`SearchTermOption`	4	4	100.0	0	0
12	`SearchThread`	25	21	84.0	0	0

5.5.8 Conclusion

The programmer committed the updated code to the repository as a new baseline. The changed set had 1 fewer code files that the estimated impact set, see Table 5.12. `SearchFuncTestSetUp` was not changed until postfactoring. The programmer implemented the change by allowing code smells to develop, then addressed them by moving responsibility during postfactoring (section 5.5.6).

Table 5.12 Change 5 Summary

Number in Code files						
Inspected Concept Location	Estimated Impact Set	Changed Set	Added during			Total Project
			Pre	Act	Post	
0	16	15	8	2	3	1,133

5.6 Change Request 6 Extension Search

5.6.1 Initialization

This change request is: "Add the ability to search for files with specific extensions."

To help understand the change request, the programmer envisioned the following functionality for the change:

1. Add a check box to turn extension searching on and off
2. Add a text box for the user to enter file extensions
3. Add extension criteria to the search algorithm

5.6.2 Concept Location

The programmer extracted relevant concepts from the change request and using their intensions he converted them to following significant concepts:

- search by file extension
- add/remove from `SearchManager`
- search algorithm
- search
- files → file name

No concept location was needed for this change. This change request has similar requirements to change requests 4 and 5. The concept to location, the class to incorporate the new functionality 1 and 2, is `BasicOptionsPanels`. The programmer knew the code responsible for functionality 3, the search algorithm, did not contain the concept location because he refactored it during change request 5. The search algorithm is now modified dynamically by user selections and therefore was not impacted by this change.

5.6.3 Impact Analysis

The programmer started impact analysis by marking the code file containing the concept location, `BasicOptionsPanels`, Impacted in JRipples. The programmer visited and marked the following code files Impacted:

- `AbstractTermOption`, compares `AbstractFile` to the search term
- `SearchTermOption`, inherits from `AbstractTermOption`
- `CaseSensitiveOption`, inherits from `AbstractTermOption`
- `InputPanel`, contains a panel that errors are displayed in

The programmer then visited `AbstractFile`; it contains the methods `getFileNameWithoutExtension()` and `getExtension()`. These methods are all the search algorithm requires from `AbstractFile`, so it was marked Unchanged. The programmer wanted to duplicate the functionality from the year input field that was part of the date chooser added during change request 4; it shows the user if input is valid by coloring it green or invalid by coloring it red. The programmer visited the code files in the following order and marked them Propagating, they were not impacted, but lead to an impacted code file:

1. `DatePanel`
2. `DateField`
3. `JCalendar`

4. `JYearChooser`

JRipples marked `JSpinField` Next and the programmer visited and marked it Impacted because it only accepts integers, this change request requires it to also accept alphabetic characters.

The programmer then visited the harness code files in JRipples' Next set and marked them Impacted:

- `BasicOptionsPanelsTest`
- `CaseSensitiveOptionTest`
- `SearchTermOptionTest`
- `JSpinFieldTest`
- `TestConstants`

Finally, the programmer visited the 19 production code files and 20 harness code files in the Next set and marked them Unchanged, see Figure 5.19.

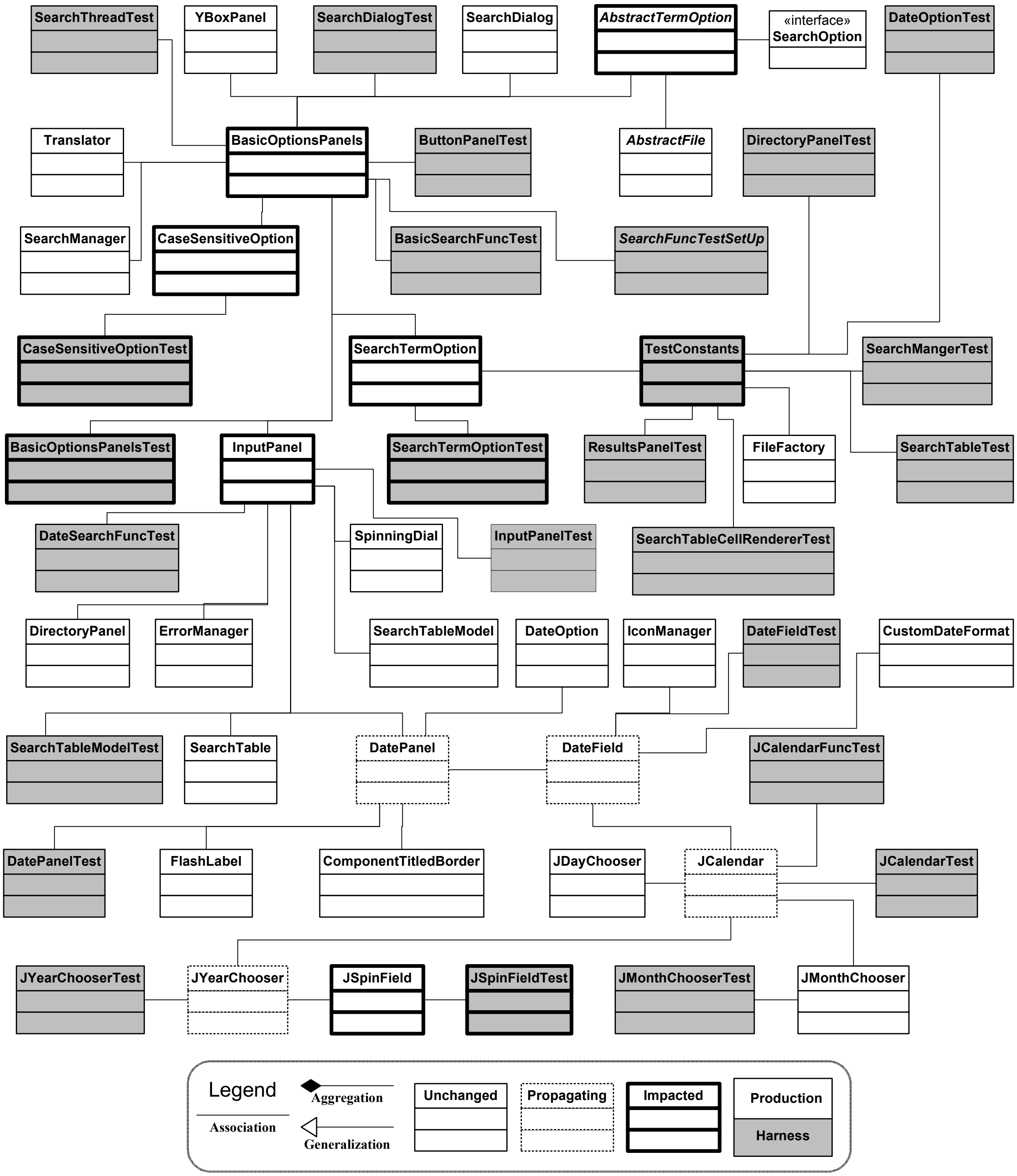


Figure 5.19 Change 6 Impact Analysis

5.6.4 Prefactoring

During impact analysis the programmer added `JSpinField` to the estimated impact set. This field colors the text green if the user input is valid and red if the user input is invalid as the user types. However, the `JSpinField` only accepts integer values. To make it easier to add the coloring feature for alphabetical values to this change request, a new class, `FeedbackField` was extracted from `JSpinField`. It extends `JTextField` and is only responsible for changing the color of the text, depending if the text is valid or invalid. To make `FeedbackField` work in general cases; the programmer added a nested interface, `InputListener`. `InputListener` has 1 method, `isInputValid()` that allows implementing classes to define what is valid and invalid input. The field and methods of `JSpinField` impacted by the extraction are:

Fields

- `JTextField textField`
- `Color darkGreen`

Methods

- `setValue()`
- `setMaximum()`
- `setHorizontalAlignment`
- `setFont()`
- `setForeground()`

A test class `FeedbackField` was extracted from `JSpinFieldTest`. It also had tests added for the new methods.

5.6.5 Actualization

To actualize the change request, the programmer incorporated a new supplier of `BasicOptionsPanels` that extends `YBoxPanel` called `ExtensionPanel`. The class contains a `JCheckBox`, `FeedbackField` and `FlashLabel`. This class adds the components to the GUI for the user to enter extensions.

The programmer also added a class that implements the SearchOption interface, ExtensionOption that is added to the list of SearchOption objects in SearchManager when an extension search is enabled. ExtensionOption's primary responsibility is to check an AbstractFile's extension against the set of user entered extensions and return true if it is.

The programmer added the responsibility of changing between classes that extend AbstractTermOption to compare an AbstractFile's name to a search term to BasicOptionsPanels. When an extension search is enabled, BasicOptionsPanels will change between 4 different implementations of the AbstractTermOption class. There were 2 classes to do this at the beginning of this change request, which compare the search term to the file's name including the extension. The programmer created 2 new classes that compare the file's name without the extension to the search term, SearchTermWithoutExtensionOption and CaseSensitiveWithoutExtensionOption that extend AbstractTermOption.

Additionally, the programmer added a FocusListener to FeedbackField to change the text color to the default when the field has lost focus.

The test classes, ExtensionSearchFuncTest, ExtensionOptionTest and ExtensionPanelTest were added by the programmer. FeedbackFieldTest and BasicOptionsPanelsTest were changed. Two new harness files for use in testing the production code related to extensions were added, testFile.log and testFile.test that are the same as testFile.txt added in change 2, but with different extensions. Final objects of type AbstractFile corresponding to these files were added to the class

TestConstants. A UML diagram showing the changed and added classes is in Figure 5.20.

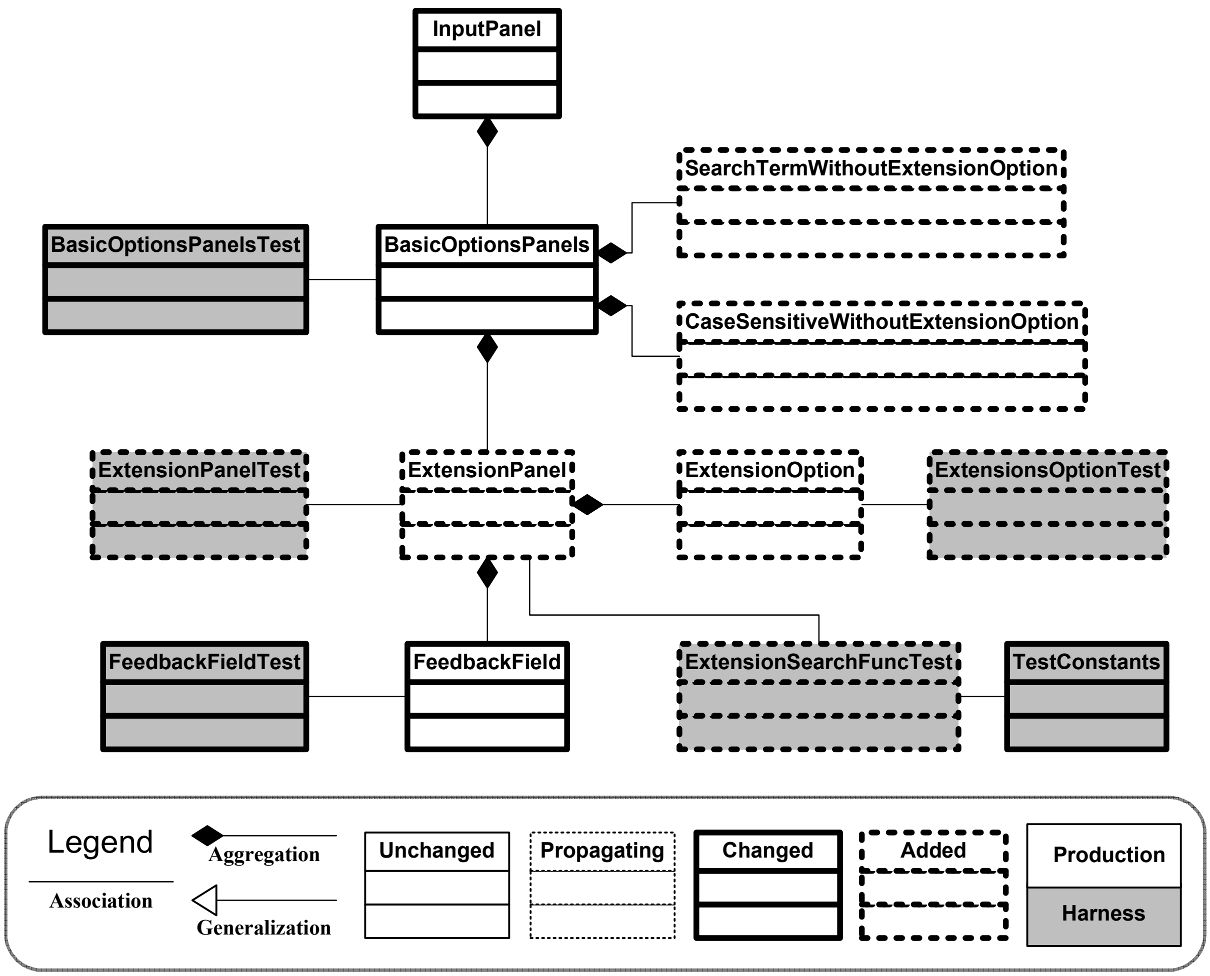


Figure 5.20 Change 6 Actualization

5.6.6 Postfactoring

After actualization the change request functionality worked, but the method in BasicOptionsPanels, swapSearchTermOptions() that switched between the 4 classes that extend AbstractTermOption was confusing and would be difficult to change in the future. The responsibility to listen to one JCheckBox and call swapSearchTermOptions() to switch between object that inherit from AbstractTermOption had grown and was spread across 2 classes, BasicOptionsPanels and ExtensionPanel. Further, the 2 classes created during actualization that inherit from AbstractTermOption, SearchTermWithoutExtensionOption and CaseSensitiveWithoutExtensionOption, had long and confusing names and very similar responsibility. The programmer decided that instead of having 4 different AbstractTermOption classes, there should be 1 class that listens to the 2 fields of type JCheckBox and uses polymorphism to switch between the compare criteria. This simplified the responsibility and combined it into 1 code file, SearchTermOption this made it easier for the programmer to handle switching between searches with and without extensions and made the code easier to understand. The super class and 3 other AbstractTermOption classes would all be merged into SearchTermOption. Additionally, Action Listener would be extracted from BasicOptionsPanels and ExtensionPanel to this code file.

The programmer changed the ExtensionOption's method, setExtensions(), which parses the user entered String into an array of String extensions, to a regular expression algorithm. The rest of the refactoring was renaming fields in FeedbackField and updating Javadoc in TestConstants. A UML diagram showing the changed and added classes is in Figure 5.21.

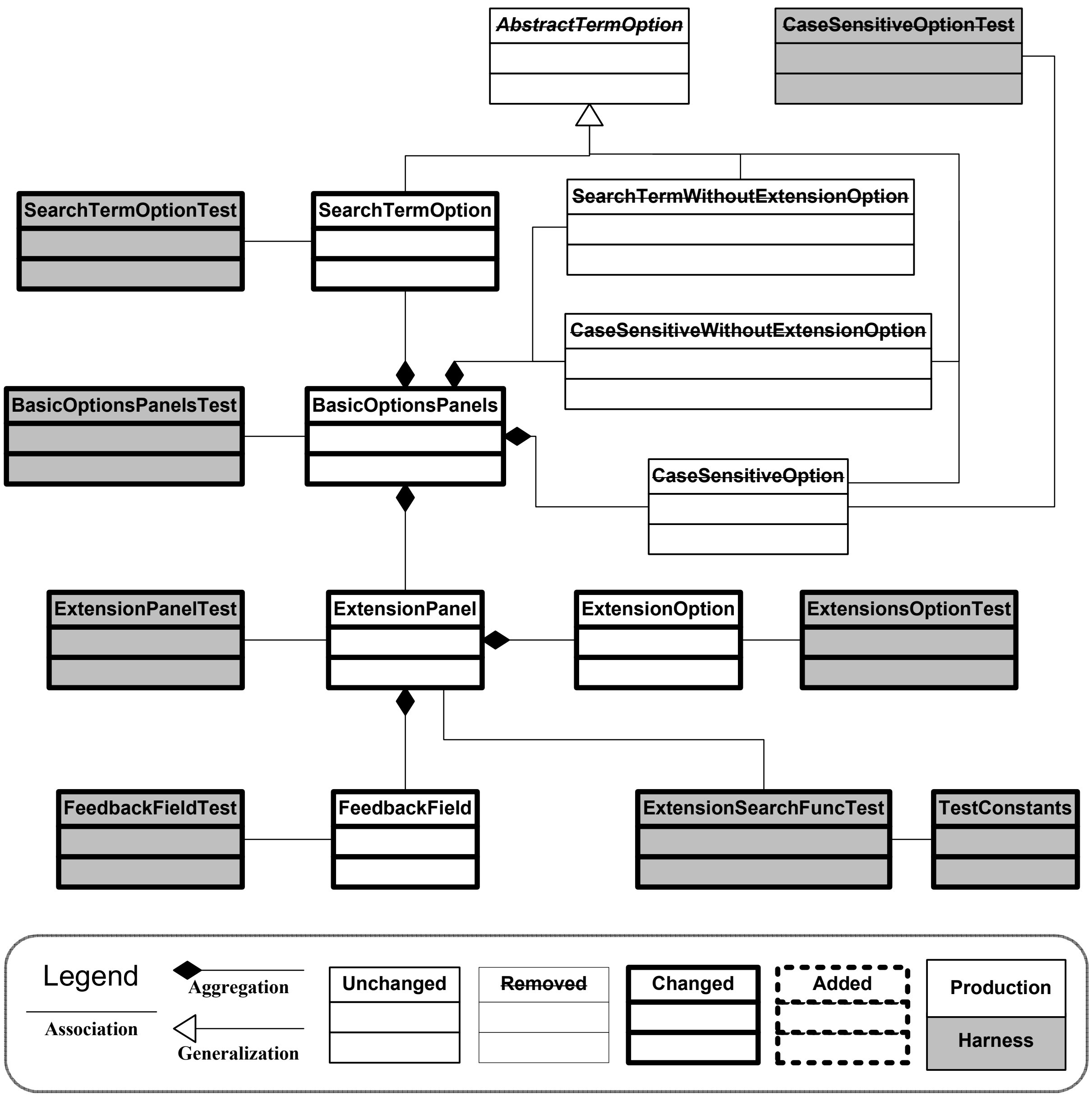


Figure 5.21 Change 6 Postfactoring

5.6.7 Verification

The test suite exposed 3 bugs during the change request, a forth bug was discovered through code inspection. Two of these bugs were part of the current change request and were fixed; the other 2 were added to the backlog.

While writing the test class for the `SearchTermOption` code file during postfactoring, the programmer found a bug in the `insertUpdate()` method. The bug was found by running the test, `testInsertUpdate()` from the `SearchTermOptionTest` class. The method `insertUpdate()` throws an exception if an empty string is input in the object of type `Document` the method listens to. This was resolved by adding a check for an empty `String` to the method.

The programmer found the second bug in `SearchTermOption` also, with the test `testActionPerformedCaseSensitiveBox()` from the `SearchTermOptionTest` class. If a case sensitive search is enabled, disabled and enabled, without changing the search term, the case of the search term would be lost. To fix the bug, the programmer added a field of type `String` to `SearchTermOption` that stores the term with case, so the case can be recovered when switching between case sensitive searches.

During impact analysis the programmer visited the `DatePanel` class; during this visit the programmer realized that the `datePanelSetEnabled()` method did not remove the `DateOption` object from the `SearchManager`. This means that if a date is entered and the date `JCheckBox` is unchecked, a date search will still be performed. This is the opposite of what a user would expect, but a there is an easy workaround; just delete the date. This bug was given a priority 3, some functionality is impaired, but a workaround can be found, therefore a change request was added to the backlog.

After prefactoring all the regression tests passed, however, during postfactoring 1 regression test, `testSetMonth()` from `JDayChooserTest`, failed. The programmer investigated this further and discovered the test will fail if run on the last day of the month if the next month has fewer days than the current month. The programmer did a test through user intervention and found that the bug did not affect the program's functionality. Therefore, a priority 4, minor problem not involving primary functionality, change request was added to the backlog to fix this bug. Table 5.13 shows the statement level coverage of the test harness for the code files added during this iteration.

Table 5.13 Change 6 Statement verification coverage of production code files

#	Code File	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	BasicOptionsPanels	38	38	100.0	0	0
2	ExtensionOption	20	20	100.0	0	0
3	ExtensionPanel	36	36	100.0	0	0
4	FeedbackField	42	42	100.0	0	0
5	InputPanel	37	37	100.0	0	0
6	JDayChooser	142	133	93.7	1	1
7	JSpinField	61	51	83.6	0	0
8	SearchTermOption	38	37	97.4	0	2

5.6.8 Conclusion

The programmer committed the updated code to the repository as a new baseline. The changed set was 5 code files less than the estimated impact set, see Table 5.14. All 5 of these code files were impacted during postfactoring. As in change 5 (section 5.5) the programmer simplified the change by allowing code smells to develop then addressed them during postfactoring. Also during postfactoring he merged 4 production code files into another during postfactoring and 1 harness code file into another (section 5.6.6), which removed 5 code files from the project.

Table 5.14 Change 6 Summary

Number in Code files						
Inspected Concept Location	Estimated Impact Set	Changed Set	Added during			Total Project
			Pre	Act	Post	
0	11	6	2	7	(5)	1,137

5.7 Change Request 7 Properties Search

5.7.1 Initialization

This change request is: "Add options to search for files based on their properties."

To help understand the change request, the programmer envisioned the following functionality for the change:

1. Add 4 check boxes to turn searching for each file type on and off
2. Add the 4 file types criteria to the search algorithm

The programmer extracted relevant concepts from the change request and using their intensions he converted them to following significant concepts

- archives and read only files
- search for a file type
- add/remove from `SearchManager`
- 4 file types →
  - hidden files
  - directories
  - read-only
  - archives

5.7.2 Concept Location

No concept location was needed for this change. This change request is similar to change request 6. The concept to location is the same as change request 6, the class to incorporate the new functionality 1, is `BasicOptionsPanels`. The programmer knew the code responsible for functionality 2, the search algorithm, did not contain the concept location just as in change request 6.

5.7.3 Impact Analysis

The programmer started impact analysis by marking the code file containing the concept extension, `BasicOptionsPanels`, Impacted in JRipples.

- `InputPanel`, `createOptionsPanel()` will need to be changed.
- `AbstractFile`; needs a method to check if an object of it is read-only
- `BasicOptionsPanelsTest`
- `InputPanelTest`
- `AbstractFileTest`
- `TestConstants`

Changes to the `AbstractFile` class can have a large impact on muCommander; JRipples added 307 code files to the Next set when it was marked Impacted. The programmer decided not to visit all of the Next classes; the method to add to this class is a non-abstract `boolean` getter this should not affect any implementing or dependent class.

5.7.4 Prefactoring

No prefactoring was done during this change. The programmer did not see any prefactoring that would make the change easier. That is not to say that prefactoring could not have been done; but rather that for this change the programmer decided to do the actualization and then perform all refactoring during the postfactoring stage.

5.7.5 Actualization

During actualization, the programmer incorporated a new supplier of `BasicOptionsPanels` that extends `JPanel` and holds the 4 fields of type `JCheckBox` for properties searches. This class, `PropertiesPanel`, has a method to enable and disable the `JCheckBox` fields. `PropertiesPanel` implements the `ActionListener` interface; it listens to the archive and directory `JCheckBox` fields. If one of these boxes is checked the other is disabled, because it is impossible for a file to be both. It also creates objects of 4 new classes that implement the `SearchOption` interface. To accommodate the new panel in the GUI, `InputPanel` was changed to modify the GUI layout. A test class, `PropertiesPanelTest`, was added for this class. The fields and methods of the class are:

Fields	Methods
• `JCheckbox archiveBox`	• `archiveBoxSetEnabled()`
• `JCheckbox directoryBox`	• `directoryBoxSetEnabled()`
• `JCheckbox hiddenBox`	• `setEnabled()`
• `JCheckbox readOnlyBox`	• `actionPerformed()`

The 4 new classes that implement the `SearchOption` interface in `PropertiesPanel` are:

- `ArchiveOption`
- `DirectoryOption`
- `HiddenOption`
- `ReadOnlyOption`

They were also added through polymorphism and they add themselves to the `SearchManager` object when their corresponding `JCheckBox` field in `PropertiesPanel` is selected. They each have a `SearchManager` field, the `actionPerformed()` method from the `ActionListener` interface and the `meetsCriteria()` method from the `SearchOption` interface that returns true, if an `AbstractFile` sent to it is an archive, directory, hidden file or read-only file. The programmer added `ArchiveOptionTest`, `DirectoryOptionTest`, `HiddenOptionTest` and `ReadOnlyTest`, test classes for these classes.

The `AbstractFile` class had methods `isArchive()`, `isDirectory()` and `isHidden()`; but it did not have an `isReadOnly()` method. The programmer added the method and added a test for it to `AbstractFileTest`. This part of the change impacted a class not found during impact analysis, `ProxyFile`. `ProxyFile` is a concrete implementation of `AbstractFile` that must override all of `AbstractFile`'s methods, so when the programmer added the method `isReadOnly()` to `AbstractFile`, a test in `ProxyFileTest` failed. To correct this the programmer added an overridden method `isReadOnly()` to `ProxyFile`.

Finally, 3 new harness files were added to the project, an archive file, a hidden file and a read-only file. The programmer then added fields corresponding to them to the `TestConstants` class. A UML diagram showing the changed and added classes is in Figure 5.22.

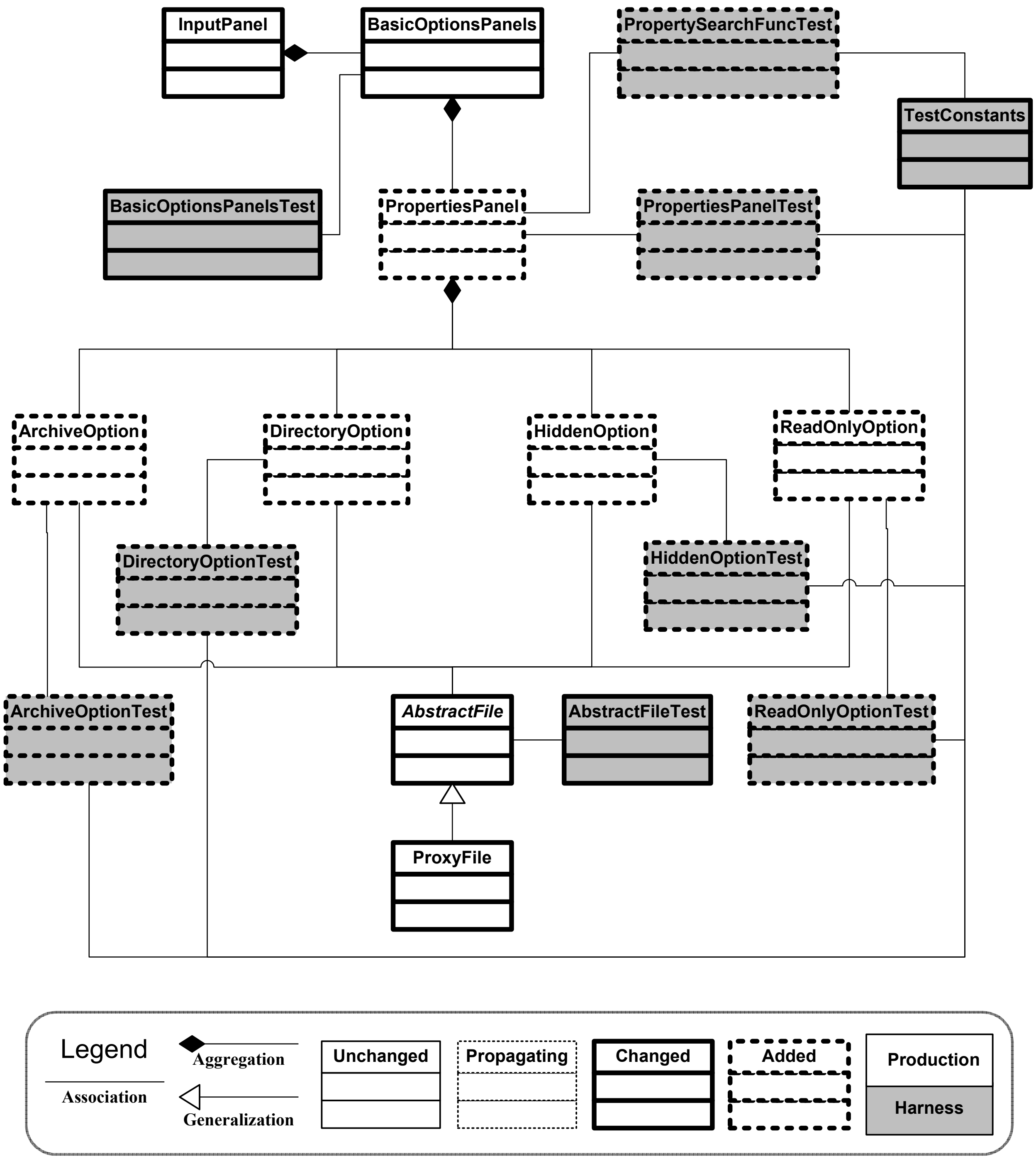


Figure 5.22 Change 7 Actualization

5.7.6 Postfactoring

During actualization the programmer caused code smells to develop in `PropertiesPanel`. The responsibility to disable the archive `JCheckBox` when the directory `JCheckBox` is selected and vice-versa is misplaced. The programmer extracted a new class from `PropertiesPanel`, called `SearchOptionBox`. It adds the responsibility of an antonym `SearchOptionBox`. When a `SearchOptionBox` is selected, it disables a registered antonym box.

The programmer placed the responsibility to add and remove the 4 code files, `ArchiveOption`, `DirectoryOption`, `HiddenOption` and `ReadOnlyOption` that implement `SearchOption` in these classes. This was also misplaced, there is duplicated because of it in these 4 classes, so the programmer extracted this responsibility to `SearchOptionBox`. This class now is solely responsible for the actions of selecting the `JCheckBox`. This left the 4 classes that implement `SearchOption` with 1 method, `meetsCriteria()`. These classes could have been made into anonymous classes, but the programmer chose to keep them in their own files, because it makes the code clearer in his opinion. The fields and methods of `SearchOptionBox` are:

Fields

- `SearchOption searchOption`
- `SearchManager searchManager`
- `SearchOptionBox antonym`

Methods

- `addAntonym()`
- `removeAntonym()`
- `hasAntonym()`
- `getAntonym()`
- `enableOption()`
- `setEnabled()`
- `actionPerformed()`

The classes `InputPanel` and `BasicOptionsPanels` shared the responsibility of laying out the GUI parts dealing with search options such as recursive searches, extension searches, property searches and date searches. After actualization it stood out that this was not clearly organized. The programmer extracted `OptionsPanel` from `InputPanel` to layout all of GUI classes that contain search options. One of these classes, `BasicOptionsPanels`, had the `JTextField` that contains the search term. The programmer does not consider the search term a search option, so it was extracted to a new class `SearchTermPanel`. The fields and methods of `OptionsPanel` are:

Fields

- `BasicOptionsPanel basicOptionsPanel`
- `ExtensionPanel extensionPanel`
- `PropertiesPanel propertiesPanel`
- `DatePanel datePanel`
- `JPanel topPanel`

Methods

- `createPanel()`
- `createTopPanel()`
- `addComponent()`
- `setEnabled()`

This left `InputPanel` responsible for the layout of 4 objects of type `JPanel`. Three of these are separate production code classes, `DirectoryPanel`, `SearchTermPanel` and `OptionsPanel`. The fourth `JPanel` holds a static `JLabel`, a `JLabel` that displays search option errors and an icon that is animated when a search is running. This panel is not significant enough for its own class; therefore it is created in a method, `createLabelPanel()` in `InputPanel`.

This refactoring resulted in broken contracts and propagated to 9 code files not in the changed set or the estimated impact set. The only one of these that is production code is `SearchDialog` it has a method call that to request the cursor be placed in; it requires a call to `SearchTermPanel` to get the object that the cursor will be placed in. It is an anti-pattern that the programmer would like to remove, but the programmer did not think the anti-pattern was worth the effort required to remove it. The other code files not in the changed set were all part of the harness see Figure 5.23.

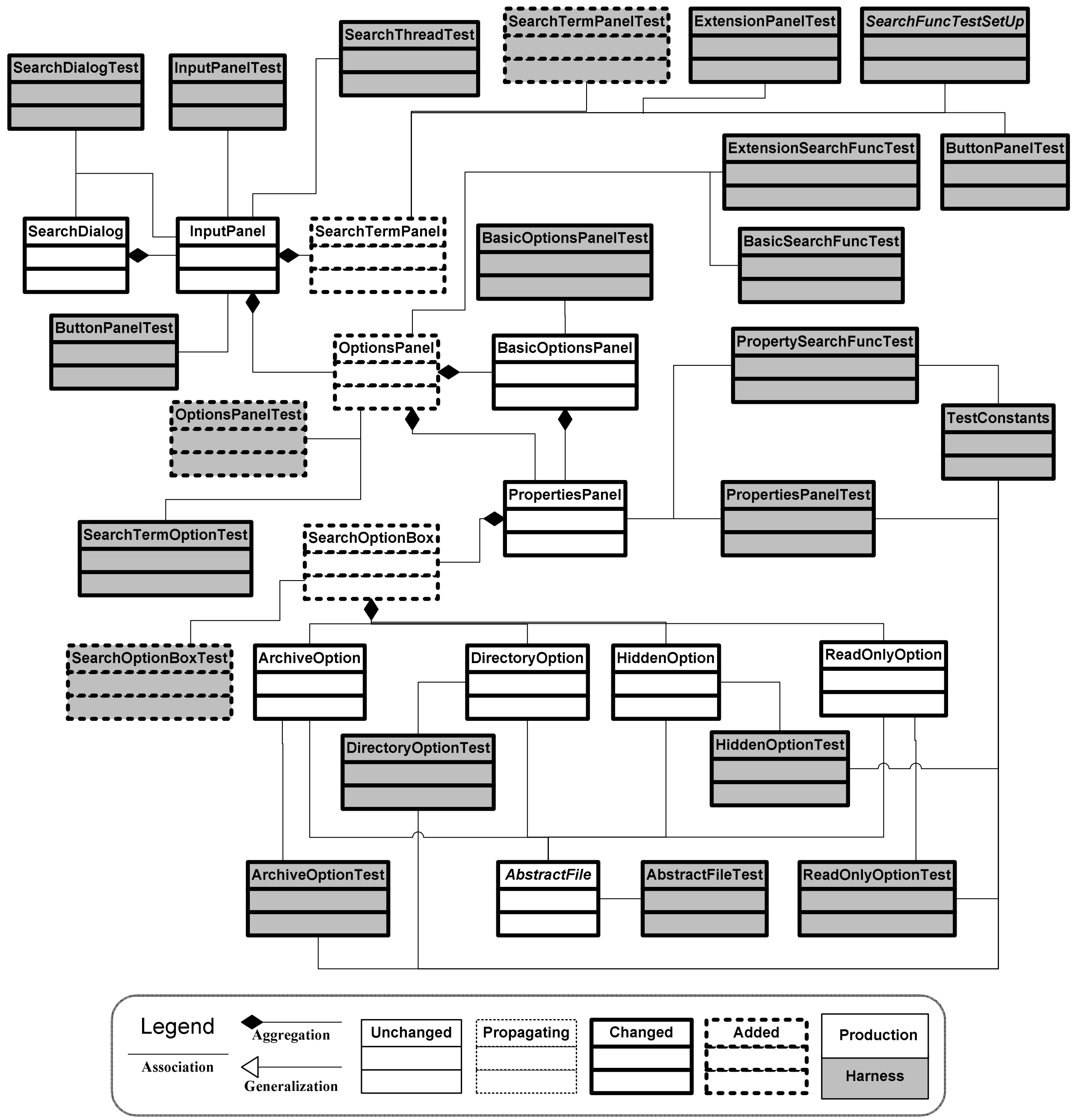


Figure 5.23 Change 7 Postfactoring

The programmer did not plan to extract SearchTermPanel and OptionsPanel classes at the start of the change. However, after the change code smells were present in BasicOptionsPanels and InputPanel that needed to be dealt with. The programmer made the mistake of thinking the harness code files had similar dependencies as the production code files they test, which is not the case. The harness code files have more dependencies than the production code files they test because the tests not only have dependencies of the class being tested, but also dependencies of the dependencies. A test class may need objects of a few levels of dependencies. Additionally, the test's assertions may require an object of a dependency of the class being tested, especially in the case of methods with void return types.

5.7.7 Verification

After actualization and postfactoring all regression tests passed. The programmer found 3 bugs during the change; 2 during actualization and 1 during postfactoring. The first bug was found during actualization, the test, testSetEnabled() in the PropertiesPanelTest code file failed when it was written. The programmer added a call to the super method in the overridden method setEnabled() in PropertiesPanel then the test passed.

The programmer discovered a bug created during a previous change request during actualization. When the programmer investigated the failed test, testSetEnabel(), he ran a manual intervention test. During this test he discovered that, if a directory to search in is chosen with the GUI file chooser, the search directory is not updated. A bug level 3 bug was added to the backlog, because there is an easy workaround, just click on the directory field before starting a search, this forces the text in the directory field to be read in and the search to execute correctly. Table 5.15 shows the statement level coverage of the test harness for the code files added during this iteration.

Table 5.15 Change 7 Statement verification coverage of production code files

#	Code File	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	`AbstractFile`	233	170	73.0	0	0
2	`ArchiveOption`	1	1	100.0	0	0
3	`BasicOptionsPanel`	13	13	100.0	0	0
4	`DirectoryOption`	1	1	100.0	0	0
5	`DirectoryPanel`	53	44	83.0	1	1
6	`HiddenOption`	1	1	100.0	0	0
7	`InputPanel`	27	27	100.0	0	0
8	`OptionsPanel`	43	43	100.0	0	0
9	`PropertiesPanel`	24	24	100.0	2	2
10	`ProxyFile`	64	54	84.4	0	0
11	`ReadOnlyOption`	1	1	100.0	0	0
12	`SearchDialog`	44	43	97.7	0	0
13	`SearchOptionBox`	23	23	100.0	0	0
14	`SearchTermPanel`	11	11	100.0	0	0

The third bug the programmer discovered during postfactoring. The tests `testArchiveBoxSetEnabled()` and `testDirectoryBoxSetEnabled()` both failed after the class `SearchOptionBox` was extracted from `PropertiesPanel`. During the class extraction the programmer neglected to add the lines `archiveBox.addAntonym(directoryBox);` and `directoryBox.addAntonym(archiveBox);` to the `PropertiesPanel` constructor. The programmer added the lines and finished postfactoring.

5.7.8 Conclusion

The programmer committed the updated code to the repository as a new baseline. The changed set and the estimated impact set were equal, see Table 5.16. However, `ProxyFile` was added to the changed set during actualization it was overlooked by the programmer during impact analysis. `InputPanelTest` was not impacted until postfactoring and is therefore not part of the changed set. Also during postfactoring 9 code files that were not part of the estimated impact set were impacted (section 5.7.6). This was because the programmer decided to do more refactoring than planned because the responsibilities of `SearchDialog` had become unclear; this affected 1 production code file and 8 harness code files.

Table 5.16 Change 7 Summary

Number in Code files						
Inspected Concept Location	Estimated Impact Set	Changed Set	Added during			Total Project
			Pre	Act	Post	
0	7	7	0	11	6	1,154

5.8 Change Request 8 File Chooser Bug

5.8.1 Initialization

This change request is a bug from the defect log: “Choosing a directory with the file chooser does not update the search directory.”

5.8.2 Concept Location

The programmer extracted significant concepts from the change request and using their intensions he converted them to following significant concepts:

- directory
- file chooser
- search directory

No concept location was needed for this change. This bug was identified during change request 7 through a code inspection; the concept extension is in the `DirectoryPanel` code file.

5.8.3 Impact Analysis

Impact analysis also was not necessary for this change request. The programmer was familiar with the concept extension. He knew the change request would propagate to no other production code files. He included 2 harness code files `DirectoryPanelTest` and `BasicSearchFuncTest` to add tests to prevent this bug from reoccurring.

5.8.4 Prefactoring

The programmer extracted a method called `directoryFieldUpdate()` from the existing `keyReleased()` method in `DirectoryPanel`. All of the body of `keyReleased()` was extracted to the new method. He did this because the `KeyListener` interface and its `keyReleased()` method will be replaced during actualization to fix the bug. The programmer also added a test for the new method, to `DirectoryPanelTest`.

5.8.5 Actualization

To actualize the change request, the programmer replaced the `KeyListener` interface in `DirectoryPanel` with a `DocumentListener` interface. This interface initiates an event if the text in a JTextField is changed regardless of the source; the KeyListener interface only initiated events if the user types a key with the KeyListener when the directory chooser updated the text field, there was no event.

The programmer then added tests to DirectoryPanelTest for the DocumentListener interface's methods and deleted the test for the keyListener() method. He added a test to BasicSearchFuncTest that uses the GUI file chooser to select a directory to search and asserts that the selected directory is the current search directory.

5.8.6 Postfactoring

No Postfactoring was necessary for this change request.

5.8.7 Verification

After actualization and postfactoring all regression tests passed Table 5.17 shows the test coverage of DirectoryPanel after the change request.

Table 5.17 Change 8 Statement verification coverage of production code files

#	Code File	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	DirectoryPanel	55	54	98.2	0	0

5.8.8 Conclusion

The programmer committed the updated code to the repository as a new baseline. The changed set include was the same as the estimated impact set, see Table 5.18.

Table 5.18 Change 8 Summary

Number in Code files						
Inspected Concept Location	Estimated Impact Set	Changed Set	Added during			Total Project
			Pre	Act	Post	
0	3	3	0	0	0	1,154

5.9 Change Request 9 Date Search Bug

5.9.1 Initialization

This change request is a bug from the defect log: "The `DateOption` is not removed from the `SearchManager` when it is disabled."

5.9.2 Concept Location

The programmer extracted significant concepts from the change request and using their intensions he converted them to following significant concepts:

- `DateOption`
- not removed
- `SearchManager`
- disabled

No concept location was needed for this change. This bug was identified during change request 6; the concept extension is in the `DatePanel` code file.

5.9.3 Impact Analysis

Impact analysis also was not necessary for this change request. The programmer was familiar with the concept extension. He knew the change request would propagate to `DateField` and `DateOption`, see Figure 5.24. He also included the following harness code files to add tests to prevent this bug from reoccurring:

- DatePanelTest
- DateFieldTest
- DateOptionTest
- DateSearchFuncTest

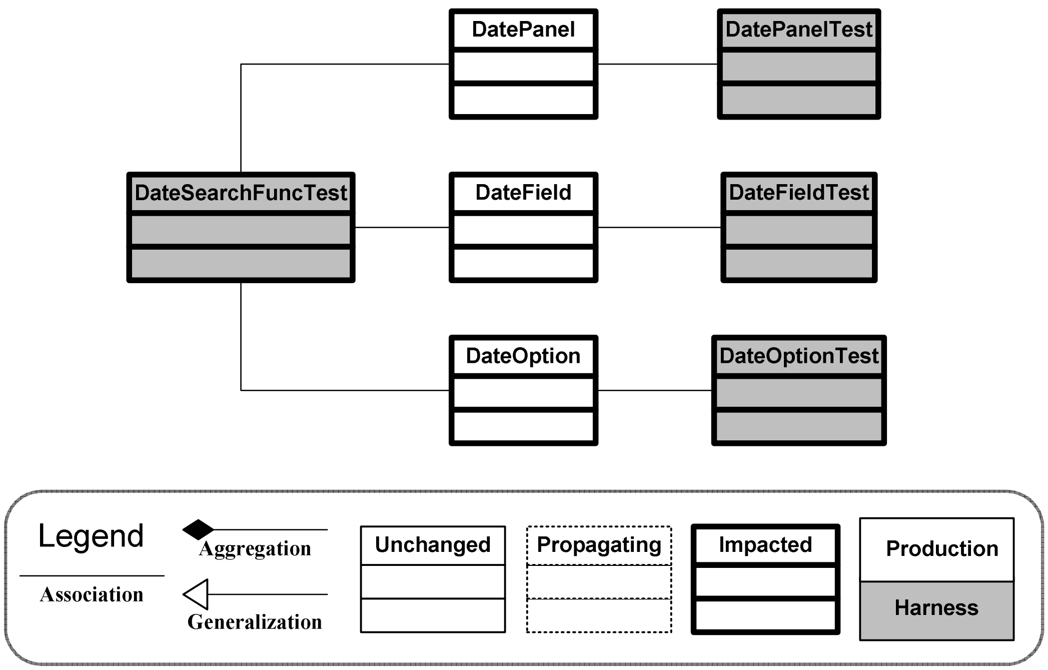


Figure 5.24 Change 9 Impact Analysis

5.9.4 Prefactoring

No prefactoring was necessary for this change request.

5.9.5 Actualization

To actualize the change request, the programmer added the `ActionListener` interface to the `DateOption` class. He then added the `DateOption` objects initialized in `DatePanel` as listeners to the `dateBox` field. This will add and remove objects of this class to the set of `SearchOption` objects in `SearchManager` as appropriate. The change propagated to `DateField`, which had a redundant method call in its `focusLost()` method that was adding the `DateOption` object back into `SearchManager`.

The programmer then changed tests in `DatePanelTest` and `DateOptionTest` to test the new contracts. He then added a test to `DateSearchFuncTest` that enables and disable a date search and asserts that the `DateOption` objects are removed from `SearchManager`. The change request did not propagate to the `DateFieldTest` harness code file, its tests still passed after the redundant call was removed from `DateField`.

5.9.6 Postfactoring

No Postfactoring was necessary for this change request.

5.9.7 Verification

After actualization and postfactoring all regression tests passed. Table 5.19 shows the test coverage of the changed production code files after the change request.

Table 5.19 Change 9 Statement verification coverage of production code files

#	Code File	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	`DatePanel`	62	61	98.4	0	0
2	`DateField`	68	64	94.1	0	0
3	`DateOption`	21	21	100.0	0	0

5.9.8 Conclusion

The programmer committed the updated code to the repository as a new baseline. The changed set was less than the estimated impact set, see Table 5.20. The programmer decided the test in `DateFieldTest` still sufficiently tested the changed code and that the tests in the 3 changed harness files would prevent the bugs return.

Table 5.20 Change 9 Summary

Number in Code files						
Inspected Concept Location	Estimated Impact Set	Changed Set	Added during			Total Project
			Pre	Act	Post	
0	7	6	0	0	0	1,154

5.10 Build

At the end of the iteration, the programmer thoroughly tested muCommander by running all the regression tests. He confirmed all tests passed and was confident that no new bugs were introduced during the iteration. He then created a special baseline, which he used to create a version of the program without the harness code for release to the users. This completed the iteration and release. There were 40 new code files added and 22 code files changed in muCommander, see Figure 5.25.

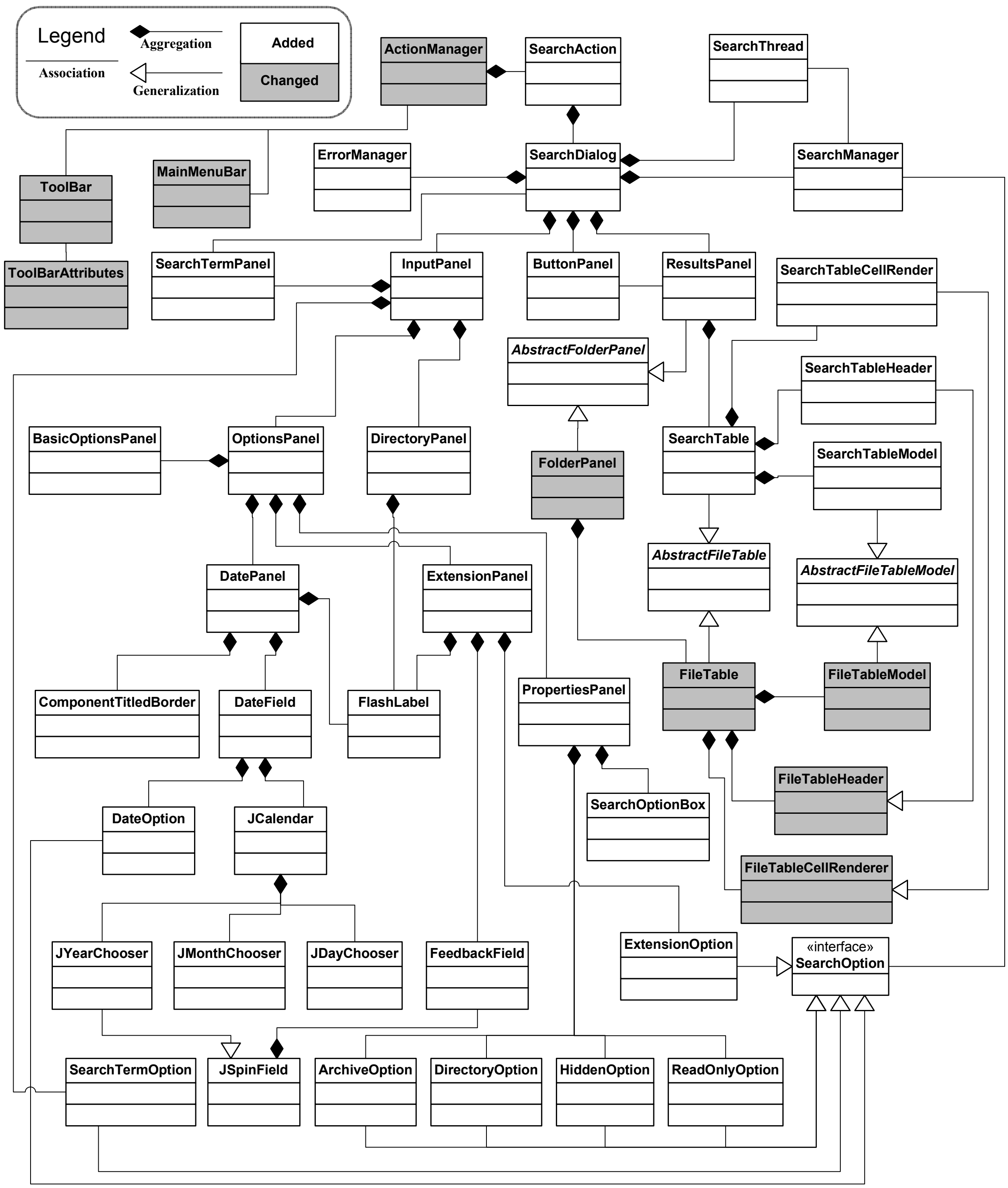


Figure 5.25 SIP Iteration

The iteration added search functionality to muCommander, see Figure 5.26.

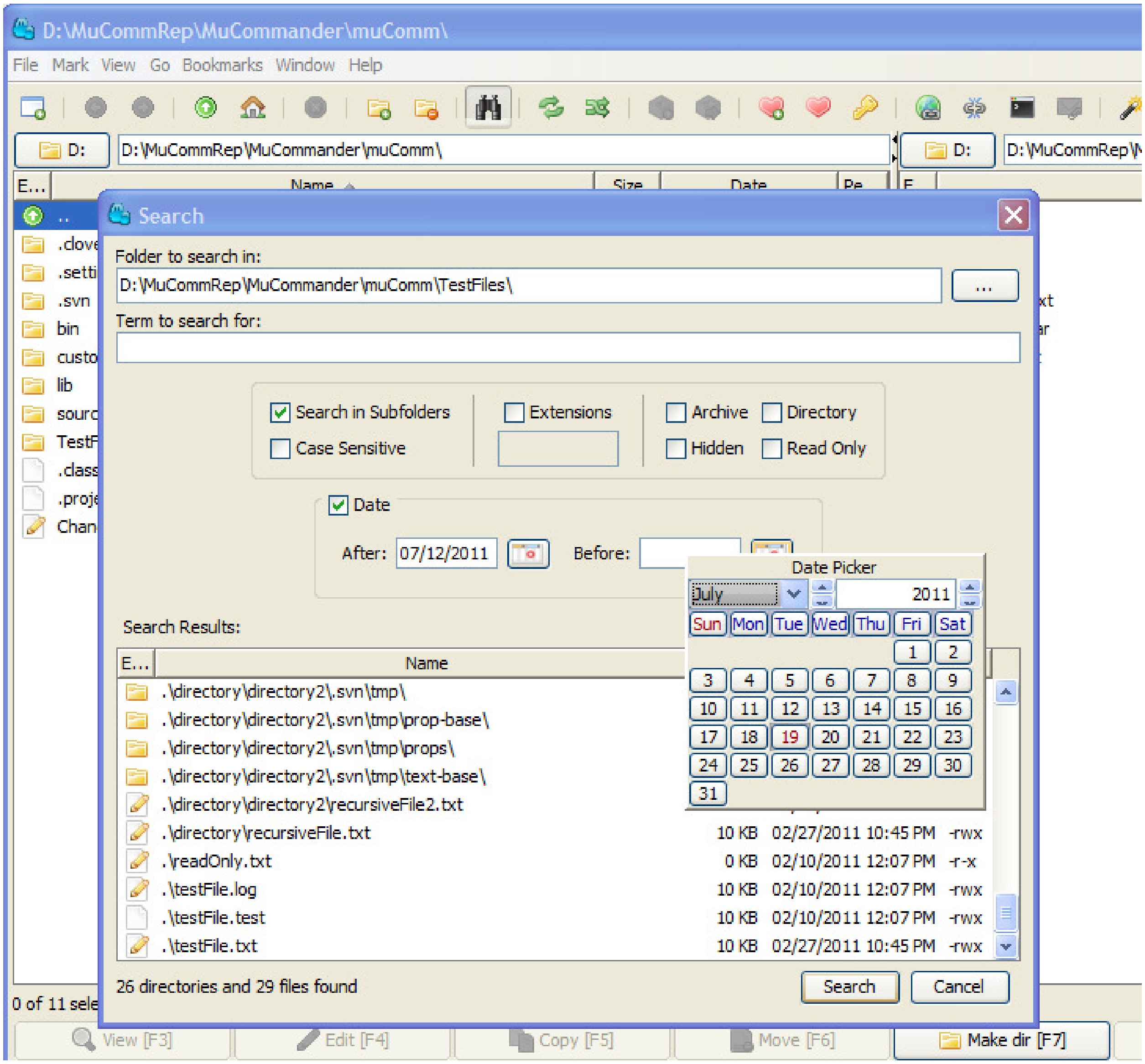


Figure 5.26 Search Feature

The programmer did not complete the entire iteration backlog. Three of the 10 changes from the iteration backlog were returned to the product backlog, see Table 5.21. The programmer completed the iteration before the iteration backlog was empty because he believed that the feature was in a high-quality state and his user were ready for the feature.

Table 5.21 Product Backlog after Iteration

#	Title	User Story
1	Size Search	Add the ability to search for a file by its size.
2	Regular Expression Search	Add capability to search by a regular expression.
3	Lucene Search	Incorporate the Apache Lucene search.
4	JDayChooserTest Bug	The test testSetMonth() fails on last day of month, if next month has fewer days

Chapter 6
Discussion

This chapter presents the programmer's experience in the phases of SC. It then presents the exceptions to SIP the programmer made during the iteration. Next reasons solo programmers should use SIP are discussed. After that the amount of rework required and criticism of the process are discussed. Then the technologies used in the iteration are reviewed. Finally, the threats to this thesis's validity are discussed.

6.1 Concept Location

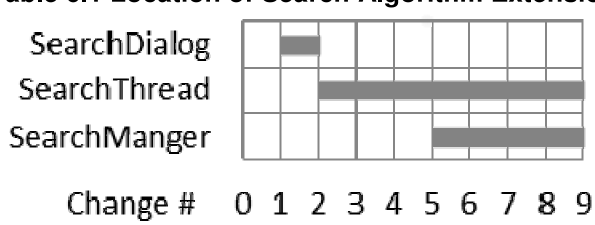
Performing multiple changes on a single program presented an opportunity to look at how a concept extension moved over the iteration. At the beginning of the iteration, the concept extension "search algorithm" was not explicitly present in the code; it was an implicit concept. It was implemented in change 1, but it was a trivial concept that didn't require its own class; it was part of `SearchDialog`. The algorithm was simply a for loop that added files to a set if the file's name contained the search term; it was simple and met the needs of the feature.

In prefactoring of change 2, the search algorithm was extracted to its own class called `SearchThread`. Then during actualization, `SearchThread` was replaced with a more complex class that created a separate thread for the algorithm to run in and also added recursive ability. When it came across a directory it called itself to search the directory. This algorithm was more complicated but at its core it still just checked if the file's name contained the search term.

The next large change to the search algorithm came in change 4, which added the capability to search by a file's last modified date. The programmer modified the search algorithm, now if the file's name contained the search term, the algorithm then checked if the date search feature was turn on and if so, checked if the file's modified date was in the search range. The algorithm became more complicated and this introduced a code smell, but the programmer didn't refactor the algorithm, because the code was still understandable and the section was small.

Change 5 was to add the ability to match a file's name to the search term including case. This required adding another criterion to the search algorithm. The programmer considered just adding another condition to the current search algorithm. However, the implementation would have been confusing, it would have had to switch between case sensitive and insensitive and then check the date search feature requirements. The resulting code would have been long and procedural, which is not good object-oriented code and would have made the code smells unacceptable. At this point the search criteria had become a concept extension significant enough to warrant its own class, so he extracted the portion of search algorithm that checks files against the search criteria to a new class, `SearchManager`, see Table 6.1.

Table 6.1 Location of Search Algorithm Extension



`SearchManager` required features that added a search criterion to implement an interface called `SearchOption`. Then at runtime as the user inputs the search criteria, the `SearchOption` implementation for that criterion is added to a list in `SearchManager`, when the search is run, each file is checked against the criteria in

SearchManager's list. This change to the search algorithm meant that future changes can add new criteria, but the change will be unlikely to propagate to SearchManager, which is what happened. Changes 6 and 7 added new search criteria, but SearchManager was not impacted.

The search algorithm shows how a concept extension can evolve from a simple trivial extension to a complex extension spanning multiple classes during SC. It started as a for loop with an if condition that didn't warrant its own class and grew to the point that it required multiple classes. This is characteristic of SC, only the requirements necessary for a feature are implemented during a change; looking ahead to future changes and implementing a search algorithm to meet their needs is improper. However, SC can still be used to implement complex features and relationships in the code.

6.1.1 Exit Criteria

Exit criteria of the concept location are well-defined: The concept location ends when the appropriate concept location has been found.

6.2 Impact Analysis

During the iteration of SIP, the programmer was not always able to accurately predict the estimated impact set. Table 6.2 shows the estimated impact set in code files for each change request versus the code files in the changed set. In 4 of the 7 change requests, the 2 are not equal. This section looks at reasons why.

Table 6.2 Comparison of Estimated Impact Set and Changed Set

#	Change Request	Production Code Files		Harness Code Files		Percent (%)	
		EIS	Changed Set	EIS	Changed Set	Precision	Recall
1	Basic Search	3	4	0	0	100.0	75.0
2	Recursive Search	1	2	2	2	100.0	75.0
3	Advanced Output	14	8	7	3	52.4	100.0
4	Date Search	6	6	7	6	92.3	100.0
5	Case Sensitive	6	6	10	9	93.8	100.0
6	Extension Search	6	3	5	3	54.5	100.0
7	Properties Search	3	4	4	3	85.7	85.7
8	File Chooser Bug	1	1	2	2	100.0	100.0
9	Date Search Bug	3	3	4	3	85.7	100.0

Legend

true positive = estimated impact set ∩ changed set

true negative = estimated impact set ∪ changed set

false positive = estimated impact set - changed set

false negative = changed set - estimated impact set

$$\text{precision} = \frac{\text{true positives}}{\text{true positives + false positives}}$$

$$\text{recall} = \frac{\text{true positives}}{\text{true positives + false negatives}}$$

6.2.1 Overestimate in Change 3

Change 3 included a super class extraction [23] from the class `FileTable`, a large class with many clients and 6 suppliers. The programmer added all 6 suppliers to the estimated impact set; however two of the suppliers were not impacted by the prefactoring or by actualization. The programmer also included 4 classes in the estimated impact set that were not impacted until postfactoring.

6.2.2 Overestimate in Change 6

The programmer added 3 classes to the estimated impact set that were not impacted until postfactoring. The classes, `SearchTermOption`, `CaseSensitiveOption` and `AbstractTermOption` handle the responsibility for the search term; the programmer predicted these classes would be impacted during the change. However, the details of the implementation were more complicated than he thought. He attempted to keep actualization as simple as possible by incorporating 2 new classes that created code smells. Later, during postfactoring he combined the responsibility impacting the 3 classes and removing the code smells.

6.2.3 Missed Impact in Change 7

An example of missed impact is in change 7 where programmer missed the impact on 1 production class and several harness classes. The programmer reported that clients and suppliers to the abstract class `AbstractFile` wouldn't be impacted by the change; `AbstractFile` interacted with 308 classes as identified by JRipples and the programmer failed to inspect all of them.

The programmer had visited and used `AbstractFile` in other change requests and became confident that he understood the class and its neighbors. However, during change 7 it became apparent that the code did not work as the programmer believed.

The programmer was unfamiliar with the proxy design pattern [42]. The class `ProxyFile` is a subclass of `AbstractFile` within that pattern and overrides all the abstract methods of AbstractFile so that subclasses of ProxyFile can override only those methods that are necessary to meet their specific responsibilities. A programmer with knowledge of this design pattern would have visited ProxyFile and added it to the estimated impact set.

6.2.4 Programmer Missteps

When a programmer does not include a class in the estimated impact set, it is easy to assume a programmer misstep is the cause. One can appreciate that in complex software even the most careful programmer can miss an impacted class. The missed impact of change 7 (section 6.2.3) is an example that demonstrates three types of programmer missteps.

The programmer was under a *deadline* and students must finish projects for grades, so that they may graduate in time. The programmer could have visited all 308 neighbors of AbstractFile and identified the impact to ProxyFile; however, visiting and analyzing all of the neighbors of AbstractFile would have been time consuming. The programmer made the decision not to spend the time and move on. This is acceptable under SIP, the programmer chooses when to stop one phase and move on to the next. This is an area the programmer would like to see defined better (see section 6.7.3).

Additionally, the programmer's reasons for not visiting all the neighbors of AbstractFile also showed *habitualization*. He had visited and used AbstractFile in other change requests and became confident that he understood the class. However, during the change request it became apparent that the code did not work as the programmer believed, leading to the addition of ProxyFile to the changed set. From the experience obtained during the iteration the programmer believes habitualization should be considered in future improvements to SIP.

Finally, the programmer was also *unfamiliar with the design pattern* proxy [42], which `ProxyFile` implements. If the programmer had been more familiar with this design pattern, he could have identified `ProxyFile` as a likely impacted class and visited it.

The missed impact of change 7 is an example that includes all three types of programmer missteps. If the programmer had not made all three of the missteps, he could have identified `ProxyFile` and added it to the estimated impact set. This suggests that a careful programmer with knowledge of the program and its technologies is unlikely to leave classes out of the estimated impact set.

6.2.5 Harness Code Impact

The impact of a change on harness code was greater than the impact on production code and was more difficult to predict. An example of this is in change 7. The programmer performed a class extraction [23] that impacted 9 classes because a field was extracted to the new class. Of the 9 classes, 1 was production code and 8 were harness code. The production code class was limited to 1 class because the programmer implemented a strategy design pattern [42] during change 5.

Refactoring specific to harness code was looked at in [45]. The paper describes how to identify bad smells that are common to harness code. The programmer didn't have this knowledge during the iteration and did not follow many of their suggestions. While these refactoring techniques would have resulted in better code, the programmer does not believe they would resolve difficulty identifying impact to harness code because he found harness classes often have many more class interactions than the classes they test. A possible area of future work is to identify design patterns specifically for harness code.

6.2.6 Exit Criteria

A perfect exit criterion would be to visit all the neighbors of the impacted and propagating classes. However in the case of large neighbor sets, this is burdensome and time consuming. An analogy is in testing which often allows less than 100 percent coverage. Whenever the programmer concluded that more than 60 percent of the impacted classes were inspected, he exited the impact analysis phase with the conviction that the scope of prefactoring and actualization is sufficiently understood and the quality of the SC will not be negatively impacted.

6.3 Actualization experience and overhead

The programmer did all of the types of actualization described previously (section 3.2.5). Change 8 simply changed a single production class by adding new methods and deleting existing methods. Other changes, such as change 2 included the more complex incorporations, like incorporation through replacement. In the programmer's experience the key to making actualization easier is prefactoring. Change 5 actualization (section 5.5) simply required modifying 1 production class and incorporating 1 production class. This was because the programmer did an extensive prefactoring. This contrasts with change 3 (section 5.3) where actualization was much more difficult for the programmer. He did perform prefactoring, but limited it to 2 classes; the code was not ready for the change. He then had to implement a workaround during actualization and correct the code smells during postfactoring at a higher cost.

From a business point of view, actualization is the most important part of the change because it is the only phase that adds to the value the user can see. For this reason it is used as the business value of software. The other phases are only important to the solo programmer and are considered overhead. If we consider the time spent performing actualization plus actualization testing to be the cost of the increase in business value then adding new business value took 49 hours and 43 minutes, see Table 6.3, while the complete work on the iteration took 144 hours and 24 minutes, then the overhead rate is approximately 66%.

Table 6.3 SIP Iteration timing (Hours:Minutes)

	Change									
Phase/Action	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**Total**
Concept Location	0:22	0:00	0:33	0:00	0:00	0:00	0:00	0:00	0:00	**0:55**
Impact Analysis	2:08	2:28	3:23	1:26	1:02	0:55	0:38	0:00	0:00	**12:00**
Prefactoring	0:00	1:22	2:11	1:41	9:32	3:06	0:00	0:07	0:00	**17:59**
Prefactoring Testing	0:00	2:43	0:07	0:41	2:53	0:55	0:00	0:09	0:00	**7:28**
Actualization	5:34	3:41	4:08	4:42	1:36	2:20	2:57	0:16	0:23	**25:37**
Actualization Testing	5:02	1:52	6:42	3:34	0:49	2:36	2:32	0:37	0:22	**24:06**
Postfactoring	0:23	2:57	15:49	4:46	2:35	3:18	3:54	0:00	0:00	**33:42**
Postfactoring Testing	0:12	7:34	5:34	1:28	1:19	2:08	4:22	0:00	0:00	**22:37**
Total	**13:41**	**22:37**	**38:27**	**18:18**	**19:46**	**15:18**	**14:23**	**1:09**	**0:45**	**144:24**

6.3.1 Exit Criteria

The programmer's exit criterion for actualization was based on a quality of implementation of the change request. The programmer determined that when all tests (unit, functional and regression) passed the requirements had been met. The programmer made sure that each part of the change request is tested, including both valid and invalid inputs, and that the statement coverage of new or modified code is close to 60 percent or more.

6.4 Refactoring Experience

Pre- and postfactoring have different purposes, but at their core they are both just opportunities to refactor. In the programmer's experience they are a good time to apply design patterns to the code. At times he found it difficult to both implement the change and apply a design pattern during actualization. Accounting for change propagation and incorporating the new functionality was difficult enough.

The programmer applied a composite pattern [42] numerous times during both refactoring phases. In prefactoring of change 2 (section 5.2) he extracted `InputPanel` from `SearchDialog` to apply it. He then applied the pattern again during postfactoring by extracting `DirectoryPanel` from `InputPanel`. From this experience the programmer found both phases to be well adapted to applying patterns because the design pattern implementation could be separated from the other programming activities.

6.4.1 Prefactoring

During change 1 the programmer skipped prefactoring. In hindsight, he could have extracted classes for the input and output panels that would have made the change actualization easier. That was later remedied by prefactoring during change 2, but at a higher cost because larger amount of code had to be moved.

This contrasts with the prefactoring phase of change 8. The programmer could have skipped prefactoring here too, but a simple extract a method [23] prefactoring made replacing one interface with another much easier. Overall, the programmer found that aggressive prefactoring often makes the following actualization much easier.

6.4.2 Prefactoring Exit Criteria

The prefactoring was completed when the local structure of the code was suitable for actualization. In particular, all large significant concepts involved in actualization had a class of their own and for that, some classes were extracted from other classes if necessary. If the planned actualization used polymorphism, the base class was introduced by refactoring. If the planned actualization used a pattern (composite) [42], the pattern was fully prefactored before actualization started.

6.4.3 Postfactoring

Impact analysis does not attempt to predict postfactoring; postfactoring involves judging the new situation that arises after actualization, and sometimes may be skipped entirely. At times it involves general clean-up that may include consequences of several changes.

For example, the class `InputPanel` was added in change 2 and it added responsibility during change 3, 4 and 5, making it large and difficult to understand. In postfactoring after change 5, the programmer solved this accumulated problem by extracting the class `BasicOptionsPanels` from `InputPanel`. It contained the GUI components responsible for the search term, case sensitive and recursive search inputs. `InputPanel` was left with the responsibility to assemble all of the panels responsible for search input. After the class extraction both of the classes were responsible for a single significant concept extension, making future changes easier.

6.4.4 Postfactoring Exit Criteria

Beck and Fowler used vaguely defined "bad smells" as the entry criterion for refactoring and quoted Grandma Beck, "If it stinks, change it." [22](p. 75). The programmer reversed this vague adage into: "When it no longer stinks, stop." More specifically, the programmer used the following criterion: When each new code construct has an identifier that explains its responsibility, all new or modified methods deal with a single responsibility, and all new or modified classes implement a single significant concept, then the postfactoring is done. The programmer used the LOC metric as a guideline to identify artifacts likely to break these criteria; methods longer than 10 LOC and classes longer than 100 LOC were scrutinized. However the postfactoring was limited to the new or modified code and the programmer did not attempt to refactor the rest of the muCommander.

6.5 Verification

There were 11 bugs introduced during the SIP iteration. Of these, 9 were fixed immediately in the same change. No regression bugs were found in the intact code in any of the changes, all bugs were introduced in the changing code. The programmer added to the test harness a new unit test class for each new production code class, and a functional tests for each new feature, such as date and extension searches.

6.6 SIP Exceptions

While SIP worked quite well for the programmer during the iteration, there were some exceptions that didn't neatly fit into the process. These exceptions to the process while relativity minor suggest that SIP can be improved upon.

6.6.1 Changing Behavior during Refactoring

The programmer performed refactorings that changed the behavior of the program. In change 3 postfactoring stage the programmer extracted the responsibility of stopping the thread that is created to iterate through the file system from `SearchDialog` to `SearchThread`. When the programmer did this he reworked the code in a way that also improved the response time of stopping the search. After actualization, there was a short delay, of about a second after pressing the “Stop” button. When the programmer extracted the responsibility to stop the thread he also added a method to `ResultsPanel` called `notifyEnd()` that `SearchThread` calls when a search is stopped. This changed the behavior of the program. The programmer justified this exception because of its small size (it added 1 LOC to `SearchThread` and a 4 LOC method to `ResultsPanel)` and because the behavior change to the program was small. However, it was an exception to SIP.

During the iteration there were several times when the programmer was not sure if the modification he was doing is allowed during that phase or not. Additionally, even if the programmer correctly separated refactoring and actualization, the programmer found the strict separation of the two phases to be burdensome at times. He makes suggestions to this issue in SIP criticism (section 6.9).

6.6.2 Additional Commits

The last exception is that the programmer committed the code to the repository not just at the end of the change, but also after prefactoring and actualization. The process only allows for the code to be committed at the conclusion of the change. This may have forced the programmer to be more diligent separating refactoring from actualization. If there was no record of the code in between phases programmers may mix these phases changing the outcome of the process.

6.7 Proposed SIP Evolution

SIP served the programmer well during the iteration. The following section describes possible improvements and times the programmer broke from the process.

6.7.1 Phase continuity has priority over concepts

The programmer found it artificial to separate the refactoring and actualization stages. Changes often dealt with multiple concepts, such as GUI and data structure. In these cases the he was tempted to do the three phases on each concept individually instead of performing all the prefactoring, then all the actualization and finally all the postfactoring. In the programmer's experience it is easier to manage one concept at a time.

An example is in change 2, the programmer extracted `InputPanel` to handle the user input and `SearchThread` for the search algorithm during prefactoring. He then added GUI components to `InputPanel` and replaced `SearchThread` with a more capable class during actualization. Finally, during postfactoring he extracted 2 classes from `InputPanel` and extracted misplaced responsibility to `SearchThread`. The programmer felt it would have been easier to with each concept individually because that is a more natural way for him to perform tasks.

A solution to this would be to have a cycle inside SC from the end of postfactoring to the beginning of prefactoring. Since phases can be skipped a programmer could do the necessary phases for each concept. A disadvantage is that the program could be in a broken state at the end of a phase. Under the current process the program is stable at the end of each phase.

6.7.2 Local and renaming refactoring during actualization

During the iteration the programmer was often temped to do local refactoring during actualization, which is not allowed under SC. An example of local refactoring is extracting a method. At times immediately after adding a method to a class, the programmer would realize that the method had multiple responsibilities and should be divided into 2 methods. However, under SC the programmer had to wait until postfactoring to address this. This means that the programmer would either have to remember or make a note to do the refactoring later. By putting it off until later the programmer could forget to do it resulting in code decay or may have to study the code again to accomplish it resulting in wasted time. The programmer found this to contrast with the importance of refactoring. These types of refactoring should be allowed during actualization.

The programmer found that sometimes the first name given to an identifier was not the best name. Under SC he is required to wait until postfactoring to rename the identifier. This makes renaming an identifier more difficult, which discourages it effectively encouraging the programmer to allow code decay. In the past renaming was problematic taking the programmer away from the subtask at hand, however, with the current state of the art refactoring tools and unit testing tools available this is an antiquated strategy. A programmer can now rename an identifier and be confident he will not introduce bugs. Therefore, this type of refactoring should be allowed during actualization.

6.7.3 Exit Criteria

During the iteration the programmer developed exit criteria based on his best judgment because SC does not have a defined set of exit criteria. After the programmer's experience from the iteration, he believes that a formally defined set of exit criteria for all phases to be a next step for SIP because it would help assure solo programmers that they are correctly enacting the process.

6.7.4 Enactment Rules

The SIP process requires enactment rules. These rules are set by the programmer and may vary from one project to the next. An example of one such rule is that the 60 percent of the program's new statements will have unit test coverage. The areas where these rules are need should be identified and possible rules should be written. One way to do this would be to have different levels such as low, medium and high levels for each rule that a programmer can choose from.

6.8 SIP versus Ad hoc

Chapter 2 presented previous research on software research. It demonstrated the idea that a well-defined process is required to produce quality software and it is clearly well accepted in the field of software engineering. However, this idea is mostly focused on teams producing software. A reasonable programmer may still ask the question, “Why should a solo programmer use a defined process over ad hoc methods?”

Humphrey wondered why it is so difficult to get programmers to adopt PSP in spite of the evidence that they produced higher quality software faster [46]. The paper continues by presenting methods that instructors can use to encourage the use of PSP. This raises the question, “If there is so much evidence that PSP works and programmers still do not want to use it, why force programmers?” This question is answered by Humphrey in his personal experience using PSP, “The results were truly amazing. I was more productive, the quality of my work improved sharply, and I could make accurate personal plans.” (p. 3) Supplementary evidence of PSP’s effectiveness is presented [11]. This case study showed programmer’s LOC per hour increased and defect rates decreased when using PSP.

The underlying reasons that programmers should adopt PSP are the same reasons programmers should use SIP; it will help them produce higher quality software faster. By recording the time the individual phases of SC take, the programmer will be able to predict how long similar phases will take in the future. Additionally, if a particular phase consumes a large amount of a programmer’s time, he will be able to address it. The programmer can change techniques, such as using a dependency search instead of a grep search for concept location, through external training or by introducing software tools to assist with the phase. An ad hoc programmer does not have this information, so he cannot use previous phase times to make future estimates and cannot target specific phases for improvement. Actually, the ad hoc programmer does not even have defined phases, which would make reasonable guesses even more difficult. Finally, a SIP programmer will know if his abilities are improving or deteriorating over time allowing him to adjust for the volatility inherent in software engineering.

The SIP programmer in this experience report experienced similar results. One specific example is the phase of prefactoring; as the programmer became more experienced in SIP he was able to take better advantage of it. During change request 1 the programmer skipped prefactoring altogether. In hindsight he could have still used `RunDialog` as a template, but also deleted unneeded code and extracted classes for the input and output panels. This would have made the phase actualization easier. This contrasts with the prefactoring phase of change request 8; which was a much smaller change. The programmer could have skipped prefactoring here too, but a simple extract method made replacing one interface with another much easier because the code was ready for the change. Overall, the more the programmer became experienced with SIP, the faster the change requests could be completed with fewer defects; even if he is not required to, he will use SIP in his future programming projects.

6.9 Amount of Rework

A proponent of up-front software design can argue that SC requires significant rework by producing temporary code that later gets discarded. The programmer estimated the amount of rework in the SIP iteration using LOC granularity. The three possibilities for each LOC changed during a change request are:

1. added - new to the program and therefore cannot be rework
2. moved - was in the wrong place, it is not rewritten, not rework
3. deleted or replaced

A LOC was deleted because it was replaced with better functionality or it was never needed to begin with; we used this deleted code as an indicator of the amount of rework. LOCs are organized by phase and rework is calculated as deleted LOC divided by added LOC, see right column of Table 6.4.

While there was a significant rework during some individual phases, the average amount of rework over the iteration was 27 percent. Boehm and Basili found that rework accounted for 40 to 50 percent of a project [47]. While this one iteration of SIP is not enough to draw the conclusion that SIP requires less rework than other processes, it does indicate that SIP does not require significantly more rework than other software processes.

These figures were collected by a program the programmer wrote for this experience report. It compared diff files created by TortoiseMerge. A LOC with a '+' as the first character is an added LOC, similarly a LOC with a '-' as the first character is a deleted LOC. The program then compared each deleted LOC to the set of added LOCs; if it was in the added set, the LOC was removed from both the added and deleted sets and it was added to the moved set. Additionally, this threat was not presented to the programmer until after the programmer finished change request 7.

Table 6.4 Rework by Phase

Change Request	Phase	Deleted ÷Added
1	Prefactoring	0.0%
	Actualization	0.0%
	Postfactoring	533.3%
2	Prefactoring	59.7%
	Actualization	11.4%
	Postfactoring	71.2%
3	Prefactoring	38.6%
	Actualization	8.6%

	Postfactoring	49.6%
4	Prefactoring	22.9%
	Actualization	0.3%
	Postfactoring	137.6%
5	Prefactoring	32.5%
	Actualization	2.6%
	Postfactoring	73.3%
6	Prefactoring	19.5%
	Actualization	6.6%
	Postfactoring	100.0%
7	Prefactoring	0.0%
	Actualization	0.6%
	Postfactoring	78.0%
8	Prefactoring	0.0%
	Actualization	25.0%
	Postfactoring	0.0%
9	Prefactoring	0.0%
	Actualization	14.0%
	Postfactoring	0.0%
Total		27.0%

6.10 Technologies

The programmer did not find collecting the data for the iteration to be overly burdensome. He believes that the software engineering tools used during the iteration made collecting the data easier; especially in the case of timing the phases. This section describes the programmer's experience with the software engineering tools used during the iteration.

6.10.1 JRipples

The JRipples tool was especially useful during impact analysis. Certain classes can have hundreds of neighbors and identifying all of them can be a tedious and time consuming task. In this study, the programmer used it to identify classes that interact with a class; without this tool, the action would be much more tedious and error prone.

While the programmer found JRipples to be very useful, he did find features that would be valuable to add. Some of the features are trivial, while others may be difficult. The most thought-provoking feature is to add the ability to tell the programmer when to stop impact analysis. While much research has been done on impact analysis (section 2.2.1) there is not a well-defined set of exit criteria, so adding this to JRipples is not straight forward.

During impact analysis the programmer ran into this problem, he didn't know when to stop impact analysis. This is especially true when a class had a large number of neighbors and visiting them all was unpractical. For instance, during change 7 marking `AbstractFile` Impacted added 307 to the Next set of classes. This is too many to effectively inspect. Even if he spent the time to visit all these classes, he believed that the visits would have become so repetitive that he would have likely missed potential impact. An analogy showing why a large set of neighbors is unreasonable is from concept location; if a programmer performed a grep search and was presented with hundreds of results he would probably revise his query. However, a programmer doesn't make queries during impact analysis; he visits the neighbors of impacted and propagating classes.

JRipples has heuristic tools to identify the neighbors that are most likely to be impacted. The analysis tools assign high values to the classes most likely to be impacted and low values to those less likely to be impacted. It has different algorithms to assign these values and it could be useful to a programmer. The programmer didn't use these tools, which could have helped. However, these tools still wouldn't answer the fundamental question, "When do I stop impact analysis?" The tools give all neighbors a value, if the programmer chooses a value and only inspected all classes with higher values; it would be arbitrary and fundamentally not any better than letting the programmer choose when to end impact analysis. More research needs to be done on identifying a stop point for impact analysis.

This presents an aspect of muCommander for evaluation; there are classes that have a large percentage of the classes of the program as neighbors. `AbstractFile` has over 300 neighbors, which is more than 25 percent of the program others such as `ActionManager` have more than 10 percent of the classes as neighbors. The classes are reused instead of being duplicated, which is good, but impact analysis becomes difficult. It is easy to argue that a file system explorer that mainly displays and manipulates files will have class that is extensively used throughout the program. However, in the case of `ActionManager`, it is less clear if it is necessary for it to interact with so many other classes. `ActionManager` implements a factory design pattern that in part limits the impact of changes; however, it has a deficiency that makes impact analysis difficult. Its implementation requires that classes to add code to `ActionManager` to register their action. If `ActionManager` had the ability to find the action classes, it would have fewer neighbors, making impact analysis seem easier, but this would also create hidden dependencies making impact analysis difficult in a different way. Further research into how design patterns affect impact analysis is needed.

Another change request for JRipples is to improve the filter. JRipples has a filter to show children and parents of a class. However, it was unclear to the user exactly how the filter defined the parent and child of a class. The programmer would rather have an option in the right click context menu that shows only the classes that interact with a selected class. Currently if a programmer marks a class as Propagating in JRipples, the classes that interact with that class will be marked Next and added to the set of Next classes. When the programmer marked a class as Propagating he wanted to visit only the classes that interact with the propagating class, however he found it difficult to identify which classes interact with the propagating class with JRipples.

JRipples also has a serious bug that needs to be addressed. The Hierarchical view, which displays classes, their fields and methods, is extremely slow to sort. It is so slow that is it unusable on a project the size of muCommander. It can be used with small projects and faster computers could probably handle larger programs than slower machines. The table view, which only displays classes, does not appear to suffer from this deficiency. However, the hierarchical view is default view, so this bug is one of the first impressions JRipples gives to the users.

The last change is to save the state of JRipples when Eclipse closes. Currently, the programmer must remember to save the current JRipples state before exiting Eclipse. On the next startup the programmer must then reload the correct state from a JRipples menu. This contradicts many other plugins that automatically save their states when Eclipse exits. On several occasions the programmer forgot do this and lost the information gathered in his programming session. Additionally, it should regularly save the state in the background in case of a program crash. This change request may be of little research interest, but is very important from a usability standpoint.

6.10.2 Clover Java Code Coverage & Test Optimization

The programmer used Clover to collect the statement level test coverage for the project. It performed well; Clover included total statements and percent of statements covered from the entire program to method granularity. It also highlights the statements executed in green and those not executed in red. Clover also allows the user to create custom metrics based on the standard metrics. The programmer created a metric containing the number of statements covered that helped him with reports.

The one problem the programmer had with Clover is that if it is used with the Eclipse debugger, it adds an extra call to a method in one of its classes for every statement. This made debugging very slow and difficult. The issue is compounded because once Clover is enabled on a project, the project must be run with it. This appears to be a bug because it adds an option to run projects with it. This implies that the Eclipse basic run should be without Clover, but it includes Clover.

6.10.3 Mylyn & Tasktop

Mylyn and Tasktop worked very well. The programmer found the interface to log timing data for different phases to be very easy. It has a feature that pauses the timer if the Eclipse window is not the active one. The programmer found this very useful, he could respond to an email without having to manually pause the timer without corrupting it.

6.10.4 Abbot Java GUI Test Framework

Abbot was easy for the programmer to use after the first 2 changes. The functional tests are written very similar to JUnit tests. The built in robot test classes are easy to work with; there are specific classes for the Swing library classes. Overall Abbot worked well for the programmer, but he did run into a few issues, which lead to change requests.

The first issue was that the tests run much slower than unit tests, instead of a fraction of a second, many took over a second. This is not just an issue of setup overhead because some of the unit tests also required a similar amount of setup. It is in part because Abbot does not support a onetime setup method for an entire test suite; if numerous objects must be created, they must be created for each test in the suite. These issues lead to 2 change requests, one to do an optimization of Abbot and the second to add the capability for a onetime setup method like in JUnit.

A related issue was that the tests were inconsistent, which seemed to be caused by the excessive use of resources. When tests classes were run individually, they would pass without problem. When all the tests in the project were run, at times they would pass and others they wouldn't. The error given was usually that Abbot couldn't find the GUI component. Rerunning the tests was one workaround. Another was to add a delay to the test, but this would slow the test even more and may not work if the tests are run on other computers. This should be addressed with the optimization change.

The last issue was that Abbot was not able to find some modified Swing components. An example of this is the `ComponentTitledBorder` class it adds Swing components to a border. This class did not have a specific Abbot tester and the existing

Abbot tester could not find the component in the border. The programmer created a workaround, based on the components coordinates, but they could fail on other computers. The programmer would like more documentation on how to write general custom testers.

6.10.5 Subversion & TortoiseSVN

Subversion and TortoiseSVN meet all the version control system needs of the programmer.

6.10.6 DiffStats

The programmer created DiffStats because he was unable to find a diff tool that could provide the metrics he required. He found a variety of diff tools that could visually show the user added, deleted and changed LOC in a single file. However, these tools didn't provide LOC totals for the categories. This tool analysis is very simple and should be expanded and refined for future use.

6.11 Threats to Validity

This experience report contains data from one iteration of SIP, done by a specific programmer in a specific program. Further research is recommended before concluding that the results apply in general. Transferring this experience to other contexts should be done with caution.

In particular the programmer that performed the iteration may be a subject that is particularly susceptible to adopting SIP. He had written a variety of programs in a university setting, which made him familiar with many aspects of programming such as object-oriented technology, design patterns and data structures. However, when introduced to SIP he did not have the skills to perform changes to large unfamiliar programs. If the programmer had been less knowledgeable, he may not have been able to successfully perform a SIP iteration at all. Likewise, if the programmer already was able to make changes on large programs he was unfamiliar with, he may have found SIP inadequate.

The program selected may also have contributed to the success of the experience report. The program used was in a state that was ready for SC. Programs can suffer from code decay to the point where it is impossible to perform SC on them [1]. If a program was used that was closer to the point where SC was impossible, the programmer may not have been successful.

Another threat is that SIP does not require, nor exclude any particular software tools. This experience report used a variety of tools. One or all of these may be required for a successful SIP iteration. In particular, the programmer is unsure how he could have performed impact analysis without JRipples. Identifying neighbors of classes would have been difficult and the iteration may have failed. The other tools may have been just as integral to the SIP iteration.

Finally, the SIP iteration was done in a university setting with a professor and a peer standing in for users. These users have different motivations than users of commercial, open source and other users of software. These other types of users are almost certainly more common than a professor and a peer. While SIP meet the needs of these users, it is possible that it would not meet the needs of other users.

Chapter 7
Future Work and Conclusions
7.1 Future Work

This chapter presents issues and questions raised during the iteration that require more study and then presents the conclusions of the experience report.

7.1.1 Level of adoption Study

The SC process at the core of SIP has been taught by Dr. Rajlich at Wayne State University for several years. An interesting follow up study would be to see if students continue to use the SC process in their future classes or professional careers. Johnson, et al. looked into the adoption of PSP (section 2.1.3) they found no studies into adoption rates, but reported that,

> *…anecdotal evidence does not support the second conjecture [that a student will use PSP when not required in a classroom setting]. For example, a report on a workshop of PSP instructors reveals that in one course of 78 students, 72 of them "abandoned" the PSP because they felt "it would impose an excessively strict process on them and that the extra work would not pay off."* [48](p. 2)

This would indicate that a study into the adoption rates of both SC and PSP could provide valuable insight. The SC process is a less invasive process for programmers to implement. However, PSP provides tailored metrics to each programmer showing its value. Measuring the adoption rate would be a real validation of each processes' value, beyond the classroom.

An adoption rate comparison would also provide valuable information to the developers of future software processes. If SIP and the SC process is adopter by programmers at a significantly greater rate, future processes should take this into account. Conversely, if PSP is adopted at a higher rate by programmers after they are no longer required to use it, the metrics convincing the programmer of its value outweigh the cost of the process. If both processes are adopted at a low rate, then new ideas could be considered.

7.1.2 Team Processes Research

In addition to SIP Rajlich also defined team processes [1]. These processes include the Agile Iterative Process (AIP) for small teams of programmers and the Directed Iterative Process (DIP) and Centralized Iterative Process (CIP) for large teams. Performing an experience report or case study to confirm these processes would be one next logical step. AIP appears to be a reasonable candidate for a group of students in a university setting such as a classroom or for a research project. DIP is more suited to a case study in an industrial setting; a suitable candidate may be difficult to identify though. A case study of CIP could be performed on an open source project. A team of students could be the managers and code owners with the open source project's community serving as the programmers and testers. A possible open source project is JRipples. An advantage to this is that it would also improve JRipples making the phases of the SC process easier. However, JRipples may not have a large enough community for the case study. Another problem for this case study would be assuring that the open source community used the SC process to implement the change requests. The code owners could require the timing data and other metrics with each commit, but it would still be difficult to know for certain.

7.2 Conclusion

This thesis shows that SIP can be followed literally and used by a single programmer to add functionality to large open source software. A single programmer who had university experience in programming, limited experience in Java programming and was unfamiliar with the muCommander project was able to add functionality to it using SIP.

The core of SIP is the task of SC. It was used in this experience report as an instructional framework to add functionality to a large open source program. The new functionality is shown to have a low number of defects through testing. Additionally, if the functionality added in this experience report does not meet the requirements of the stakeholders for any number of reasons, SIP has a mechanism in place to meet the requirements; new change requests can be added to the product backlog at any time. Further iterations of SIP could add to the functionality of this experience report, change it or remove it completely as the stakeholders require. New change requests also provide a method to fix any defects found in the future. This is important since testing cannot guarantee the absence of defects [1]. This demonstrates how a solo programmer can use SIP to meet the project's needs and goals.

APPENDIX A.

SIP – Change 1 Basic Search

This appendix contains the change reports summarize in chapter 5. The programmer of this experience report filed after each change request.

A.1.1 Initiation

Add a basic search function that allows a user to search in the current directory for all or part of the title of a folder or file, and return a list of the matching files and directories It is an application which enhances an operating system's file explorer. However, it does not have any search capabilities, which would help a user find files, folders or contents of files.

This change request will add a basic search function. The search will allow a user to search in the current folder for all or part of the title of a folder or file. It will return a list of the matching files and folders.

The search functionality can be activated in three different ways. First the user can use the programs menu to select Go → Search..., second the user can select a binocular icon on the quick launch toolbar, finally, the user can use a hot or virtual key combination of Ctrl + F. All three options open a new window where the user can type search terms and start a search. The window will also display the list of results, if any.

A.1.2 Concept Location

The concept location to find is the muCommander "Go" menu where the option will be added to initiate a search. The programmer started a dependency search by marking the Launcher class, which contains the program's main method as propagating.

JRipples added 43 neighbors of Launcher to the set of Next code files. Since the programmer did not know anything about the program, he decided to visit them one by one. `AbstractFile`, `AbstractNotifier` and `ActionKeymapIO` were visited and marked Unchanged. The programmer then visited `ActionManager` this file contains a library of all the possible actions in the program. It is used as a central location to keep all the possible actions of the program organized. Upon inspection, the programmer realized that this was where the search functionality would be added, the “Go” menu would be part of the impact analysis. This completed concept location. Table A.1summarizes the concept location code file totals and Table A.2 lists the code files visited during concept location. Figure A.1 is a UML diagram of concept location.

Table A.1 Change 1 Concept Location Summary

Title	Code Files			Comments
	Visited	Propagating	Unchanged	
Basic Search	5	1	3	

Table A.2 Change 1 Concept Location Code Files Visited

#	Code File	Tool used	Located?	Comments
1	`Launcher`	JRipples → Propagating	Propagating	This is the main start location for the program
2	`AbstractFile`	JRipples → Unchanged	Unchanged	This class is used by muCommander to store data about files
3	`AbstractNotifier`	JRipples → Unchanged	Unchanged	This class displays user notifications
4	`ActionKeymapIO`	JRipples → Unchanged	Unchanged	This class read user defined keyboard commands or hot keys
5	`ActionManager`	JRipples → Located	Located	This class is where all the concepts of the program are registered

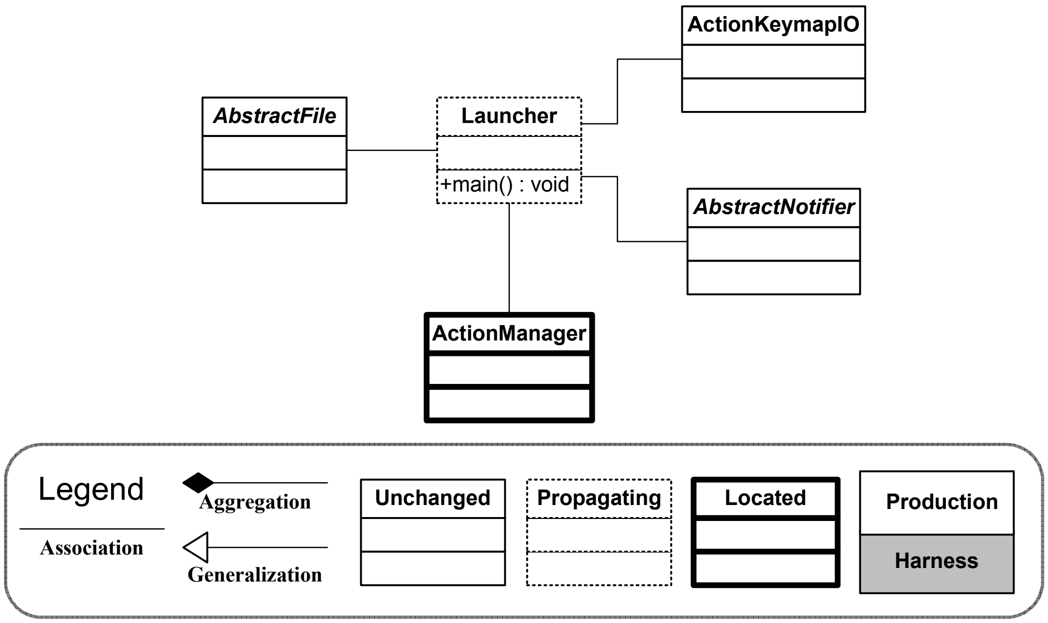


Figure A.1 Change 1 Concept Location UML

A.1.3 Impact Analysis

To start impact analysis the programmer switched JRipples from concept location phase to impact analysis phase. This changed `ActionManager`'s mark from Located to Impacted and created a new Next set of code files composed of 172 of `ActionManager`'s neighbors. Since the programmer was unfamiliar with the `ActionManager`, he visited the 6 likely clients and suppliers of `ActionManager` because their names started with Action. The programmer marked these 6 code files, `ActionDescriptor`, `ActionFactory`, `ActionKeyMap`, `ActionKeyMapReader`, `ActionParameters` and `ActionProperties` Unchanged.

The programmer gained knowledge from these visits and decided to concentrate further impact analysis on finding the menus where the options to open a search window would be added. He visited `CommandBar` and `CommandBarButton` and marked them Unchanged, they did not handle the menu responsibility. The next visit was to MainMenuBar, which is responsible for the “Go” menu where the search option would be added, it was marked as Impacted. JRipples added its neighbors to the Next set of code files for a current total of 194. The programmer continued looking for the class responsible for the toolbar, which will also get a search option. During this search he noticed the NewWindowAction code file marked Next and visited it because its name sounded like it may be relevant. It did not need to be changed and so he marked it Unchanged. He then visited RunCommandAction for the same reason but also marked in Unchanged.

The programmer then found ToolBar in the list of Next code files and visited it. It did contain the responsibility for adding buttons, but it depends on a supplier to define the buttons; it was marked as Propagating. ToolBarAttributes was visited next; it is responsible for defining the toolbar buttons, so the programmer marked it Impacted. The programmer still was not sure how to access files to search them. He visited FileTable from the Next set, it did not contain a method to access the files displayed in it. The programmer suspected its field of type FileTableModel would, so he marked it as Propagating. FileTableModel was added to the set of Next code files by JRipples, which now totaled 241. It contained the necessary methods to access the files to search so it was marked as Unchanged. At the point FileTable should be marked Unchanged because it does not propagate to an impacted class, but JRipples does not allow this.

The programmer performed one final task, because he was unfamiliar with the code conventions of muCommander, he visited the code file RunDialog and marked it Unchanged. The programmer chose RunDialog because it was part of the Next set and it had dialog in the name. He will use it during actualization; the new class that will handle the responsibility of creating a dialog for the search will be modeled after it. The programmer stopped impact analysis because the he determined the impact of the change would not propagate further; there were 240 code files in the Next set that were not visited. Table A.3 is a summary of the code files visited during impact analysis. Table A.4 shows the total of each type of code file during impact analysis. Figure A.2 is a UML diagram of impact analysis.

Table A.3 Change 1 Impact Analysis Summary

Title	Code Files					Comments
	Visited	Impacted	Propagating	Unchanged	Not Visited	
Basic Search	17	3	1	13	240	

Table A.4 Change 1 Impact Analysis Code Files Visited

#	Code File	Tool used	Impacted?	Comments
1	`ActionManager`	JRipples → Impacted	Impacted	This class registers all actions in the program
2	`ActionDescriptor`	JRipples → Unchanged	Unchanged	
3	`ActionFactory`	JRipples → Unchanged	Unchanged	
4	`ActionKeyMap`	JRipples → Unchanged	Unchanged	Thought this class might register hot keys but it does not register them in the code
5	`ActionKeyMapReader`	JRipples → Unchanged	Unchanged	Thought this class might register hot keys but it does not register them in the code
6	`ActionParameters`	JRipples → Unchanged	Unchanged	
7	`ActionProperties`	JRipples → Unchanged	Unchanged	

8	`CommandBar`	JRipples → Unchanged	Unchanged	Not the toolbar I am looking for
9	`CommandBarButton`	JRipples → Unchanged	Unchanged	Not the toolbar I am looking for
10	`MainMenuBar`	JRipples → Impacted	Impacted	This toolbar has the Go menu
11	`NewWindowAction`	JRipples → Unchanged	Unchanged	
12	`RunCommandAction`	JRipples → Unchanged	Unchanged	
13	`ToolBar`	JRipples → Propagating	Propagating	This is the quick launch toolbar
14	`ToolBarAttributes`	JRipples → Impacted	Impacted	This is the class that loads the icons for the quick launch toolbar
15	`FileTable`	JRipples → Unchanged	Unchanged	This was marked as Propagating, but the path was found not to be Impacted. The data was never undone in JRipples, it is incorrectly marked..
16	`FileTableModel`	JRipples → Unchanged	Unchanged	This class will be used for the search feature, but it does not need to be changed, its interface can be used as is
17	`RunDialog`	JRipples → Unchanged	Unchanged	This class will be the model for a new class responsible for the search

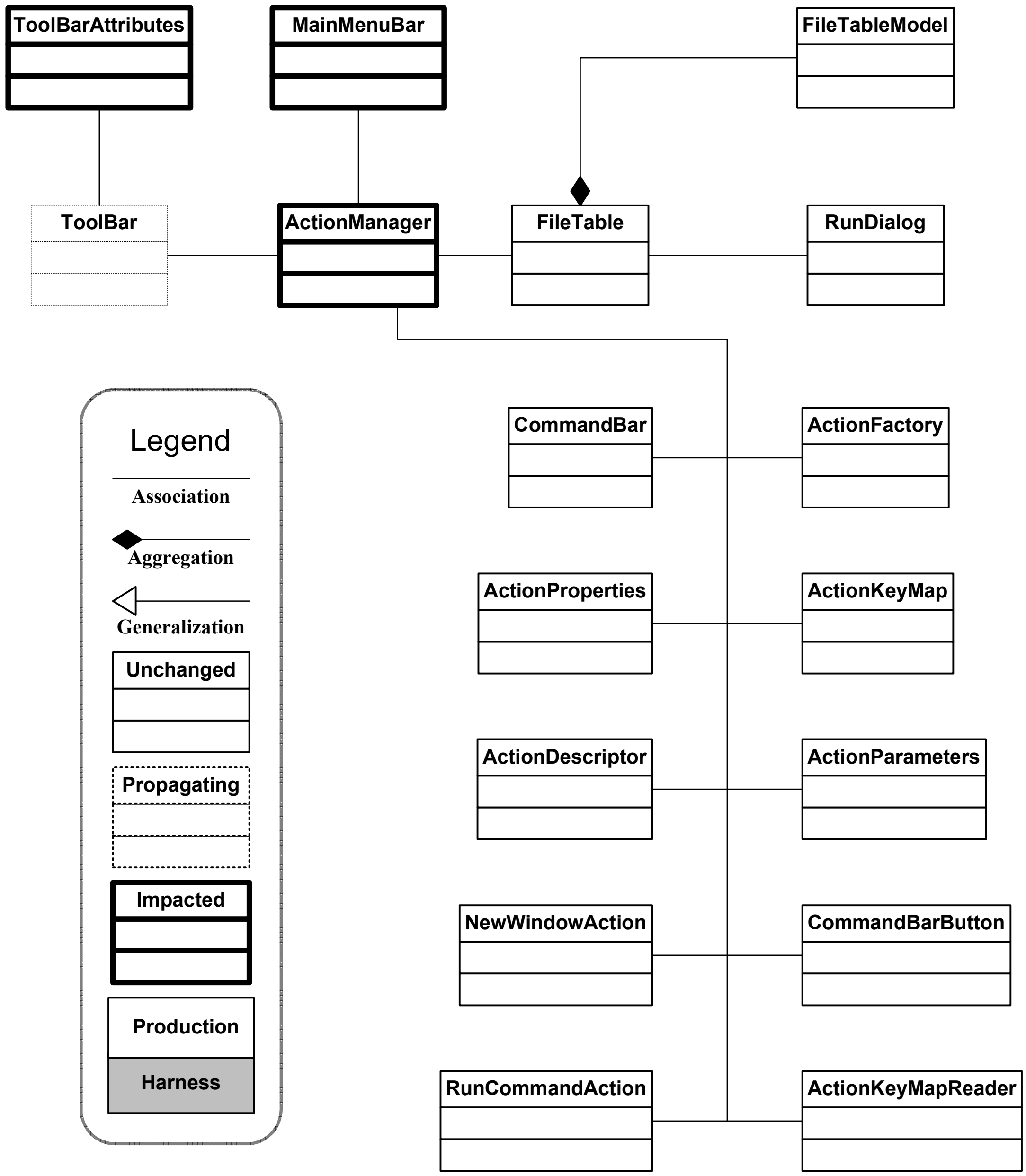


Figure A.2 Change 1 Impact Analysis UML

A.1.4 Prefactoring

There was no prefactoring done in this change request.

A.1.5 Actualization

To actualize the change request the existing `ActionManager` class required 3 new classes to register a new action. These classes are defined in 1 code file; 2 of the classes are nested inside the third. The programmer added a new supplier code file, `SearchAction` through incorporation. It contains a class also called `SearchAction` with 2 nested classes inside of it called Factory and Descriptor, which return attributes of the action as required by `ActionManager`.

The second code file, `SearchDialog`, contains a single class. It creates a new window that contains components for the search criteria to be entered and a list of results displayed. This class was based upon an existing muCommander class, `RunDialog`, which also opens a new window for user input. It was used so that the code's current naming conventions and styles could be followed. This way the change request will blend in with the existing code.

The programmer encountered a problem while adding the harness code files. The tests would throw an exception because the singleton `Translator` class was not initialized, the translator needs to be loaded by each harness code file in its `oneTimeSetUp()` method. This caused another problem, if 2 harness code files were run at the same time they both would initialize the `Translator`. To correct for this the programmer add a `boolean` field, `isLoaded`. The field is initialized to false and then set to true when the `Translator` is initialized. The programmer did not realize this would be an issue during impact analysis. The `Translator` code file was added to the changed set

Two additional code files were added for the purpose of verification; 1 class for unit testing, `BasicSearchUnitTest` and 1 for functional testing, `BasicSearchFuncTest`. These classes will be described in verification (section A.1.7). The total of each class by type of visit is listed in Table A.5. Table A.6 is a summary of the changes made to each class during actualization and the LOC added and deleted. Figure A.3 is a UML of actualization.

Since there is no search feature in the current program, there was no package that the new search feature fit into. Therefore, the programmer added a new package `org.severe.main.ui.SearchDialog` to hold the new code files.

Table A.5 Change 1 Actualization Summary

Title	Code Files					
	Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
Basic Search	8	4	4	1	3	1

Table A.6 Change 1 Actualization Code Files

#	Code File	Task	Lines of Code		
			Added	Deleted	Total
1	`SearchAction`	Added class	28	0	28
2	`SearchDialog`	Added class	209	0	209
3	`MainMenuBar`	Changed method	3	0	3
4	`ToolBarAttributes`	Changed method	2	0	2
5	`ActionManager`	Changed method	2	1	3
6	`Translator`	Added field, method	3	0	3
7	`BasicSearchUnitTest`	Added test class	92	0	92
8	`BasicSearchFuncTest`	Added test class	104	0	104

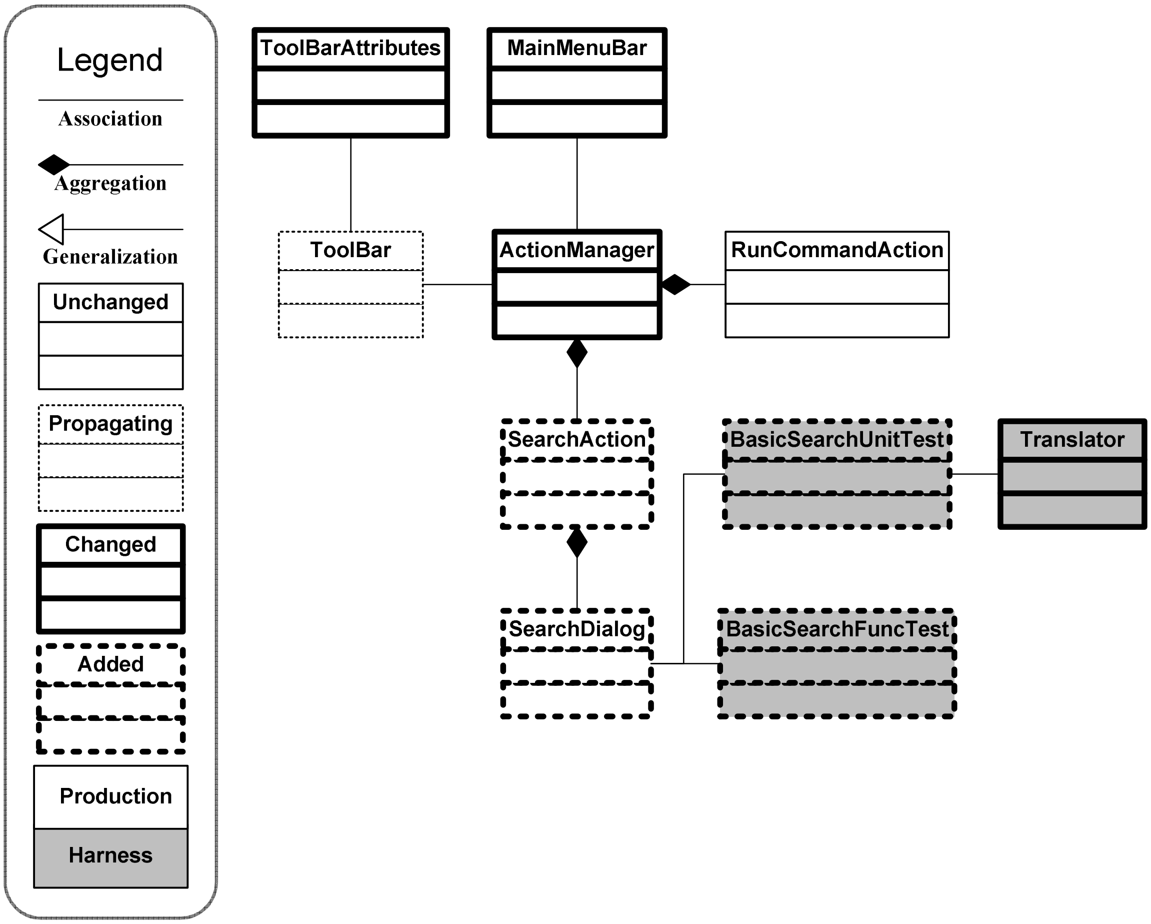


Figure A.3 Change 1 Actualization UML

A.1.5.1 SearchAction code file

`SearchAction` is a class that requires a few simple methods that return parameters so the `ActionManager` class can register what to do upon certain events. All these methods and the `Factory` class must be defined, but if a parameter does not apply to the registered class, it can just return `null`. `SearchAction` has 2 nested classes, `Factory` and `Descriptor`.

The Factory class is a static class that actually creates an instance of the `SearchAction` class and registers it with the `MainFrame` window. It only contains a constructor that calls the `SearchAction` constructor. The program uses a factory pattern of static classes inside of a class to create the class instead of calling the constructor of the class directly. It appears the development team does this to keep track of class instances so they are not created repeatedly. The `Descriptor` class is also static and contains the parameters for the class. The class also registers a hot key.

A.1.5.2 SearchDialog class

The `SearchDialog` class was modeled on the `RunDialog` class. The programmer did this because both of the classes create a new window; this allowed the new code to blend with the existing code. `RunDialog` takes a text command and creates a new process to execute the command, then reports back any error messages; `SearchDialog` gets the current folder that the user has selected in its parent window and searches it. While they do both create a window to get user information from the user, their functionality ends there, so they are very different classes.

A.1.5.3 MainMenuBar class

The programmer added a separator bar and the Search selection to the `MainMenuBar` method. Additionally the added code was limited to the Go menu section of the method.

A.1.5.4 ToolBarAttributes class

The `ToolBarAttributes` class actualization was very similar to the `MainMenuBar` actualization. They both define toolbars through which the user can select specific functionality. Because, a search feature is probably an often used feature, it was added to the quick launch toolbar defined in the `ToolBarAttributes`; this allows the user to open the search window with a single mouse click.

To modify this class only 2 LOCs need to be added to the method that adds the toolbar icons. To make this work, an image of the icon was added to the custom\images.action folder named Search.png. This was done quickly because of previous Java programming experience. The methods of software evolution do not provide strategies to do this.

A.1.5.5 ActionManager class

This class is set up so that it only requires 1 LOC to be added to register a new action. The single LOC calls the 2 static classes from the `SearchAction` class. The change is done to the `registerActions()` method; all actions are listed in alphabetical order.

A.1.5.6 Translator class

The programmer added a `boolean` field, `isLoaded` which is initialized to false by default and a getter for it. The `loadDictionaryFile()` method sets the `isLoaded` field to true, so that the method will not be called again.

A.1.5.7 BasicSearchUnitTest class

This class was added, it is the unit test suite for the search classes; it has 5 tests.

A.1.5.8 BasicSearchFuncTest class

This class is a functional test suite for searches; it has 1 test. There is an issue with this test class. It passes it assertions, but stops before it finishes. It then displays a gray result, instead of the desired green or test fail red. This harness class uses the Abbot functional test framework. The programmer is unfamiliar with the framework and is therefore unsure the cause of the problem. The programmer decided to complete the change and correct the issue at a later date.

A.1.6 Postfactoring

The postfactoring was very straight forward. Old comments were deleted and new comments added. Additionally, 2 unused methods were deleted. The total of each class by type of visit is listed in Table A.7. Table A.8 is a summary of the refactoring type and LOC added and deleted during postfactoring. Figure A.4 is a UML of postfactoring.

Table A.7 Change 1 Postfactoring Summary

Title	Code Files					
	Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
Basic Search	3	2	0	0	1	0

Table A.8 Change 1 Postfactoring Code Files

#	Code File	Task	Lines of Code		
			Added	Deleted	Total
1	`SearchDialog`	Javadoc	1	17	18
2	`BasicSearchFuncTest`	Removed unused code	3	0	3

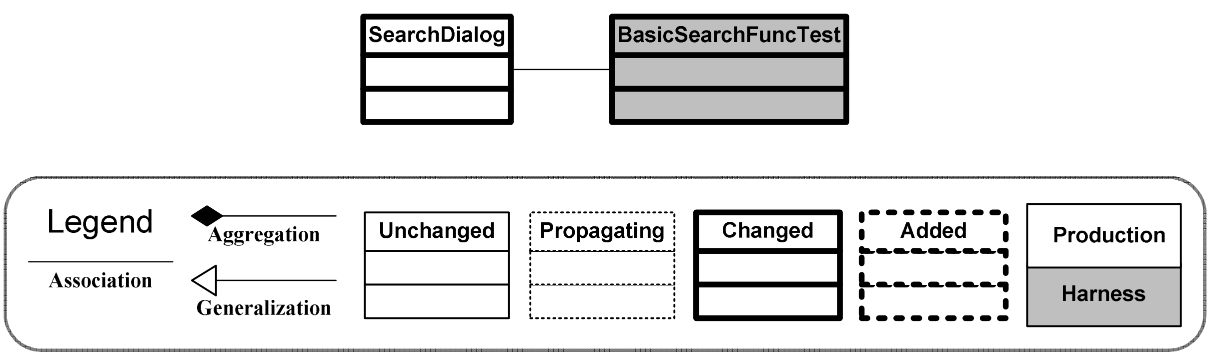


Figure A.4 Change 1 Postfactoring UML

A.1.6.1 SearchDialog class

The programmer updated the Javadoc of this class.

A.1.6.2 BasicSearchFuncTest class

This class uses the Java robot to automate functional tests. The robot can run so fast that the programmer cannot tell what the test is doing, to assist with actualization, the programmer added delays to the test. Those delays were removed during postfactoring.

A.1.7 Verification

Functional and Unit testing was added to the code for the new search functionality. During verification no bugs were found. This is most likely due to the simple nature of the request. There is an issue with the single functional test in `BasicSearchFuncTest`. It runs and passes its assertions but ends displaying a gray or unfinished result. The programmer was unfamiliar with the Abbot GUI Test Framework and decided to address this issue in a future changes. Verification was time consuming; however, because the programmer was unfamiliar with testing in Java. Coverage for each production code file is available in Table A.9.

Table A.9 Change 1 Statement Verification

#	Code File	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	`SearchAction`	7	7	100.0	0	0
2	`SearchDialog`	100	87	87.0	0	0
3	`MainMenuBar`	259	155	59.8	0	0
4	`ToolBarAttributes`	33	3	9.1	0	0
5	`ActionManager`	205	187	91.2	0	0
6	`Translator`	146	69	47.3	0	0

A.1.8 Timing

The Google Desktop Gadget, Task List and Timer worked very well for the first part of the Feature Request. It is a very simple tool that worked well and came with the added benefit of also having a note pad. Unfortunately, it developed an issue after using it for a while. When a task is closed out it is erased immediately and cannot be saved. So all tasked must be paused and left open or the data will be lost. For this reason, the programmer will try Mylyn with Tasktop, a tool for Eclipse during the next change request. Table A.10 contains the timing data for the change.

Table A.10 Change 1 Timing Totals

Phase	Time (hh:mm)
Concept Location	0:22
Impact Analysis	2:08
Prefactoring	0:00
Prefactoring Testing	0:00
Actualization	5:34
Actualization Testing	5:02
Postfactoring	0:23
Postfactoring Testing	0:12

A.1.9 Conclusions

The basic search function is complete. The feature is very simple and it is likely that it will not have enough functionality for many users. It is a good start for a fully functional search feature.

Table A.11lists the totals for each set of code files for each change request of this iteration to date. The current state of the product backlog is in Table A.12. Figure

A.5 to Figure A.7 are screen shots of muCommander showing the change request functionality.

Table A.11 Change 1 Code File Summary

#	Change	Number in Code Files						
		Visited Concept Location	Estimated Impact Set	Changed Set	Added during			Total Project
					Pre	Act	Post	
0	Original Baseline	N/A	N/A	N/A	N/A	N/A	N/A	1,070
1	Basic Search	5	3	4	0	4	0	1,074

Table A.12 Change 1 Current Product Backlog

#	Title	Complete	User Story
1	Basic Search	x	Add a basic search function that allows a user to search in the current directory for all or part of the title of a folder or file, and return a list of the matching files and directories.
2	Recursive Search		Add the ability to search inside all directories.
3	Advanced Output		Change the output to a table similar to the main muCommander window.
4	Date Search		Allow the user search by a date of file's modification.
5	Case Sensitive Search		Add capability to search by case sensitive search terms.
6	Extension Search		Add the ability to search for files with specific extensions.
7	Properties Search		Add options to search for files based on their properties.
8	Size Search		Add the ability to search for a file by its size.
9	Regular Expression Search		Add capability to search by a regular expression.
10	Lucene Search		Incorporate the Apache Lucene search.

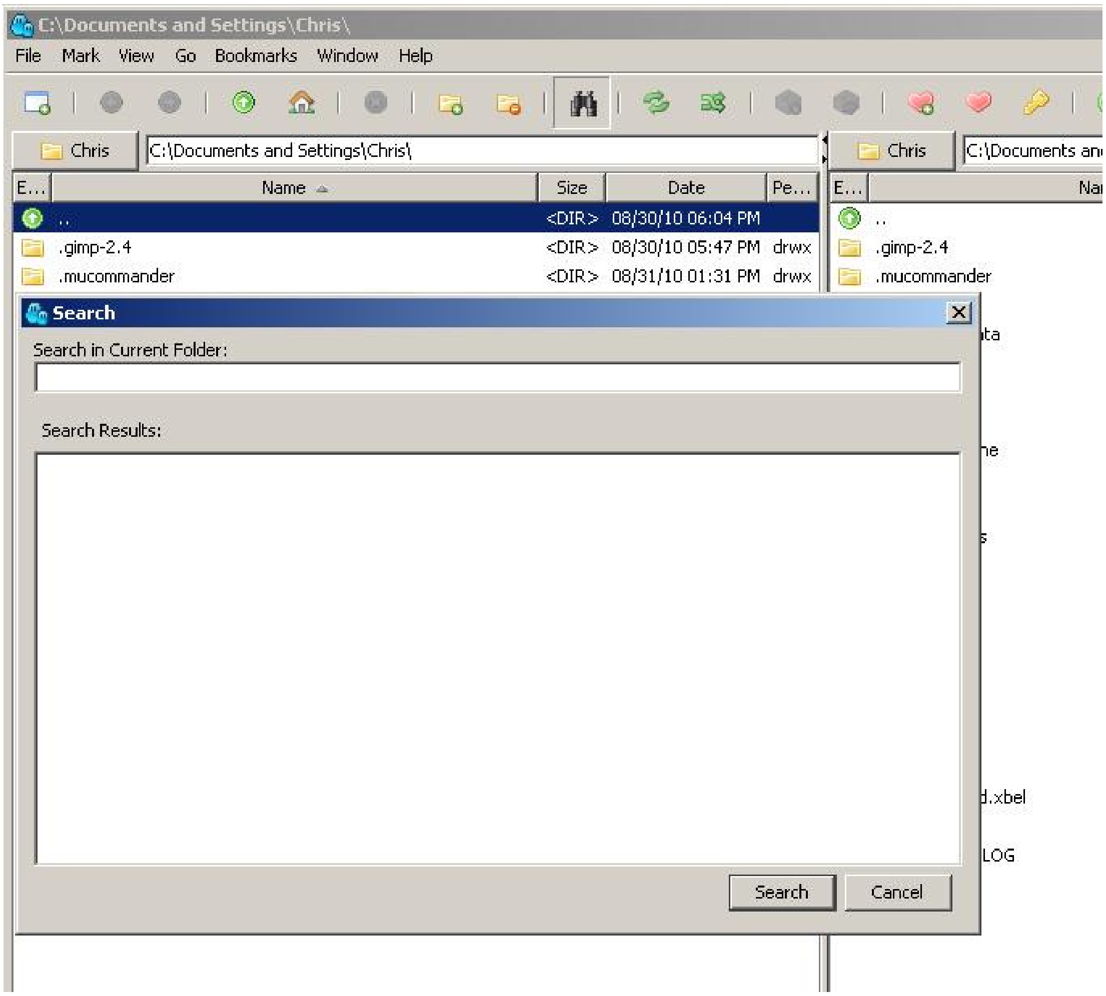


Figure A.5 muCommander with search window

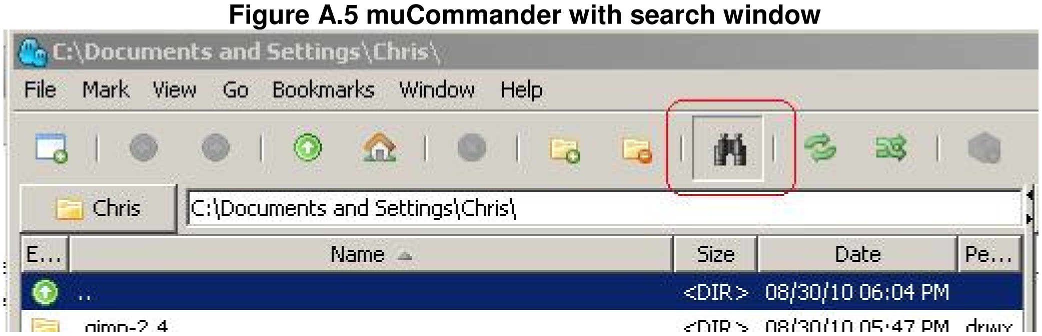


Figure A.6 muCommander Toolbar with Search icon circled

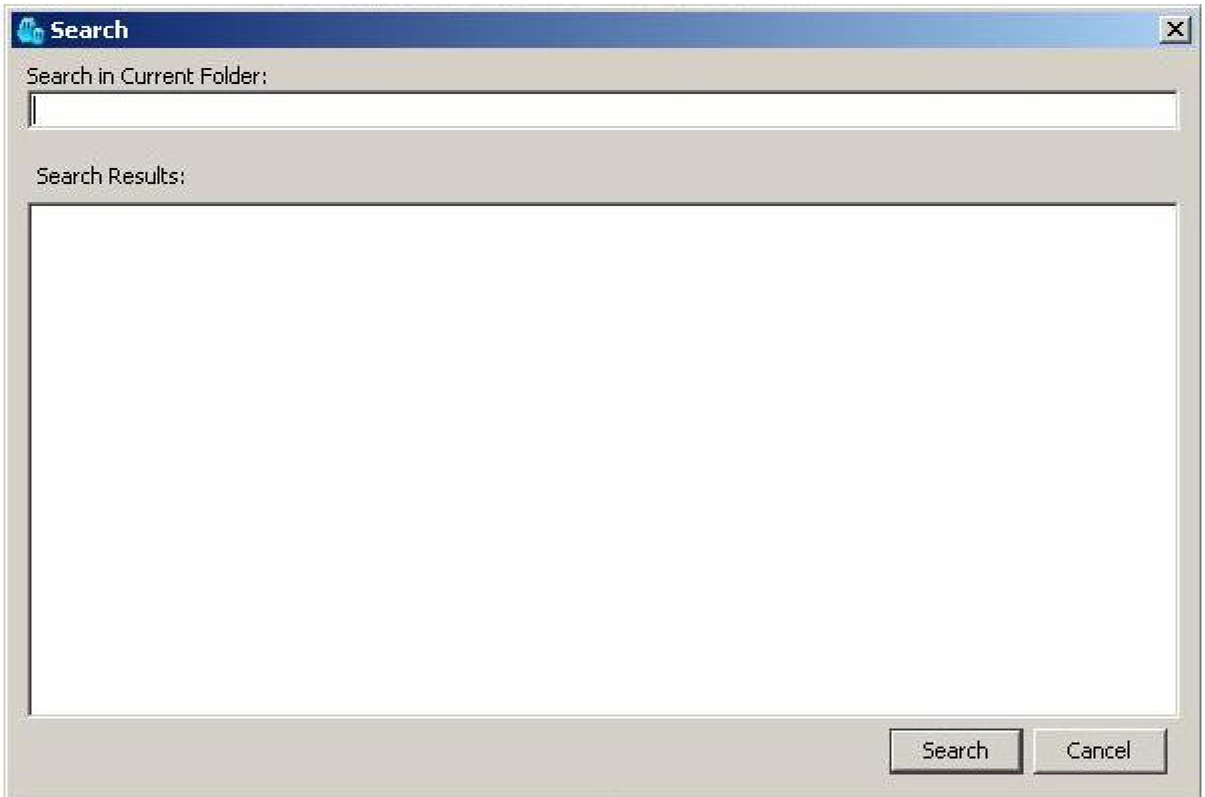


Figure A.7 Basic Search Feature window

SIP – Change 2 Recursive search

A.2.1 Initiation

Add the ability to search inside all directories. The program muCommander is an application which enhances an operating systems file explorer. During the first change request, basic search capabilities were added which helps a user find files in a specific directory.

This change request will add recursive search features to the basic search functionality. The search feature will now have the ability to recursively search the file system, commonly known as searching in subdirectories or searching in subfolders. When the search window opens, it will have the current directory entered as a default, which is basically what the basic search did; however, now the user will also be able to type in a new directory or use a standard GUI window to open any directory in the file system. There is also error checking with messages to help the user select a valid directory to search in.

Finally, an option was added to allow the user to stop the search before it completes and display the partial results. This option is needed for searches in directories that have a large number of directories and files.

A.2.2 Concept Location

The programmer identified the search algorithm as the significant concept extension. No concept location was necessary because he just implement it in change request 1 and knew it was located in `SearchDialog`. Table A.13 contains a summary of the number of each type of class.

Table A.13 Change 2 Concept Location summary

Title	Code Files			Comments
	Visited	Propagating	Unchanged	
Recursive search	0	0	0	Concept located in `SearchDialog` class

.

A.2.3 Impact Analysis

The concept location was found in `SearchDialog` and was labeled as impacted by JRipples. When visiting a class during impact analysis, it was evaluated to see if it would be impacted by the following tasks:

1 – Adding an input box so that the user may specify the directory to search in.

2 – A procedure to provide a way for the user to browse the file system.

3 – Adding error checking techniques to alert the user to the incorrect directory and to stop a search that may cause unintended issues.

4 – A way to choose to search in the subdirectories of the search directory

5 – Display the entire path of each result to the user in the output area

Only the `SearchDialog` class itself and its test classes were found to be impacted. There were no propagations. The `SearchDialog` was created in the first change request of this project. It allowed very basic search functionality. It was just a way to add search functionality to muCommander without the change becoming very large and unmanageable. As such, `SearchDialog` needs some changes performed on it to build it into something that has real value to a user. A UML diagram of all the dependencies listed by JRipples is in Figure A.8.

The estimated impact set contains the `SearchDialog` test class and its test classes, `BasicSearchUnitTest` and `BasicSearchFuncTest`. The number of code files analyzed and their counts are provided in Table A.14. Table A.15 shows the code files visited during impact analysis.

Table A.14 Change 2 Impact Analysis Summary

Title	Code Files					Comments
	Visited	Impacted	Propagating	Unchanged	Not Visited	
Recursive search	19	3	0	16	Recursive search	

Table A.15 Change 2 Impact Analysis Code Files Visited

#	Code File	Tool used	Impacted?	Comments
1	`SearchDialog`	JRipples → Impacted	Impacted	This class contains the current search capability.
2	`BasicSearchUnitTest`	JRipples → Impacted	Impacted	Test class will have to be updated.
3	`BasicSearchFuncTes`	JRipples → Impacted	Impacted	Test class will have to be updated.
4	`AbstractFile`	JRipples →	Unchanged	This is the class with the information on the file

		Unchanged		system.
5	ActionProperties	JRipples → Unchanged	Unchanged	This class is part of the system that manages actions.
6	DialogToolKit	JRipples → Unchanged	Unchanged	This class helps create windows in muCommander look and feel.
7	FileSet	JRipples → Unchanged	Unchanged	This class is a container that holds files.
8	FileTable	JRipples → Unchanged	Unchanged	This class works with FileTableModel to display a directories contents.
9	FileTableModel	JRipples → Unchanged	Unchanged	This class works with FileTable to display a directories contents.
10	FocusDialog	JRipples → Unchanged	Unchanged	This class adds to the basic Swing component JDialog functionality
11	MainFrame	JRipples → Unchanged	Unchanged	This class creates the main window the user sees when muCommander is started.
12	SearchAction	JRipples → Unchanged	Unchanged	This registers the SearchDialog class with muCommander
13	SpinningDial	JRipples → Unchanged	Unchanged	This class is a GUI component.
14	Theme	JRipples → Unchanged	Unchanged	The Theme classes help keep the GUI componenets consistent throughout muCommander.
15	ThemeData	JRipples → Unchanged	Unchanged	
16	ThemeManager	JRipples → Unchanged	Unchanged	
17	Translator	JRipples → Unchanged	Unchanged	This class contains different languages for GUI components.

18	XBoxPanel	JRipples → Unchanged	Unchanged	This class helps create GUI components in muCommander look and feel.
19	YboxPanel	JRipples → Unchanged	Unchanged	This class helps create GUI components in muCommander look and feel.

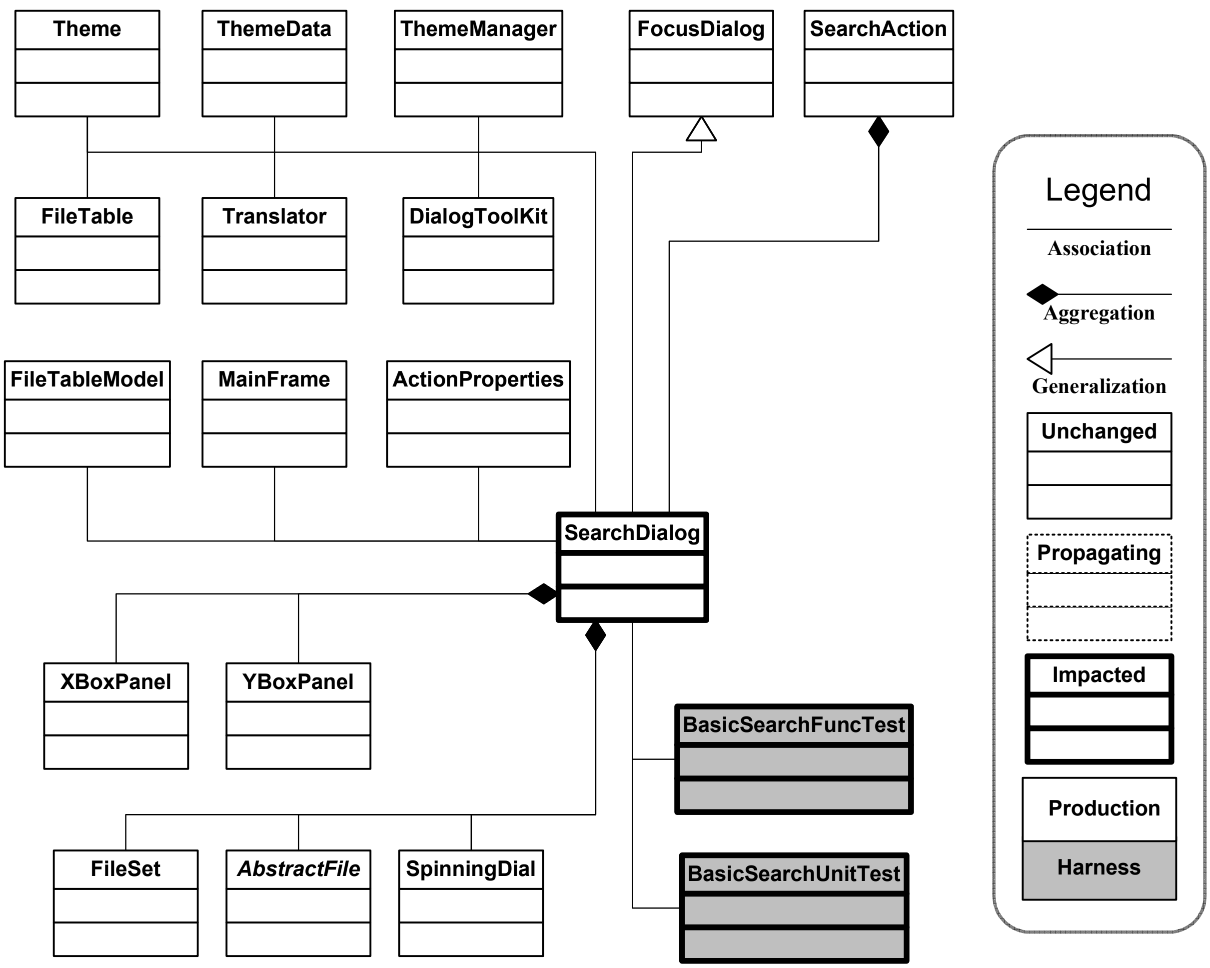


Figure A.8 Change 2 Impact Analysis UML

A.2.4 Prefactoring

In preparation for the implementation of this change request, the programmer extracted 2 classes from `SearchDialog`; which contained the entire search functionality. One class extracted `formSearchDialog`, `SearchThread`, was to remove the logic of the search and another, `InputPanel`, was extracted to remove the GUI features displayed in the top half of the dialog. `SearchDialog` contained as much responsibility as it reasonably could, this will allow those features to grow during this change without any one class becoming cumbersome. Also, by separating the search logic from the GUI components, it will be possible to have the logic run in a separate thread. This way the GUI can still respond to user input while the search is being run.

The programmer also extracted 2 test classes from `BasicSearchUnitTest`. The first, `SearchThreadTest` contains the tests for `SearchThread` and the second `InputPanelTest` contains the tests for `InputPanel`.

The programmer modified the `ShutdownHook` class so that the functional tests could be extended. This class was not identified during impact analysis. During regression testing the programmer realized that the issue with the functional test, which is it would pass its assertions, but display a gray instead of green color, was that somewhere a `System.exit()` call was being made and this was stopping JUnit from completing the test. The programmer did a grep search and found that only the classes `Launcher` and `ShutdownHook` contained this call. Launcher only made the call, if the program could not be started, so a method was added to `ShutdownHook` to allow the program to be shut down without calling `System.exit()`. The functional test then passed. This increased the change set to 4 classes.

A table with the count of each type of class is in Table A.16. Additionally, a summary of each refactored class is in Table A.17. A UML showing the significant relationships of this refactoring is in Figure A.9.

Table A.16 Change 2 Prefactoring Summary

Title	Code Files					
	Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
Recursive search	4	4	4	0	0	1

Table A.17 Change 2 Prefactoring Code Files

#	Code File	Task	Lines of Code		
			Added	Deleted	Total
1	`SearchThread`	Extracted class	16	0	16
2	`InputPanel`	Extracted class	39	0	39
3	`SearchDialog`	Extracted class from	33	101	134
4	`SearchThreadTest`	Extracted class	75	0	75
5	`InputPanelTest`	Extracted class	49	0	49
6	`BasicSearchUnitTest`	Renamed class & Classes extracted from	51	48	99
7	`BasicSearchFuncTest`	Extracted method	46	73	119
8	`ShutdownHook`	Changed & modified method	7	1	8

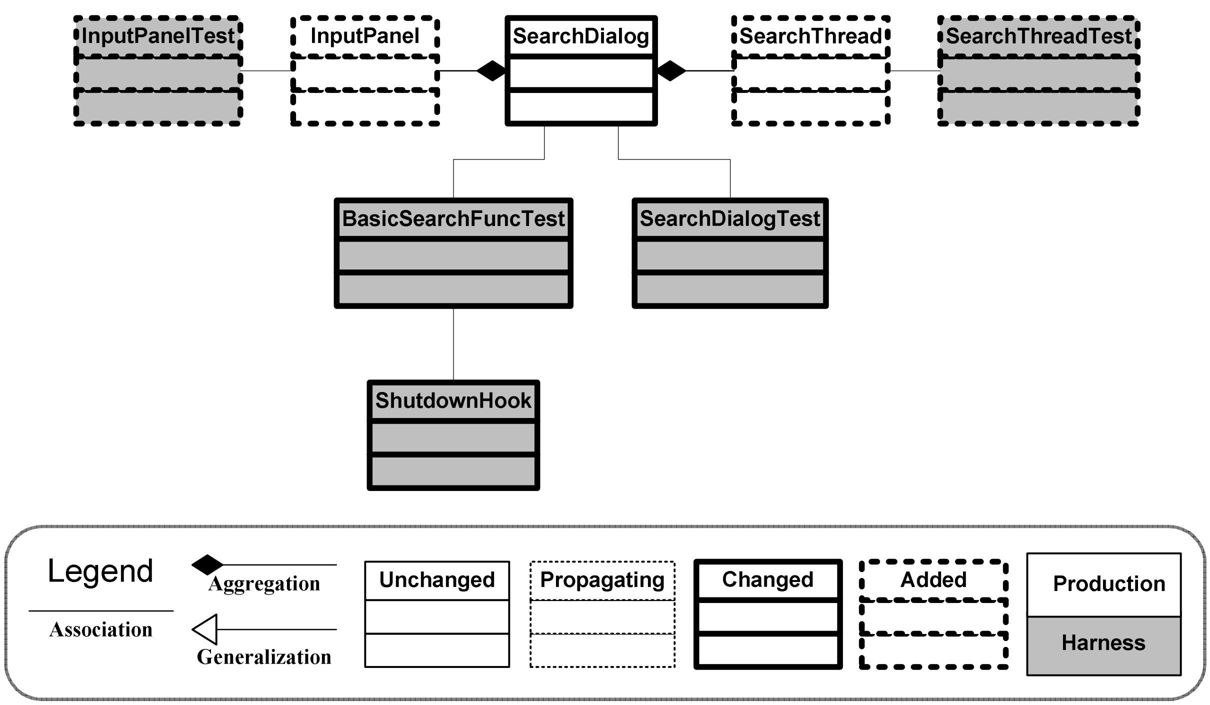


Figure A.9 Change 2 Prefactoring UML

A.2.4.1 SearchThread class

The class extraction consisted of moving the part of the `searchCommand()` method that searches the file system, to the new class. Eight LOCs were removed from the original 49 line method `searchCommand()` in `SearchDialog`. The method was then refactored again to inline variables. It is now 10 LOCs and is the only method in `SearchThread`.

A.2.4.2 InputPanel class

This class extraction consisted of moving the `createInputArea()` method from `SearchDialog` and its 14 LOCs to a new class that inherits from the return type, `XBoxPanel` of the method. Getters for the GUI input box were also needed for `SearchDialog`'s `searchCommand()` method. A new data member of type `InputPanel`, named `inputPanel` was added to `SearchDialog`. It was then initialized in the `SearchDialog` constructor. The data member `inputBox` was also moved to `InputPanel`, so getters were substituted for it. The class has 5 methods, 1 is for testing.

A.2.4.3 SearchDialog class

Eight LOCs were removed from the original 49 line method `searchCommand()`, to extract the `SearchThread` class. A field type of `SearchThread` was added to `SearchDialog`. The `InputPanel` class extraction removed a method, `createInputArea()` and a data type, `inputBox`, but added a data type of `InputPanel`.

The `switchToSearchState()` method added a `boolean` parameter, so it can now enable or disable the search state. The `searchCommand()` method now calls this method to disable the search state. This removed another 4 LOCs from `searchCommand()`; it is now 16 LOCs. The class now has 13 methods, 5 are for testing.

A.2.4.4 SearchThreadTest class

This test class was extracted from `BasicSearchUnitTest`. One test was extracted from the `testSearchCommand()` method. It was then divided into 2 tests, 1 for a file that existed and should be found and 1 that did not exist that should not. A test for the constructor was also added for a total of 3 tests.

A.2.4.5 InputPanelTest class

This test class was also extracted from `BasicSearchUnitTest`. One test was extracted from the `testSwitchToSearchState()` method. It tests the `switchToSearchState()` method that was extracted from `SearchDialog`'s `switchToSearchState()`. Tests for the constructor and getters were also added for a total of 4 tests.

A.2.4.6 BasicSearchUnitTest

This test class had the test functionality for the `SearchThread` and `InputPanel` classes removed. It now contains 5 tests. Since all test are aimed at the `SearchDialog` class, it was renamed `SearchDialogTest`.

A.2.4.7 BasicSearchFuncTest

This test class had 1 test divided into 2, or 1 extracted from the first test. One test tests for a search that returns a result and the other for a search that returns no results. This will make diagnosing future bugs easier.

The `setUp()` method was also refactored, changing some of the Abbot finder calls to getters that already exist for the unit test. This makes the code easier to read and faster.

A.2.4.8 ShutdownHook class

This class was modified to allow for multiple functional tests. The abbot functional test suite could not close the program without this class calling `System.exit()`, which causes JUnit to stop running tests. A type, new constructor and if statement were added to stop the `System.exit()` call when desired.

A.2.5 Actualization

To add the recursive search capabilities, no new classes were added after the prefactoring and the change did not propagate to any other classes. A summary of the change propagation is in Table A.18. The change did require substantial new code to be added to the `SearchDialog`, `SearchThread` and `InputPanel` classes along with their test classes. Each class actualization is summarized in Table A.19. A UML diagram showing the relationships of the actualization is in Figure A.10.

Table A.18 Change 2 Actualization Summary

Title	**Code Files**					
	Visited	**Changed**	**Added**	**Propagating**	**Unchanged**	**Added to Changed Set**
Recursive search	7	7	0	0	0	0

Table A.19 Change 2 Actualization Code Files

#	Code File	Task	Lines of Code		
			Added	Deleted	Total
1	`SearchDialog`	Added methods	62	18	80
2	`SearchThread`	Added Inheritance & methods	36	10	46
3	`InputPanel`	Added methods	201	39	240
4	`SearchDialogTest`	Added and modified tests	64	6	70
5	`SearchThreadTest`	Added and modified tests	81	75	156
6	`InputPanelTest`	Added and modified tests	106	49	155
7	`BasicSearchFuncTest`	Added and modified tests	30	4	34

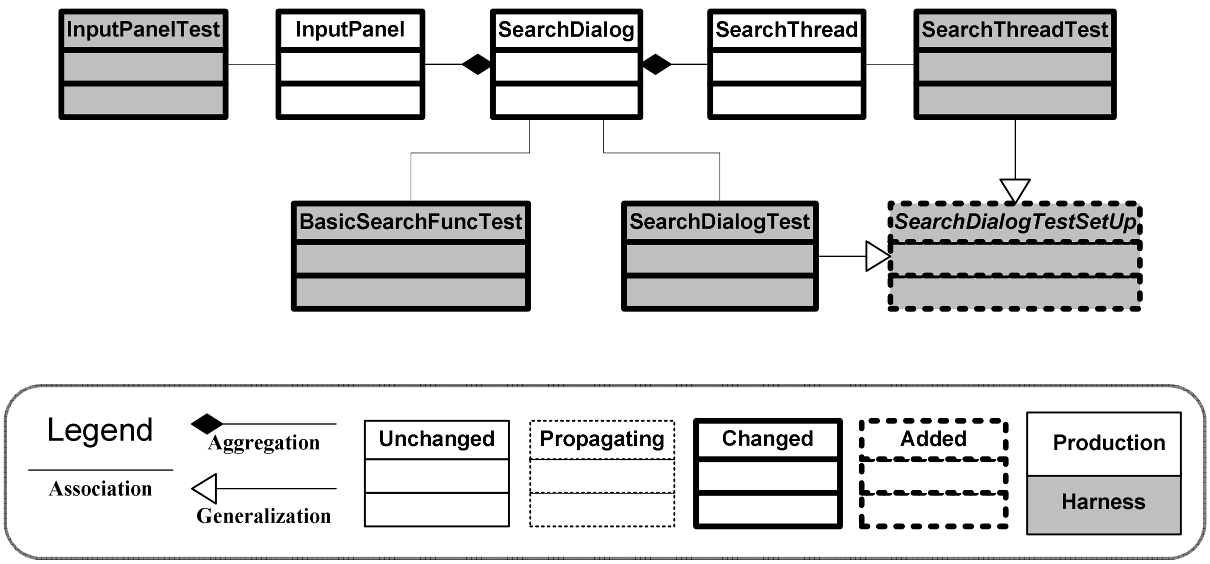


Figure A.10 Change 2 Actualization UML

A.2.5.1 SearchDialog Class

The `SearchDialog` class at the start of the change held the entire search functionality. However, after the search method was extracted from it, it became the user interface class for the search functionality. The input panel was also extracted; it now contains the output and search and cancel buttons.

A `boolean` field was added to notify `SearchThread` if the user stops a search in progress. This is effectively a thread kill, which was deprecated in Java.2. There are also methods added that SearchThread can call to add search results to the output area, notify SearchDialog that a search has completed and to display an error.

Searches that search recursively can be much longer, so the button that starts a search, searchButton had the capability to stop an in progress search added to it. The cancel button that closes the window also had this capability added.

The capability to add results as they are found was added by extracting a method from searchCommand() and changing its parameter from a FileSet as to a single AbstractFile.

A method called notifyEnd() was added for SearchThread to notify SearchDialog that it had completed the search. The method changes the SearchDialog back to the search state and displays a message to the user if the search returned no results.

A method was added that displays any errors to the user in the same box as the results. It is called by the searchCommand() method if there is an Exception during the SearchThread creation or by the SearchThread if there is an Exception while searching.

These were the major parts added to the SearchDialog class during actualization.

A.2.5.2 SearchThread Class

The SearchThread class was extracted from the SearchDialog class during prefactoring. The search method created during prefactoring replaced its search code with a recursive method all to add the recursive capability to it, so it can search in subdirectories.

The class extraction was done in prefactoring, which defined the basic class responsibility. The class was made to extend the Thread class, allowing it to run in its own thread. This required the addition of a constructor that initializes 4 fields and 2 other methods, `main()` and `run()`.

Also, the `searchCommand()` method was made recursive, so that it can search in directories. A helper method of the same name was added to provide the recursive method with the initial directory to search and the term to search for.

A.2.5.3 InputPanel class

The first part added was an interface for the user to choose a directory to search in. At the start of actualization when the `InputPanel` class was instantiated, it would only search in the directory defined as the current directory by the `MainFrame` class. Now the user can choose the directory, but the default is still the current directory as defined by the `MainFrame` class. This required a parameter be added to the constructor so the directory field can start in the current directory.

To choose a directory the user can either type out a path or choose one through another dialog that is a standard Java dialog. If the user types an invalid directory, error checking is in place so a search cannot start unless a valid directory is entered. Basically, the `AbstractFile` class that was used in the first change has a method that returns true if a path is valid. `SearchDialog` checks for a valid directory when user moves the cursor off the input line. If the directory is invalid a red “Invalid Directory” error appears and a search will not start. If the user then inputs a valid path the error will disappear and the search capability will become re-enabled. To accomplish this, listeners were added for focus events and key events, along with the GUI components to display the error message.

Also added was a box which the user can check or uncheck to include or not include subdirectories in their search. When a search is initiated the box is inspected for the presence of a check and the search acts appropriately.

Five fields were added that display the directory field, the button to open another dialog to browse for the start directory, a label with an error to be displayed if an invalid directory is typed in, a checkbox to turn the recursive mode on and off and a `JPanel` to organize the components. These fields are initialized in the constructor or a `createDirectoryArea()` method that is called by the constructor. They were also added to the `setEnabled()` method so they can be disabled during searches and enabled after the search is over. A method `isRecursive()` was added that just returns true if the recursive checkbox field is selected.

The methods `chooseFile()`, `isvalidDirectory()`, `isErrorEnabled()` and `getDirectory()` were added. The `chooseFile()` method opens a `JFileChooser()` when the browse button is pressed and `isErrorEnabled()` returns true if the error is visible to the user. The method `isValidDrectory()` checks to make sure a valid directory is entered in the directory field and `getDirectory()` takes the `String` from the directory and retrieves the `AbstractFile` associated with it.

Five additional fields were created in the class that flash the invalid directory error to the user if the user tries to search without entering a valid directory. These fields are either initialized when declared or in the constructor. The methods `flashError()`, `actionPerformed()`, `focusLost()` and `keyReleased()` were added. The `flashError()` method starts a `Timer`. When the `Timer` goes off, the `actionPerformed()` method alternates the error label form visible to invisible. The `focusLost()` makes the error visible if the user leaves the directory field with an invalid directory entered. The `keyReleased()` method will turn the error off if the user enters a valid directory.

A.2.5.4 SearchDialogTest class

Three tests were modified to work with the new search process. Five new tests were added to test the new methods added to `SearchDialog` to communicate with `SearchThread`.

A.2.5.5 SearchThreadTest class

The 2 existing tests were modified to allow for searching with the new thread capability. A test was added to test the new recursive capability.

A.2.5.6 InputPanelTest class

Seven tests were added to test the new components and functionality added to the `InputPanel` class. One test was modified to include testing for the new components.

A.2.5.7 BasicSearchFuncTest class

Two tests were added, one to test the recursive search capability and one to test the invalid directory error. The 2 existing tests had to be modified to enter a directory as is now required.

A.2.6 Postfactoring

After finishing the actualization stage and the feature was up and running, but the code needed to be refactored because of the actualization. This consisted mainly of cleaning up the code and adding getters and setters for the verification process. The `InputPanel` class had grown too large and had too much responsibility. The `DirectoryPanel` and `FlashLabel` classes were extracted from it. To keep the test suite organized the tests in `InputPanelTest` that test methods extracted to these new classes were moved into new test classes `DirectoryPanelTest` and `FlashLabel`. In `SearchDialogTest` and `SearchThreadTest` the 4 methods that setup and teardown for the tests were very similar; they were extracted to a new abstract class `SearchDialogTestSetUp`.

Finally, to better organize the project, 3 new packages were created: `org.severe.ui.dialog.search.panels`, `org.severe.ui.dialog.search.tests` and `org.severe.ui.dialog.search.panels.tests`. Then the appropriate classes were placed into each package.

A summary of postfactoring is available in Table A.20 and a summary of postfactoring changes of each class is in Table A.21. A UML diagram of the postfactoring class relationships is in Figure A.11.

Table A.20 Change 2 Postfactoring Summary

Title	Code Files					
	Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
Recursive search	7	7	5	0	0	0

Table A.21 Change 2 Postfactoring Code Files

#	Class	Task	Lines of Code		
			Added	Deleted	Total
1	`SearchDialog`	Extracted method from	42	42	84
2	`SearchThread`	Rename method	3	3	6
3	`InputPanel`	Extracted class from	12	152	164
4	`DirectoryPanel`	Extracted class	126	0	126
5	`FlashLabel`	Extracted class	42	0	42
6	`SearchDialogTest`	Extracted super class from	46	77	123
7	`SearchThreadTest`	Extracted super class from	15	52	67
8	`InputPanelTest`	Extracted class	9	59	68
9	`DirectoryPanelTest`	Extracted class	88	0	88
10	`FlashLabelTest`	Extracted class	34	0	34
11	`SearchDialogTestSetUp`	Extracted super class	51	0	51
12	`BasicSearchFuncTest`	Javadoc	23	18	41

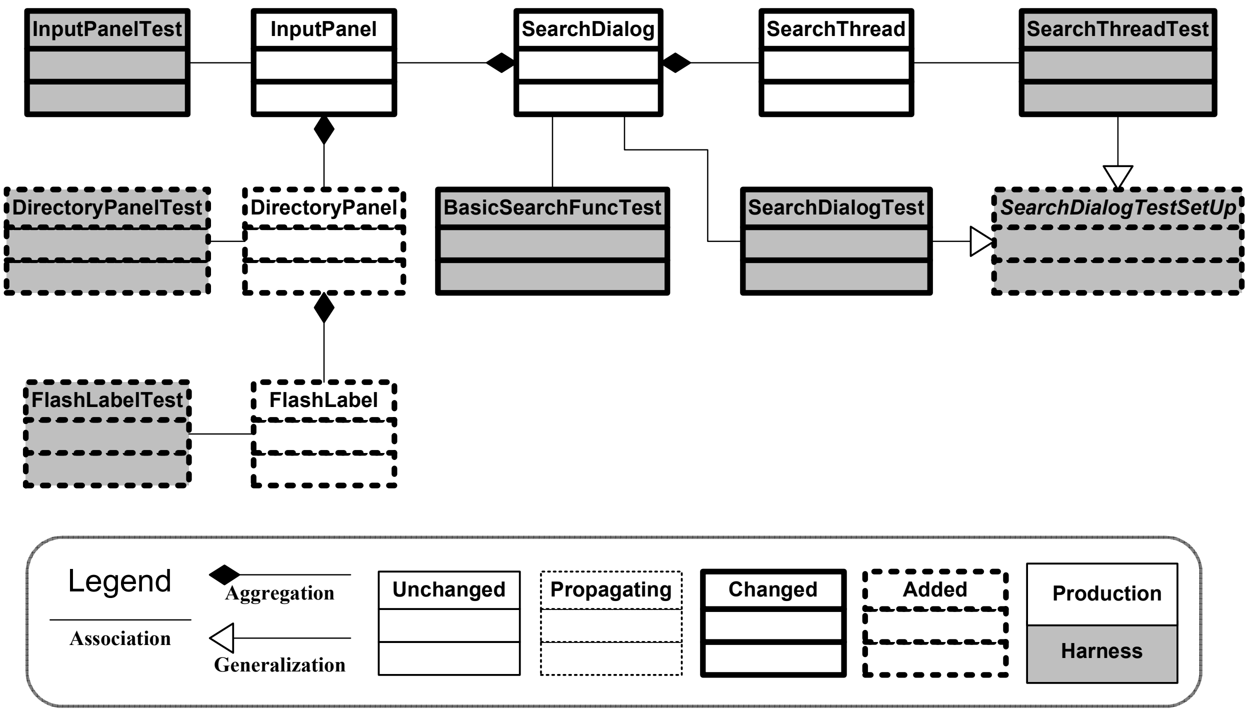


Figure A.11 Change 2 Postfactoring UML

A.2.6.1 SearchDialog class

This class had a method extracted, a field renamed and Javadoc updated. The method `stopSearchThread()` was extracted from `actionPerformed()`. It replaced duplicated code activated when the cancel button or search button were pressed. The field `searchButton` was renamed `searchStopButton`, to better reflect the functionality that was added during actualization.

A.2.6.2 SearchThread class

One method was renamed. The method `searchCommand()` with parameters `AbstractFile` and `String` was renamed to `recursiveSearch` with the same parameters. This method gained recursive functionality during actualization and this new name better reflects that.

A.2.6.3 InputPanel class

This class was moved from the `org.severe.ui.dialog.search` package to the new `org.severe.ui.dialog.search.panels` package. It also had 2 classes extracted, which included 10 fields extracted, 1 field added, 16 methods extracted 3 methods modified and all of the interfaces it implemented were also removed with the class extractions. The classes `DirectoryPanel` and `FlashLabel` were the classes extracted.

A.2.6.4 DirectoryPanel class

This class was extracted from `InputPanel`. It is located in the new `org.severe.ui.dialog.search.panels` package. It contains the text field that the user enters a directory to search in, a button for the user to open a dialog to select a directory from the file system and a text label of type `FlashLabel` that displays an error to the user when an invalid directory is entered. This class implements the interfaces `ActionListener`, `KeyListener` and `FocusListener` and implements the methods required by these. It has 2 methods to layout the GUI components, an overridden `setEnabled()` method and the methods `isValidDirectory()`, `getDirectory()`, `flashError()` and `isErrorEnabled()` all extracted from `InputPanel`.

A.2.6.5 FlashLabel class

This class was extracted from `InputPanel`. The object of its type is contained in `DirectoryPanel`. Its class is located in the new `org.severe.ui.dialog.search.panels` package. It implements the `ActionLisener` interface. It is an extension of the swing `JLabel` class. It adds a method `flash()`, which will cause the label to flash to notify the user that corrective action is necessary. It accomplishes this by using the Timer class to set itself visible or not visible, when the `flash()` method is called.

A.2.6.6 SearchDialogTest class

This class had a super class `SearchDialogTestSetUp` extracted from it, had 2 fields added to replace numerous inline calls. The super class extracted removed the `oneTimeSetUp()`, `setUp()`, `oneTimeTearDown()` and `tearDown()` methods. The `setUp()` method was only partially extracted, this class still contains an implementation that calls the super constructor. Two fields were also extracted to the new super class.

A.2.6.2.7 SearchThreadTest class

This class also had the super class `SearchDialogTestSetUp` extracted from it, which included removing the same methods as `SearchDialogTest` and removing 1 field. Also, 1 test was modified to inline a method call.

A.2.6.8 InputPanelTest class

This class had the test classes `DirectoryPanelTest` and `FlashLabelTest` extracted. This included 7 tests and was done to keep the tests organized. An inline method call used by one of the tests was also updated to a new name.

A.2.6.9 DirectoryPanelTest class

This class was extracted from `InputPanelTest`. It contains 8 tests, 6 of which were extracted from `InputPanelTest`.

A.2.6.10 FlashLabelTest class

This class was extracted from `InputPanelTest`. It contains 3 tests, one of which was extracted from `InputPanelTest`.

A.2.6.11 SearchDialogTestSetUp abstract class

This super class was extracted from `SearchDialogTest` and `SearchThreadTest`. It contains 4 methods `oneTimeSetUp()`, `setUp()`, `oneTimeTearDown()` and `tearDown()`. These methods create an instance of the `SearchDialog` class that can be used to test it or its components. The code to do this was repeated in both classes, so it made more sense to put it in its own class that can be extended. It contains 3 fields.

A.2.6.12 BasicSearchFuncTest class

This class had 2 fields added to replace numerous long inline method calls. This caused all 4 of its tests to be modified.

A.2.7 Verification

Unit tests expanded from 1 class to 5 plus a super class. A total of 42 new tests were added to test the new functionality, 15 were deleted and 23 modified. The functional tests were also expanded, but remained in 1 class. During verification 3 bugs were found.

Two tests were added to check for proper behavior of the GUI components with a variety of user inputs. Two bugs were found as a result of this testing.

In the case when a user inputs a blank value for the directory an error message would appear, but when the test tried to type in a valid directory it would be redirected to another input location before it could complete. The automated testing was stopped and the defect was manually confirmed. Then, upon code inspection, the bug was identified, when a user went back to enter a correct directory an exception was being thrown. An error handling method, `setError()` was causing this unwanted input redirection, when it was called from the exception catch. Now the exception is not caught because the user needs a chance to enter a valid directory. If the user does not enter a valid directory the error will be caught and handled later.

The second bug discovered, was again an exception throwing error. There can be certain directories that the file system marks as readable, but are set as read-only through a different mechanism. An example of this is a quarantine directory used by an antivirus program. When the search ran into this type of directory, it throws an exception. Code was added to catch this exception which stopped the search. This gave an unwanted behavior of stopping the search when valid results might still be possible. The `setError()` method was altered to handle the exception by just printing a message to the user with the directory path that was not searched, but continue the search to the rest of the file system.

The unit test classes were organized so that there is a test class for each class added. Furthermore, the test classes were placed in their own packages with the same name as the class that are directed at with tests appended to the end. This was done to facilitate removal for a release.

By modifying the tests from change 1 Basic Search it was realized a message displayed to the user that there were no search results found, was no longer functioning. The message was re-enabled, so that the user would know that the search had run without a match. The original 2 tests passed after prefactoring, `testSwitchToSearchState()` and `testSearchDialog()` were not modified; however, `testSearchCommand()` had to be reworked for the new functionality. Coverage for each production code file is available in Table A.22.

Table A.22 Change 2 Statement Verification

#	Code File	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	SearchDialog	81	76	93.8	0	1
2	SearchThread	19	19	100.0	0	1
3	InputPanel	29	29	100.0	0	0
4	DirectoryPanel	52	41	78.8	0	1
5	FlashLabel	14	14	100.0	0	0
6	ShutdownHook	41	4	9.8	0	0

A.2.8 Timing Data

Table A.23 contains the timing data for the change.

Table A.23 Change 2 Timing Totals

Phase	Time (hh:mm)
Concept Location	0:00
Impact Analysis	2:28
Prefactoring	1:22
Prefactoring Testing	2:43
Actualization	3:41
Actualization Testing	1:52
Postfactoring	2:57
Postfactoring Testing	7:34

A.2.9 Conclusions

The recursive search change is complete. It makes the overall search feature much more useful. The overall feature does need more to be at the level users expect, but the next few changes should make a large difference.

This change included more refactoring then the first change. The prefactoring for this change prepared the code for the change. The change would have been difficult without refactoring, extracting the SearchThread class made it easier to add a separate thread to search the file system. Without this refactoring, SearchDialog would have been suffered from code decay; it would have been large and had many responsibilities.

The changed set was 4 classes, 1 larger than the estimated impact set, because a class, ShutdownHook, needed a method added so that the functional tests could finish running. During the change the programmer discovered why the functional test had displayed gray during change 1 and added a workaround as described in the prefactoring phase.

Table A.24 summarizes the number of classes for the different phases of the change. Table A.25 is the current state of the product backlog. Figure A.12 to Figure A.16 are screen shots of before and after the change request.

Table A.24 Change 2 Code File Summary

#	Change	Number in Code Files						
		Visited Concept Location	Estimated Impact Set	Changed Set	Added during			Total Project
					Pre	Act	Post	
0	Original Baseline	N/A	N/A	N/A	N/A	N/A	N/A	1,070
1	Basic Search	5	3	4	0	4	0	1,074
2	Recursive search	0	3	4	4	0	5	1,083

Table A.25 Change 2 Current Product Backlog

#	Title	Complete	User Story
1	Basic Search	x	Add a basic search function that allows a user to search in the current directory for all or part of the title of a folder or file, and return a list of the matching files and directories.
2	Recursive Search	x	Add the ability to search inside all directories.
3	Advanced Output		Change the output to a table similar to the main muCommander window.
4	Date Search		Allow the user search by a date of file's modification.
5	Case Sensitive Search		Add capability to search by case sensitive search terms.
6	Extension Search		Add the ability to search for files with specific extensions.
7	Properties Search		Add options to search for files based on their properties.
8	Size Search		Add the ability to search for a file by its size.
9	Regular Expression Search		Add capability to search by a regular expression.
10	Lucene Search		Incorporate the Apache Lucene search.

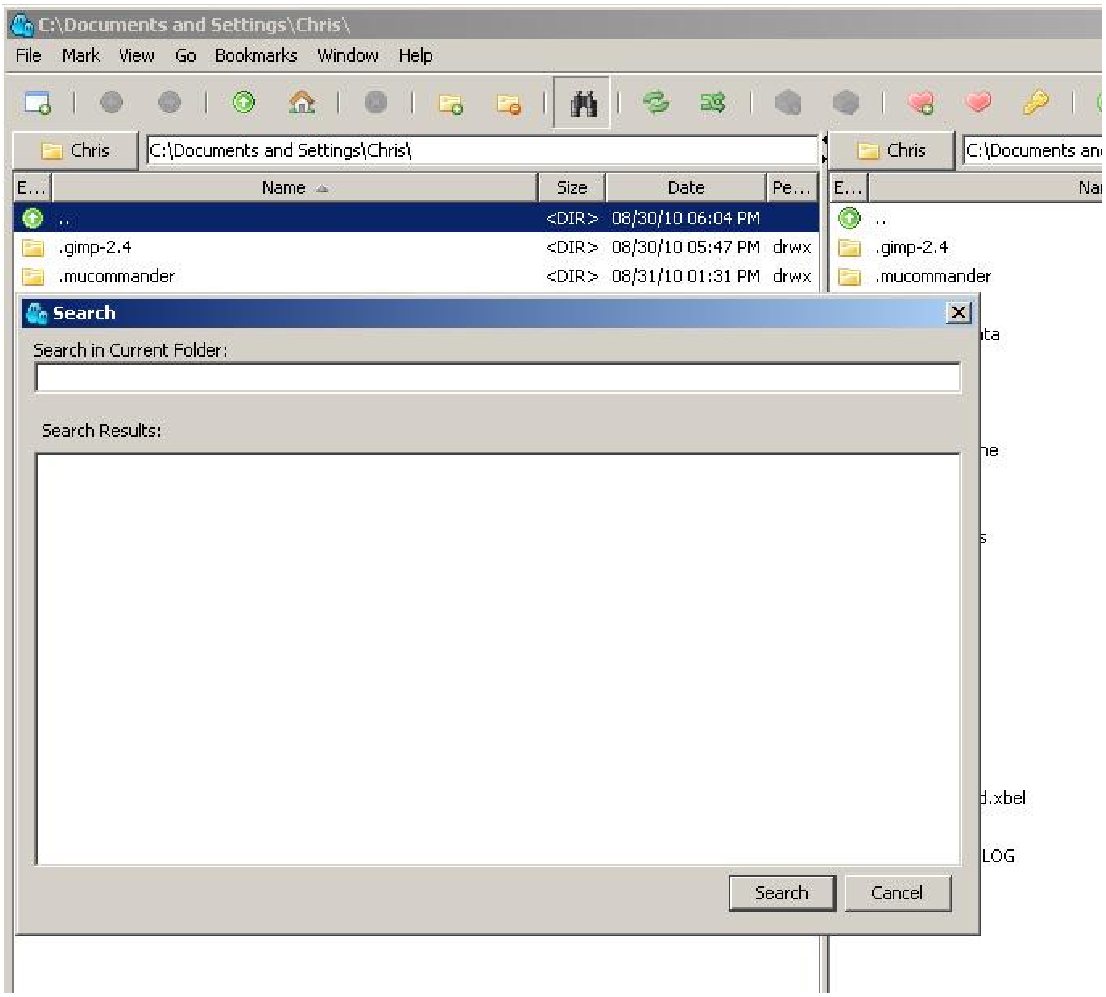


Figure A.12 Search window before Recursive search Change

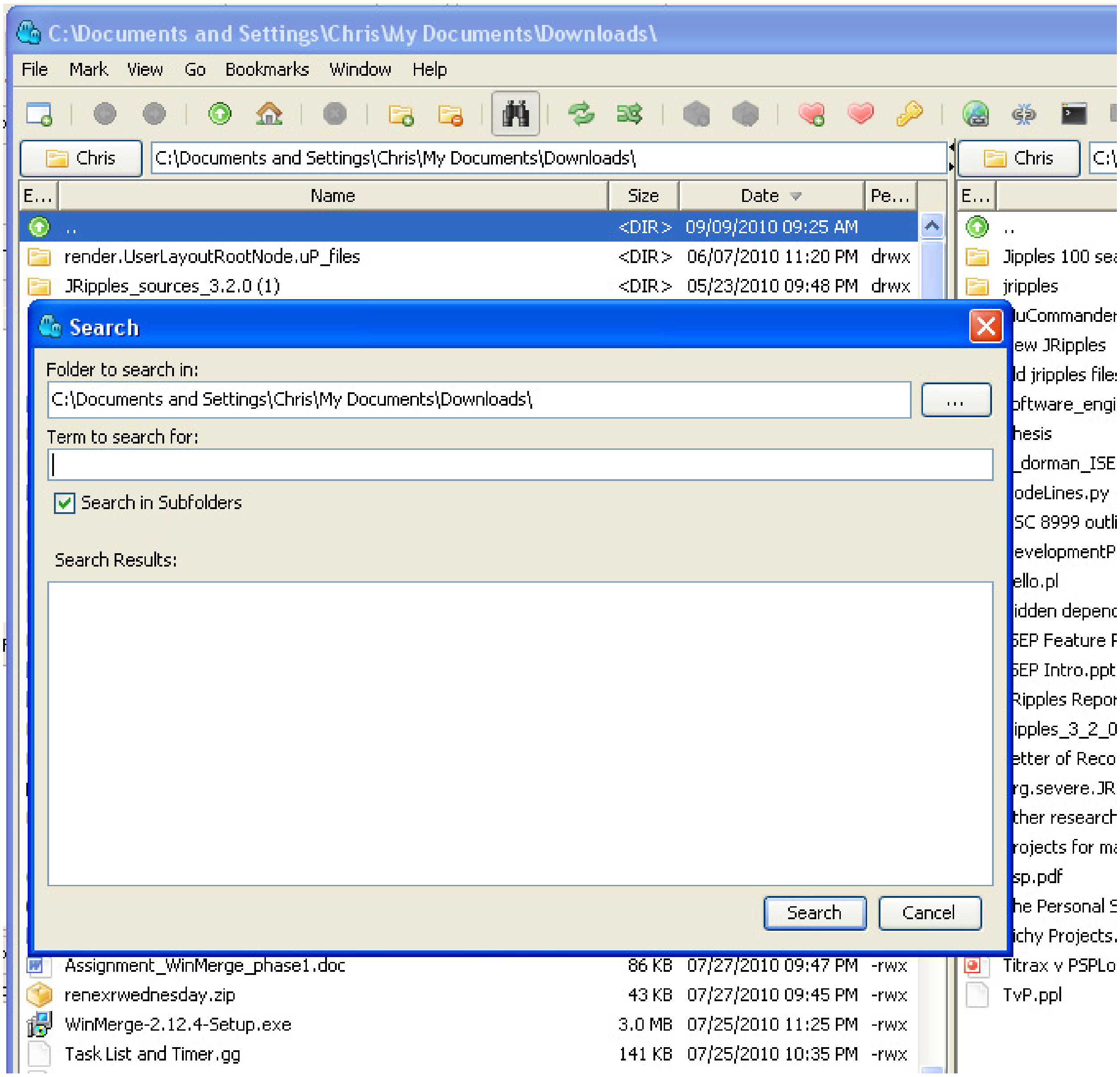


Figure A.13 Search window after Recursive search Change

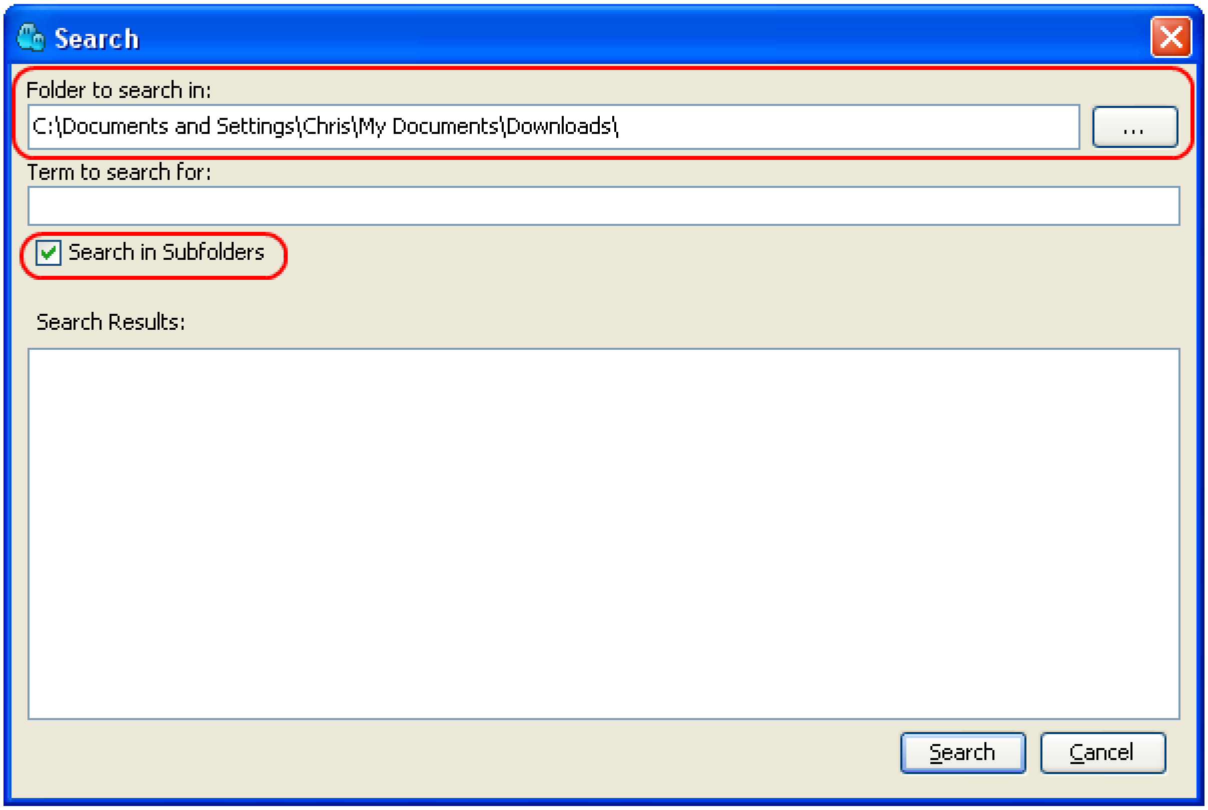


Figure A.14 Search window with new input features circled

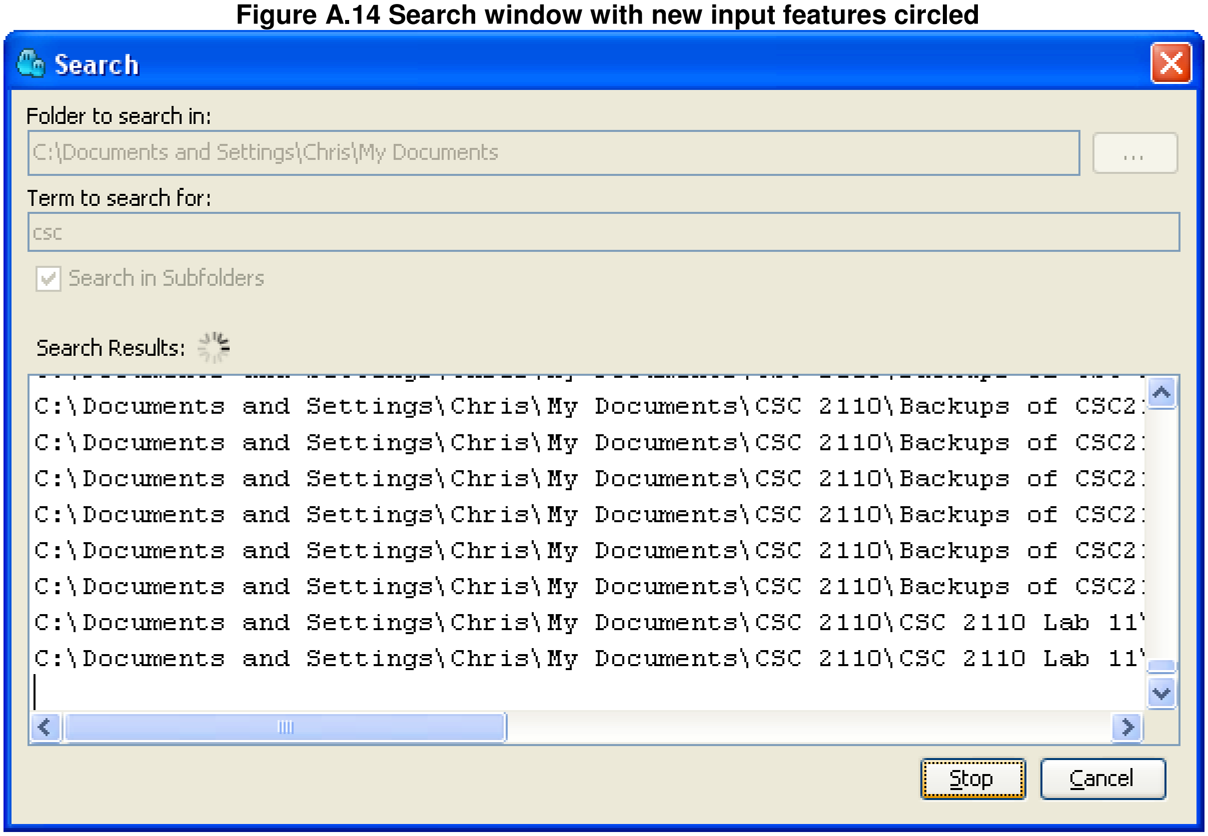


Figure A.15 Search window with search running

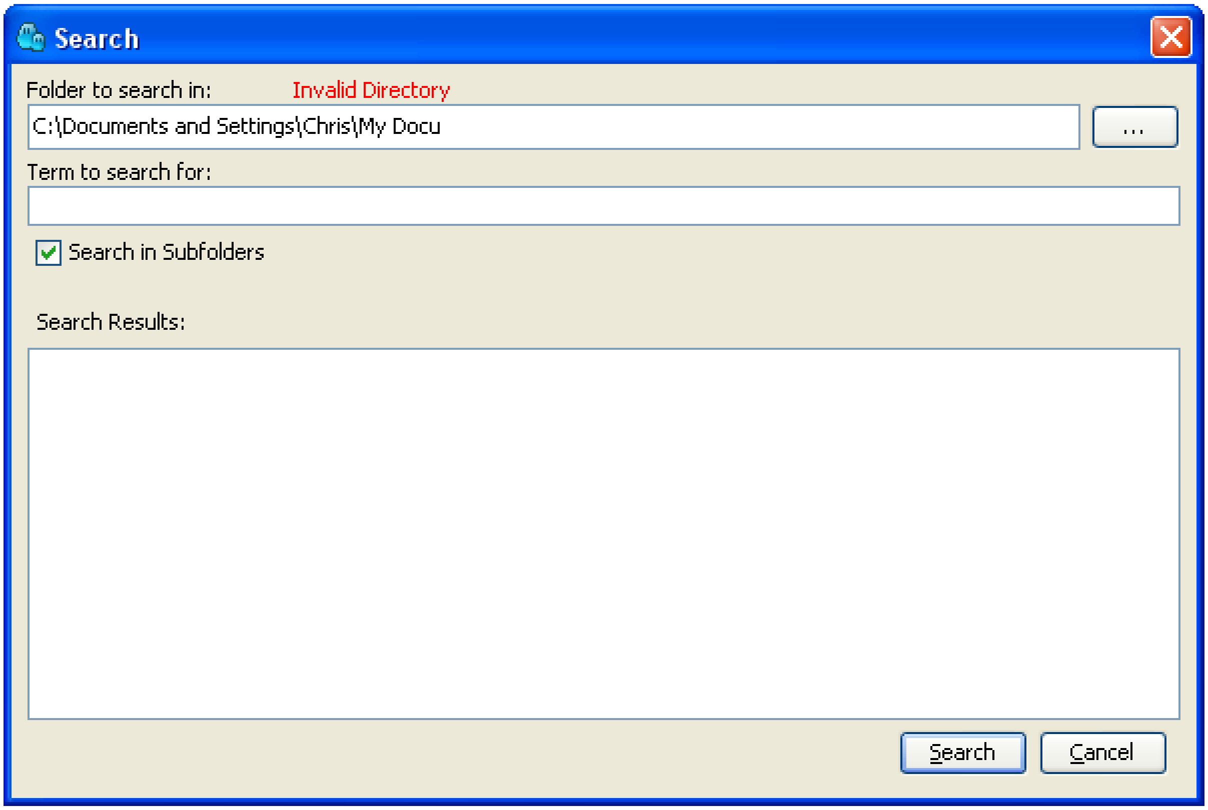


Figure A.16 Search window with invalid directory error message

SIP – Change 3 Advanced Output

A.3.1 Initiation

Change the output to a table similar to the main muCommander window. It is an application which enhances an operating systems file explorer. During the first change request, basic search capabilities were added; which helps a user find files in a specific directory. For the second change request recursive search features were added. These allowed the user to choose directories and search them recursively.

This change request will add advanced output features to the search functionality. The search window will now display the search results in the same format as the rest of muCommander. This is a more attractive GUI that includes icons, the size of the file and other information. It will also allow the user to select a file and display it in the main muCommander window. However, it was decided that only a limited feature set of muCommander would be included. So the user will be able to sort the files by name, size and others and select a file and go to it in the main muCommander program. The user will not have access to features such as opening the file directly or renaming files. The number of files and directories will also be displayed.

A.3.2 Concept Location

This change request is to combine 2 parts of muCommander; the search window output area and the table display that is used in the main window of muCommander. To accomplish this, 2 concepts needed to be located; the search window and the table file display. For one concept, no concept location was necessary; the advanced output features are to be added to the search window, which shares it concept location with the last change, the `SearchDialog` code file.

To find the other concept, the file display in the main muCommander window, a dependency search was done starting in the Launcher code file, which has the program's main method. The programmer marked Launcher as Propagating in JRipples, which in turn marked 44 code files as Next. The code file `FocusDialog` was visited, but was marked as Unchanged because it was described as a modal dialog. Since the main window of an application cannot be modal, no further investigation was necessary. Returning to the set of Next code files the next promising code file was `WindowManager`. This code file contained a variable of type `MainFrame`, which because of its name sounded very promising. The programmer marked the `WindowManger` code file as Propagating in JRipples, which marked an additional 35 code files as Next. The variable type `MainFrame` was one of these and it was visited.

`MainFrame` contains 2 variables of type `FolderPanel` and 2 of type `FileTable`; both of these code files sounded promising, because of their names. `MainFrame` was marked as `Propagating`; this caused JRipples to mark 247 more code files as Next. The code file `FolderPanel` from the `MainFrame` visit, was of particular interest and was visited first. It has a `boolean` variable `treeVisible`, which was changed to true. This caused the tree view to be visible when the program was run, which confirmed that the concept location had been found.

Table A.26 contains the totals for each type of code file visited and Table A.27 summarizes the code files visited during concept location. Figure A.17 is a UML of the dependency search path.

Table A.26 Change 3 Concept Location Summary

Title	Code Files			Comments
	Visited	Propagating	Unchanged	
Advanced Output	6	3	1	No CL was done for one concept

Table A.27 Change 3 Concept Location Code Files Visited

#	Code File	Tool used	Located?	Comments
1	`Launcher`	JRipples → Propagating	Propagating	This is the main start location for the program
2	`FocusDialog`	JRipples → Unchanged	Unchanged	
3	`WindowManager`	JRipples → Propagating	Propagating	This singleton class creates all the MainFrame objects
4	`MainFrame`	JRipples → Propagating	Propagating	This class creates the main muCommander window
5	`FolderPanel`	JRipples → Located	Located	The concept is located here

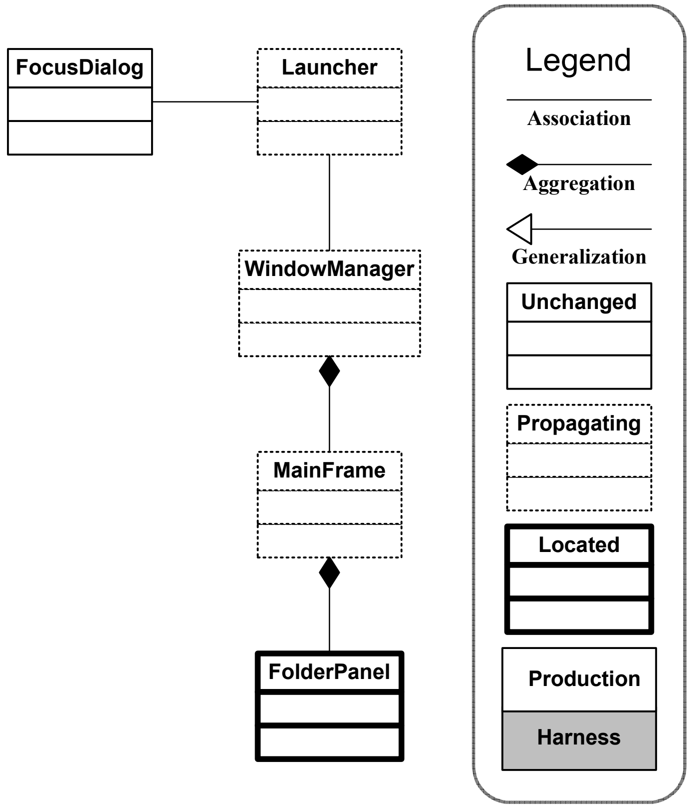


Figure A.17 Change 3 Concept location UML

A.3.3 Impact Analysis

During concept location the programmer located 2 concepts. One the search window was located in the `SearchDialog` code file. The second the table that displays files was located in the `FolderPanel` code file.

The first step of impact analysis by the programmer was to mark the code file `SearchDialog` as Impacted in JRipples. JRipples marked 19 code files as Next. Then `SearchThread` was visited and marked as Impacted; it performs the search and will have to change how it returns results. No additional code files were marked as Next as a result. After that, 4 test classes were marked as Impacted, `SearchDialogTestSetUp`, `SearchDialogTest`, `SearchThreadTest` and `BasicSearchFuncTest`; this caused JRipples to add 10 additional code files to the Next set. Three suppliers and clients of `SearchDialog` were visited and marked as

Impacted: InputPanel, FlashLabel and DirectoryPanel, along with their test code files. JRipples added 3 code files to the Next set; for a total of 24 code files marked as Next. Included in this set was FolderPanel, which holds the second concept location.

FolderPanel was visited and marked as Impacted; 112 code files were now included in the Next set. FileTable was visited because an object of its type is created in FolderPanel and it was seen in MainFrame with FolderPanel during concept location. The Javadoc description states that it “displays a folder’s contents”; the programmer it was marked as Impacted. Now 188 code files were marked as Next in JRipples. The code files that were suspected to contain suppliers of FileTable because their names started with FileTable were visited. FileTableModel, FileTableHeaderRenderer, FileTableHeader, FileTableConfiguration, FileTableColumnModel and FileTableCellRenderer were all marked as Impacted. JRipples still had 188 code files marked as Next. These code files were visited; MainFrame was marked as Impacted because it had a method that created a FileTableConfiguration class need to create a FileTable.

At this point 328 code files were in the Next set. The programmer marked all of these code files as Unchanged; for some of the code files an inspection of just reading the name was sufficient, such as CalculateCheckSumDialog which could easily be confidently marked Unchanged. However, others such as FolderTreePanel, which clearly could have been impacted, were visited more closely along with code fides whose responsibilities could not be determined, such as DataList. These code files have been left of the UML of impact analysis in Figure A.18 because of space constraints.

The estimated impact set contained 21 code files at the end of impact analysis. These code files are listed in Table A.29; the 328 code files marked Unchanged have been left off. Table A.28 summarizes the number of code files visited during impact analysis and their final marks.

Table A.28 Change 3 Impact Analysis Summary

Title	Code files					Comments
	Visited	Impacted	Propagating	Unchanged	Not Visited	
Advanced Output	349	21	0	328		Advanced Output

Table A.29 Change 3 Impact Analysis Code Files Visited

#	Code File	Tool used	Impacted?	Comments
1	`SearchDialog`	JRipples → Impacted	Impacted	This code file contains one concept location
2	`SearchThread`	JRipples → Impacted	Impacted	This code file is responsible for actually searching the file system.
3	`SearchDialogTestSetUp`	JRipples → Impacted	Impacted	
4	`SearchDialogTest`	JRipples → Impacted	Impacted	
5	`SearchThreadTest`	JRipples → Impacted	Impacted	
6	`BasicSearchFuncTest`	JRipples → Impacted	Impacted	Creates SearchDialog
7	`InputPanel`	JRipples → Impacted	Impacted	Supplier to SearchDialog
8	`FlashLabel`	JRipples → Impacted	Impacted	Supplier to SearchDialog
9	`DirectoryPanel`	JRipples → Impacted	Impacted	Supplier to SearchDialog

10	`InputPanelTest`	JRipples → Impacted	Impacted	
11	`FlashLabelTest`	JRipples → Impacted	Impacted	
12	`DirectoryPanelTest`	JRipples → Impacted	Impacted	
13	`FolderPanel`	JRipples → Impacted	Impacted	This code file contains the second concept location
14	`FileTable`	JRipples → Impacted	Impacted	This code file is the main supplier to FolderPanel
15	`FileTableModel`	JRipples → Impacted	Impacted	Supplier to FileTable
16	`FileTableHeaderRenderer`	JRipples → Impacted	Impacted	Supplier to FileTable
17	`FileTableHeader`	JRipples → Impacted	Impacted	Supplier to FileTable
18	`FileTableConfiguration`	JRipples → Impacted	Impacted	Supplier to FileTable
19	`FileTableColumnModel`	JRipples → Impacted	Impacted	Supplier to FileTable
20	`FileTableCellRenderer`	JRipples → Impacted	Impacted	Supplier to FileTable
21	`MainFrame`	JRipples → Impacted	Impacted	Creates FileTableConfiguration

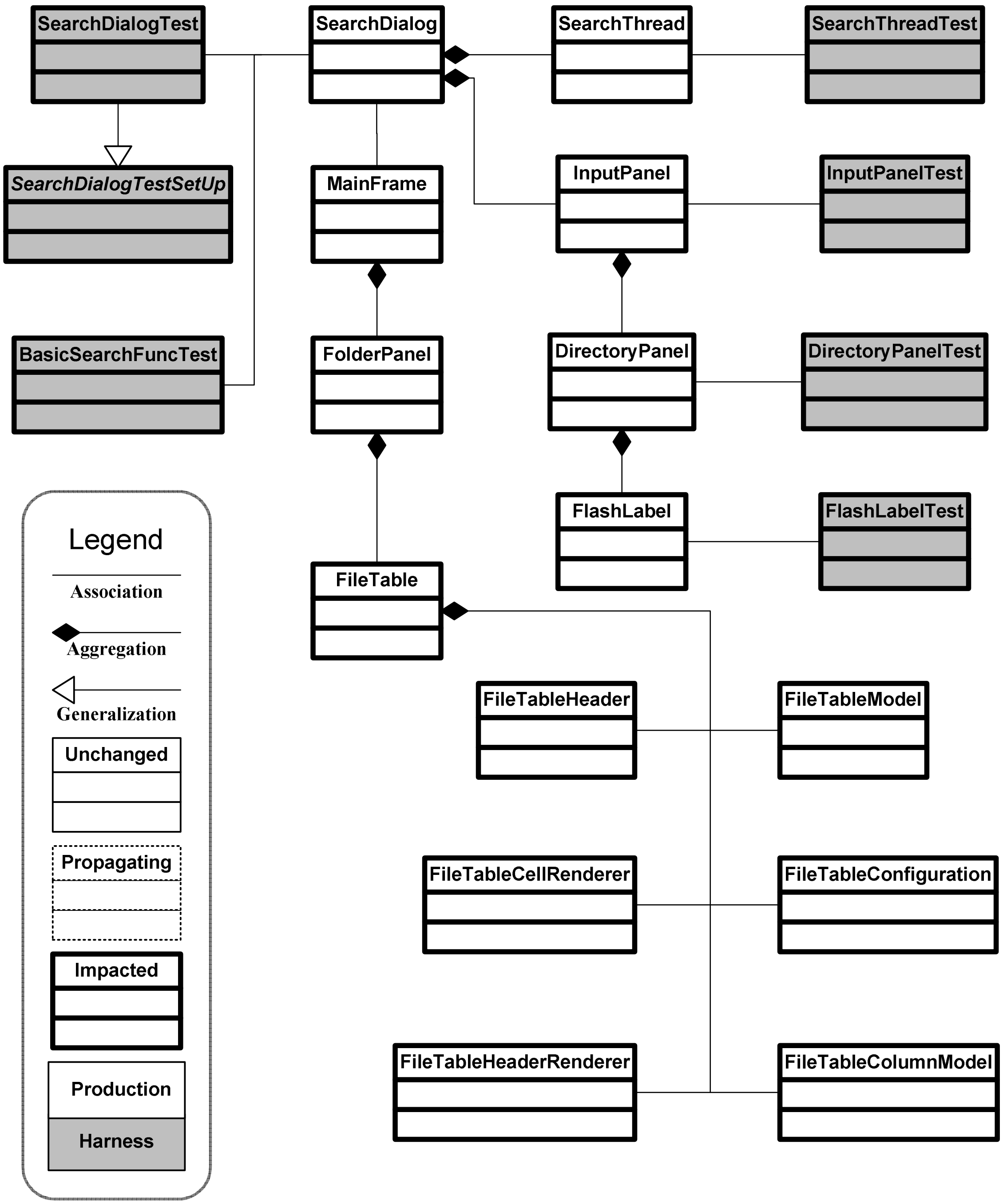


Figure A.18 Change 3 Impact Analysis UML

A.3.4 Prefactoring

`FileTabe` and `FolderPanel` classes can only be contained in an object of type `MainFrame` the programmer did this prefactoring to allow the file table display to be contained in other types of objects. To prepare for this change the classes

AbstractFileTable and AbstractFolderPanel were extracted from FileTable and FolderPanel respectively. These were very large class extractions the original code files were 2069 and 1478 LOC respectively. Because of the size of the class extractions the task was not broken up into smaller tasks, such as extracting methods in the current class then moving them to the new abstract class. While that strategy may be a safe strategy, because of the size of the class extraction, it was perceived to be overly burdensome.

The strategy used was to move universal functionality to the abstract class and leave the rest. For example, the FolderPanel class has a field, currentFolder, of type AbstractFile, which contains the parent directory currently displayed in muCommander. Since search results do not have a common parent directory, this attribute was left in FolderPanel. However, since all types of displays can have more files to display then their size allows, the attribute JScrollPane scrollPane was moved to the abstract class. This will allow all AbstractFolderPanels to have the capability to scroll through the displayed files when necessary.

Additionally, 2 suppliers of FileTable, FileTableHeader and FileTableCellRenderer had attributes of their parent type FileTable this had to be changed to type AbstractFileTable. Table A.30 shows the change propagation set of prefactoring. Table A.31 shows the LOC added and deleted during prefactoring. Figure A.19 is a UML diagram of the code files changed and added during prefactoring.

Table A.30 Change 3 Prefactoring Summary

Title	Code Files					
	Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
Advanced Output	4	4	2	0	0	0

Table A.31 Change 3 Prefactoring Code Files

#	Code File	Task	Lines of Code		
			Added	Deleted	Total
1	`FileTable`	Extracted super class from	103	466	569
2	`FileTableCellRenderer`	Changed method	3	3	6
3	`FileTableHeader`	Changed method	3	3	6
4	`FolderPanel`	Extracted super class from	47	129	176
5	`AbstractFileTable`	Extracted super class	574	0	574
6	`AbstractFolderPanel`	Extracted super class	121	0	121

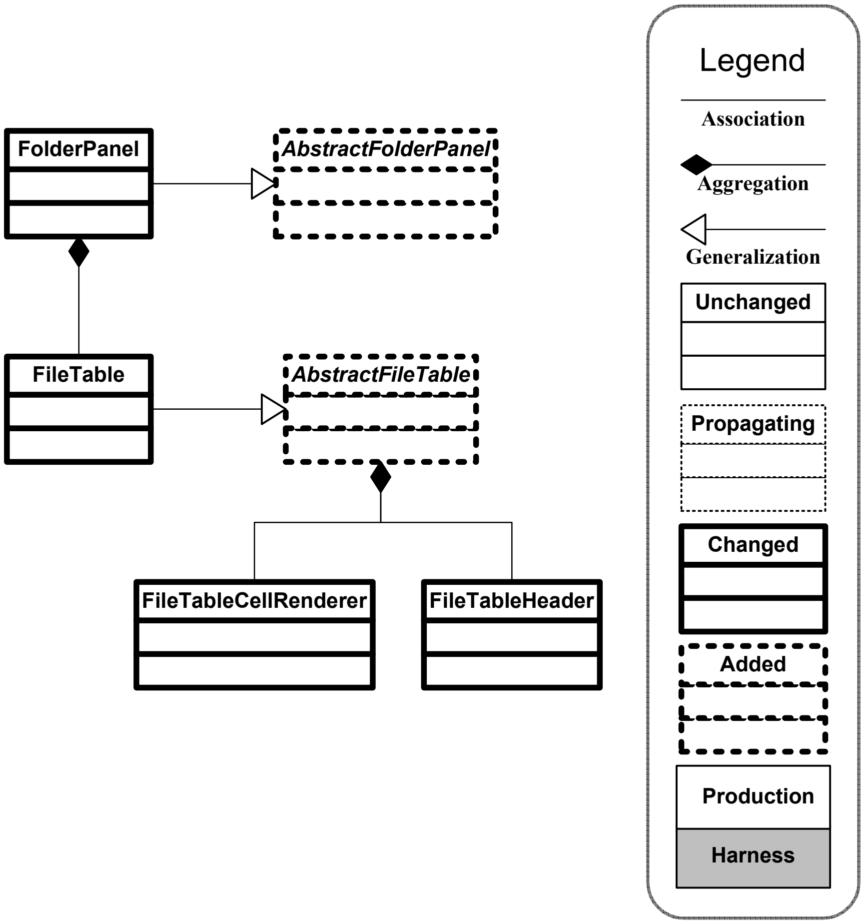


Figure A.19 Change 3 Prefactoring UML

A.3.4.1 AbstractFolderPanel abstract class

This class was extracted from `FolderPanel`. It extends `JPanel` and contains an `AbsractFileTable`. Its other fields are a `JScrollPane`, a `MainFrame` and 5 fields of type `Color` to set the border and background colors. This also represents its responsibilities.

A.3.4.2 FolderPanel code file

`AbstractFolderPanel` was extracted from this code file. It was left with the responsibility for the current folder displayed in its `FileTable`. It also has a tree view display and a nested class to change the current folder. This code file was still large, 619 LOC.

A.3.4.3 AbstractFileTable abstract class

This class was extracted from `FileTable`. It contains a `FileTableModel`, which holds the table's data and a `FileTableCellRender` that formats each cell of the table. It also has fields to set default column values, the current row, if the table is the active table and double click timing information. This was all deemed to be common to all tables and would facilitate the change.

A.3.4.4 FileTable class

`AbstractFileTable` class was extracted from this class. The remaining responsibilities of this class include, a `MainFrame` class that it belongs to, changing a file's name, a field of type `QuickSearch`, which allows a simple search in a folder and a `HashMap` that contains the table's listeners. This class was still large after the class extraction, 590 LOC.

A.3.4.5 FileTableHeader class

This class needed to have its constructor parameter changed from `FileTable` to `AbstractFileTable` because it was being called from `AbstractFileTable` with a `this` call.

A.3.4.6 FileTableCellRenderer

This class needed its constructor parameter changed from type `FileTable` to `AbstractFileTable` for the same reason as `FileTableHeader`.

A.3.5 Actualization

To actualize the change, 2 new classes were created, `SearchFolderPanel` and `SearchTable`. These classes inherit from the classes extracted during prefactoring `AbstractFolderPanel` and `AbstractFileTable`. Parts of the change propagated through these new classes to their suppliers. Then an object of type `SearchFolderPanel` was created in `SearchDialog` and an object of `SearchTable` in `SearchFolderPanel`.

The overall flow to display the results starts in `SearchThread`, which finds the files that match the search term in the file system. It then calls methods in `SearchDialog` to display the results. There were methods to do this at the start of the change, created in change 2 (section A.2). These methods were modified and added to; then `SearchDialog` sent the results to `SearchFolderPanel`, which sent them to `SearchTable`. `SearchTable` sends the results to the class that manages its data structure, `FileTableModel` and `FileTableCellRenderer` actually displays them to the user.

All of the previous code files were impacted by the change. In addition, 3 more suppliers to `SearchTable` needed to be modified along with 3 test classes and 2 new test classes were added. Table A.32 shows the change propagation set of actualization. Table A.33 shows the LOC added and deleted during actualization by code file. Figure A.20 is a UML diagram of the code files changed and added during actualization.

Table A.32 Change 3 Actualization Summary

Title	Code Files					
	Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
Advanced Output	18	10	4	0	4	0

Table A.33 Change 3 Actualization Code Files

#	Code File	Task	Lines of Code		
			Added	Deleted	Total
1	`SearchThread`	Changed method	10	6	16
2	`SearchDialog`	Added, changed methods	138	25	163
3	`SearchFolderPanel`	Added class	52	0	52
4	`SearchTable`	Added class	67	0	67
5	`FileTableModel`	Added methods	42	0	42
6	`FileTableCellRenderer`	Changed method	23	5	28
7	`FileTableHeader`	Added, changed methods	53	2	55
8	`FileTableHeaderRenderer`	Changed variable type	1	1	2
9	`AbstractFileTable`	Added, deleted, changed methods	6	4	10
10	`SearchFolderPanelTest`	Added test class	55	0	55
11	`SearchTableTest`	Added test class	90	0	90
12	`BasicSearchFuncTest`	Added, changed tests	133	4	137
13	`SearchDialogTest`	Added, deleted, changed tests	65	25	90
14	`SearchThreadTest`	changed tests	19	5	24

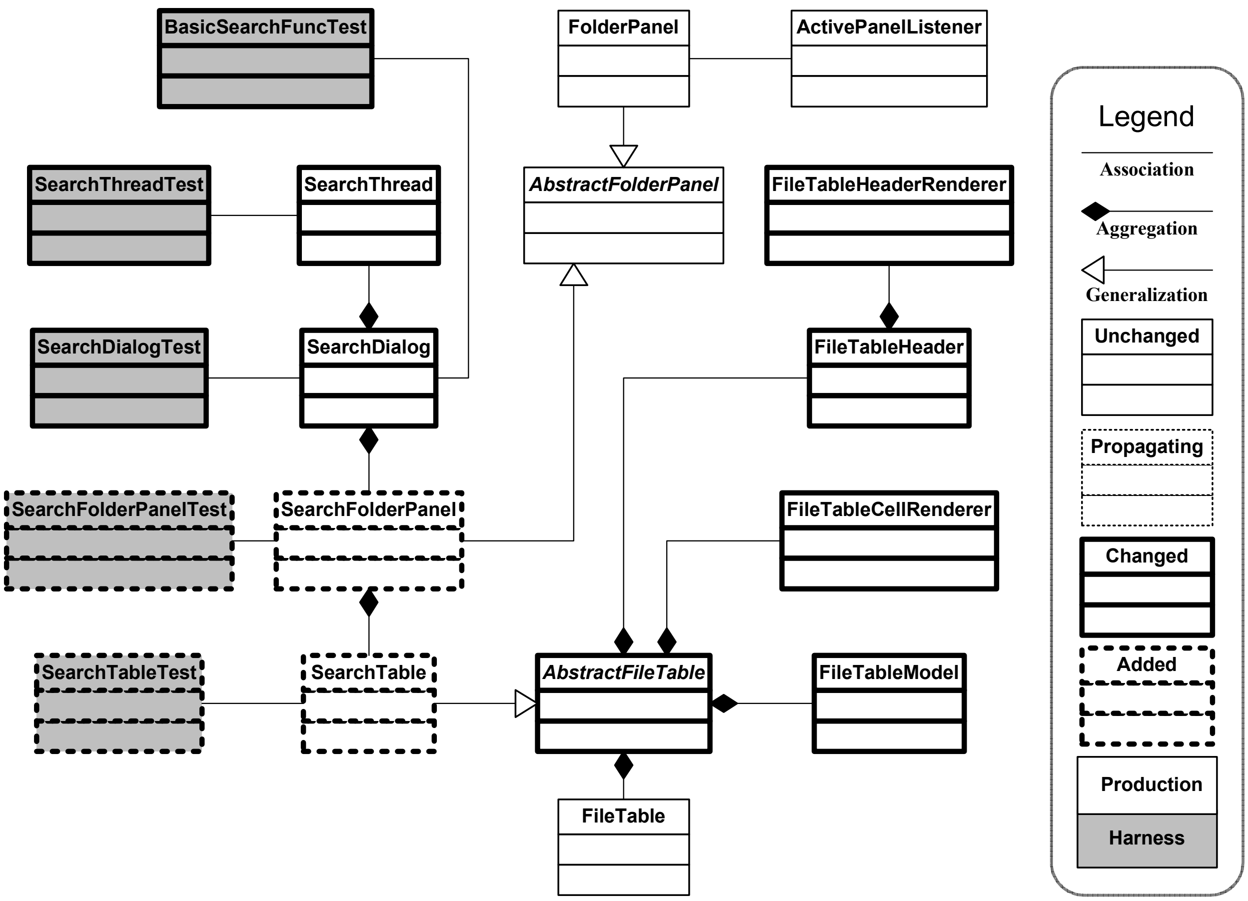


Figure A.20 Change 3 Actualization UML

A.3.5.1 SearchThread class

The changes to this class were all done to its `recursiveSearch()` method. The method was sending error messages to `SearchDialog`, but it was not supported anymore, the new table can only display files, so errors are now sent to the applications log. A second check to make sure the search should continue was added. `SearchDialog` used to ignore a few extra results found before the `SearchThread` would die, but this also would not be supported in the modified methods.

A.3.5.2 SearchDialog class

This class had the largest amount of code change, 138 LOC added and 25 LOC deleted. The method `addSearchResult()` was substantially modified. It previously just sent the results to another method to be displayed in a `JTextArea`, but now sends them to a new field of SearchFolderPanel. So addSearchResult() was changed to initializes a new array field to store the results and resize it as needed. It also increments 2 integer fields to keep a count of directories and files found during the search. Finally, it starts a timer, so that results are displayed in batches.

The timer is activated every 200ms and it calls a method, repaintSearchTable(), to send the current set of results to SearchFolderPanel; it also displays the results totals. To stop the timer, when SearchThread has finished the search, it calls a modified notifyEnd() method. This method, stops the timer, calls repaintSearchTable(), to make sure all the results are displayed and calls switchToSearchState() with a true value.

The method switchToSearchState() was modified. If invoked with its parameter is set to true, it now calls stopSearchThread(). If set to false, it resets the results total fields and reinitializes the array of results. It also clears the results totals that are on display and calls the clearOutput() method in the SearchPanel class.

A method goToSelection() was added that takes an AbstractFile as a parameter. It calls a method in the parent class of SearchDialog, MainFrame, to open the AbstractFile's parent and set the AbstractFile as selected. Then it closes the SearchDialog.

A method was added that was copied from MainFrame, called getFileTableConfiguration(). It creates a configuration class that is required when SearchDialog creates an instance of the new SearchFolderPanel class. The only change made to this method was to remove a boolean parameter, isLeft and replace it with the value of false.

The FocusListener interface was added to SearchDialog along with its 2 methods focusLost() and focusGained(). These methods change the default button to null, if the SearchTable has focus; if the SearchTable loses focus the searchStopButton of SearchDialog is set to default. Finally, the constructor was modified to create an instance of SearchFolderPanel instead of calling createOuputArea().

A.3.5.3 SearchFolderPanel class

This class was created to implement AbstractFolderPanel. It has a clearOutput() method that calls a method from SearchTable called clearSelection() and the method setSearchResults() calls setSearchResults() also in SearchTable . Its constructor calls the super class constructor and creates an instance of SearchTable.

A.3.5.4 SearchTable class

This class was created to implement AbstractFileTable. It has a method setSearchResults() that takes an array of objects of type AbstractFile and sends them to FileTableModel. It also calls the methods setLastRow() and resizeAndRepaint() from its super class. It overrides the method doubleClick() that calls the goToSelection() method in SearchDialog, when the user clicks on a result in the SearchTable.

The method isColumnDisplayable() was overridden, it decides what columns in the table are valid to be displayed based on the directory chosen by the user.

The `keyReleased()` method was overridden to catch the up, down and enter keys. It enables the user to select the next file in the table with the up and down arrow keys or to close the search and open the selected file in `MainFrame` with the enter key.

The constructor calls the super class constructor and a method `sortByNothing()` in the super class. This is done to show the user the table is not sorted by default, they can sort it after a search, if they desire.

A.3.5.5 FileTableModel class

This class contains the data structure for the results displayed in classes that extend `AbstractFileTable`. A method, `setSearchResults()` was added that takes an array of objects of type `AbstractFile`. It takes data from the objects of `AbstractFile` such as theirs names and sizes and creates loads it into a 2 dimensional array and creates 2 more arrays of the same size; one for the sort order of the files and one of the files in the array that are marked.

A.3.5.6 FileTableCellRenderer class

This class creates the Objects that the cells in an `AbstractFileTable` class display. The method `getTableCellRendererComponent()` was modified. If its parent `AbstractFileTable` is an instance of a `SearchTable`, instead of its normal behavior of displaying just the `AbstractFile`'s name, it will display a period plus the path after the directory that was searched in, plus the file name. This gives the user the full path of the file in an easy to read format that is less likely to be cut off. It also sets the cells tooltip to the entire file path and name.

A.3.5.7 FileTableHeader class

This class creates a content menu that the `MainFrame` class listens to. The method `mouseClicked()` was modified to create a context menu that it can listen to, if its parent is a `SearchTable`. The `ActionListener` interface was added to listen for this new menu; its `actionListener()` method changes the `SearchTable` header as requested.

A.3.5.8 FileTableHeaderRenderer class

This class was a client of `FileTable`, to enable it to be a client for all classes that extend `AbstractFileTable`, it was necessary to change a type of a temporary variable and a cast assigned to the variable from type `FileTable` to `AbstractFileTable` in the method `getTableCellRendererComponent()`.

A.3.5.9 AbstractFileTable abstract class

This class had a method added. The responsibility to sort the table is here. All the existing sort methods required a column to be selected. However, results are added in the order they are found, which does not match any of the columns. So a method `sortByNothing()` was added that does not sort by any column.

A.3.5.10 SearchFolderPanelTest class

This class was created to unit test the `SearchFolderPanel` class. It extends `SearchDialogTestSetUp` and has 4 tests.

A.3.5.11 SearchTableTest class

This class was created to unit test the `SearchTable` class. It extends `SearchDialogTestSetUp` and has 8 tests.

A.3.5.12 BasicSearchFuncTest class

This class is a functional test suite. It had 3 tests modified and 8 tests added.

A.3.5.13 SearchDialogTest class

This class is the unit test suite for the `SearchDialog` class. It had 7 tests modified, 6 tests added and 1 deleted.

A.3.5.14 SearchThreadTest class

This class is the unit test suite for the `SearchThread` class. It had 3 tests modified.

A.3.6 Postfactoring

After the actualization phase, many code smells were present. This was addressed during postfactoring. During actualization the programmer added too much responsibility to the `SearchDialog` class. It had 1 class extracted, `ButtonPanel` and responsibility moved to 3 other classes, `SearchThread`, `SearchFolderPanel` and `MainFrame`.

The suppliers to `AbstractFileTable` now had 2 sets of responsibilities, 1 set if the inherited class is `FileTable` and 1 set if the inherited class was `SearchDialog`, in hindsight, this should have been addressed during prefactoring. To resolve the situation the programmer extracted a super class, `AbstractFileTableModel` from `TableModel` and also extracted the `SearchModel` class that inherits from it.

In the case of `FileTableCellRenderer` and `FileTableHeader` classes 2 new classes, `SearchTableCellRenderer` and `SearchTableHeader`, were created that inherited from the existing supplier and they just overrode a subset of their super class's methods; see code file descriptions for more information. This actualization phase gave these classes 2 different responsibilities depending on the caller, therefore to make future changes easier this was done to preserve the code. Once all these extra classes were created the `org.severe.ui.dialog.search.panels` package had too many classes, many of which were not panels, so a new package `org.severe.ui.dialog.search.table` was created for them. The package `org.severe.ui.dialog.search.components` was also created for `FlashLabel`.

The new extracted class `AbstractFileTableModel` propagated to 7 classes not in the estimated impact set or changed set that depended on `FileTableModel` as a supplier. Six of these classes required a field or temporary variable type to be changed to `AbstractFileTableModel` and one required a getter call to be cast to a `FileTable`. During impact analysis, it was thought that the type of the getter that these classes use to get the `FileTableModel` could be kept. However, the getter is inherited from `AbstractFileTable`; it was determined that the best solution was to change these classes. By using a generic type future should be easier.

Many of the test classes were creating the same objects of `AbstractFile` or using instances created in the `SearchDialogTest` class. These were all extracted to a new class `TestConstants`.

Table A.34 shows the change propagation set of postfactoring. Table A.35 shows the LOC added and deleted during postfactoring. Figure A.21 is a UML diagram showing all the classes changed and added during postfactoring.

Table A.34 Change 3 Postfactoring Summary

Title	Code Files					
	Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
Advanced Output	31	31	10	0	2	7

Table A.35 Change 3 Postfactoring Code Files

#	Code File	Task	Lines of Code		
			Added	Deleted	Total
1	`SearchDialog`	Extracted class from, moved field from, deleted unused methods	64	250	314
2	`SearchThread`	Moved field	19	6	25
3	`MainFrame`	Changed method	1	1	2
4	`ResultsPanel`	Renamed class	41	24	65
5	`SearchTable`	Moved class	31	17	48
6	`AbstractFileTableModel`	Extracted super class	110	0	110
7	`FileTableModel`	Extracted super class from	15	124	139
8	`SearchTableModel`	Extracted class	144	0	144
9	`FileTableCellRenderer`	Extracted class from	55	49	104
10	`SearchTableCellRenderer`	Extracted class	42	0	42
11	`FileTableHeader`	Extracted class from	46	97	143
12	`SearchTableHeader`	Extracted class	71	0	71
13	`AbstractFileTable`	Changed methods	10	11	21
14	`CompareFoldersAction`	Changed field	3	3	6
15	`InvertSelectionAction`	Changed field	2	2	4
16	`MarkAllAction`	Changed field	2	2	4
17	`MarkExtensionAction`	Changed field	2	2	4
18	`OpenInBothPanelsAction`	Added cast	1	1	2
19	`FileDragSourceListener`	Changed field	2	2	4
20	`StatusBar`	Changed field	2	2	4

21	FileTable	Changed field	6	4	10
22	FlashLabel	Moved class	1	1	2
23	ButtonPanel	Extracted class	57	0	57
24	DirectoryPanel	Changed method	3	3	6
25	InputPanel	Javadoc	0	0	0
26	SearchDialogTest	Changed tests, moved tests from	21	82	103
27	SearchThreadTest	Extracted constants, changed tests	25	31	56
28	ResultsPanelTest	Renamed class	48	27	75
29	SearchTableTest	Moved class	37	37	74
30	SearchTableModelTest	Added test class	241	0	241
31	SearchTableCellRendererTest	Added test class	46	0	46
32	SearchTableHeaderTest	Added test class	56	0	56
33	FlashLabelTest	Moved class	2	2	4
34	ButtonPanelTest	Added test class	58	0	58
35	DirectoryPanelTest	Extracted constants	5	5	10
36	InputPanelTest	Javadoc	0	0	0
37	SearchDialogTestSetUp	Extracted constant, field	3	2	5
38	BasicSearchFuncTest	Changed tests	48	59	107
39	TestConstants	Extracted class	18	0	18

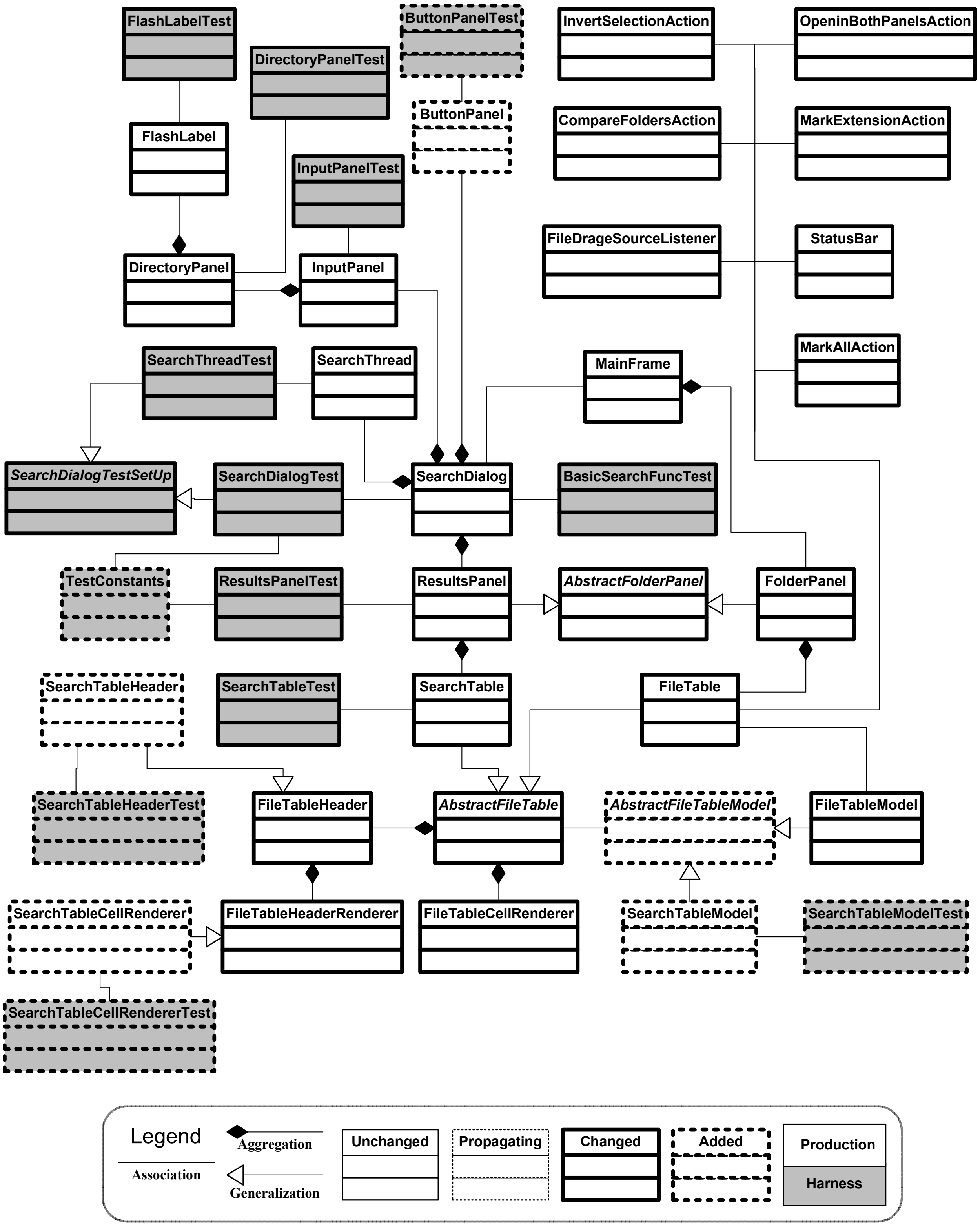


Figure A.21 Change 3 Postfactoring UML

A.3.6.1 SearchDialog class

The fields `searchStopButton`, `cancelButton` and `resultsTotalLabel`, which are all in the south portion of `SearchDialog` and initialized in the method `createButtonArea()` were extracted to a new class `ButtonPanel`. Appropriate parts of the `actionPerformered()` method was also extracted to `ButtonPanel`

The field of the array of objects of `AbstractFile` that holds the results from the search, the integers that hold the results totals and the timer were all moved to the `SearchFolderPanel` class. The `FocusListener` responsibility was also moved and the `FocusListener` interface was removed. After the remaining responsibilities were extracted from the `actionPerformed()` method, it was removed along with the `ActionListener` interface.

The field `keepSearching`, was moved to the `SearchThread` class. The man in the middle that existed, `repaintSearchTable()`, which now just called a method in `SearchFolderPanel` was removed, it was replaced with a call directly from `SearchThread` to `SearchFolderPanel`.

A method that was copied from `MainFrame`, `getFileTableConfiguration()` was removed and replaced with a call to the method in `MainFrame`.

A.3.6.2 SearchThread class

The field `keepSearching` was moved here from `SearchDialog`. A method `stopSearching()` that sets it to false, to tell `SearchThread` that a user has canceled a search was added. Calls to `SearchDialog` and to `SearchFolderPanel` that replaced a man in the middle in `SearchDialog` were added.

A.3.6.3 MainFrame class

This class is the parent of `SearchDialog`. The `MainFrame` class had a method copied to `SearchDialog`, but not substantially changed during actualization. It was responsible for creating a `FileTableConfiguration` class. This responsibility was transferred back to `MainFrame`, which required the visibility of the method to be reduced to public.

A.3.6.4 SearchFolderPanel and ResultsPanel class

`SearchFolderPanel` was renamed `ResultsPanel`, which better describes what it is, namely a `JPanel` that contains the search results; it does not contain a folder and does not search.

The timer field from `SearchDialog` was moved here but was later removed altogether after the extraction of `AbstractFileTableModel` and `SearchModel` rendered it unnecessary.

The responsibility to change the default button focus was moved from `SearchDialog` here. The `FocusListener` interface was already implemented by the base class `AbstractFolderPanel`, so the methods already existed.

The integer fields that hold the results totals were moved here from `SearchDialog`. The method `clearOutput()` was modified to reset these along with clearing the results from the `SearchTable`; it was then renamed `clearResults()`.

A `notifyEnd()` method was added that calls the `update()` method in `SearchTable`. It also sets the final results totals in the `resultsTotalLabel`, by calling `setResultsTotal()` in `ButtonPanel`.

The method setSearchResults() was modified to take a single AbstractFile as a parameter. It sends it to the SearchTable by means of the addSearchResult() method. This method also adds to the results totals and calls the setResultsTotal() method in ButtonPanel.

A.3.6.5 SearchTable class

This class added an integer field named row that keeps track of the maximum row. The method addSearchResult() was modified to accept a single AbstractFile; it calls addSearchResult() with a single AbstractFile as a parameter. It then calls the repaintRow() method in its parent class, AbstractFileTable with the field row as a parameter; it then increments row.

The update() method was added. It is just a delegate method to call the inherited protected method resizeAndRepaint() from JTable. The class ResultsPanel needs to call this method at the end of each search so a new method with a visibility of public was needed.

The method clearSelection() was overridden. It now calls its super method, reset the row field, calls the TableModel clear() method and resizeAndRepaint().

Finally, this class was moved to the new org.severe.ui.dialog.search.table package.

A.3.6.6 AbstractFileTableModel abstract class

This super class was extracted from FileTableModel. It contains the data structure that an AbstractFileTable can display. The FileTableModel class allowed search results to be displayed, but was implemented with arrays. This works well for displaying the contents of a directory, however, it was easy to overload this class with data. A timer was added during actualization to only add an array of objects of `AbstractFile` to this class every 200ms. This workaround was not ideal; this class was extracted, so that a new class could be implemented using collections that automatically expand instead of arrays.

The fields long `markedTotalSize`, `int nbRowsMarked`, `SortInfo sortInfo` and `int sizeFormat` were extracted to this new class. The methods associated with these responsibilities were also extracted. These included, `setSizeFormat()`, `setSortInfo()`, `getFirstMarkableRow()`, `isRowMarked()`, `setRangeMarked()`. The methods that referred to the file data to be displayed in the `AbstractFileTable` were made into abstract methods that the classes implementing this class could override. These included `getCachedFile()`, `getFiles()`, `getFileRow()`, `getFileAt()`.

The method `getFileComparator()` changed visibility from default to protected, so that the implementation classes could call it. The `sortRows()` and `fillCellCache()` methods were also made abstract, because they also depend on the data storage implementation.

A.3.6.7 FileTableModel class

This class had `AbstractFileTableModel` extracted from it. No new methods were added. See `AbstractFileTableModel` (section A.3.6.6) for a description of the methods removed. If a method was made abstract in `AbstractFileTableModel`, its implementation was not changed in this class. Additionally, the 2 overloaded methods `addSearchResults()` were moved to `SearchTableModel`.

A.3.6.8 SearchTableModel class

This class implements the `AbstractFileTableModel` class. It is similar to the `FileTableModel` class but instead of storing the `AbstractFile` objects in arrays that need to be manually resized as results are added; it uses Java standard collections that automatically resize. Specifically, it stores all `AbstractFile` objects passed to it in an `ArrayList`. It then caches the file's data, such as name, date and size as objects in a `HashMap`, with the `AbstractFile` object as the key. When called upon to sort the `AbstractFile` objects by a criteria, it sorts the `ArrayList` using the Java `Collections.sort()` method. It can then look up the sorted file's data from the `HashMap` as needed. This method made much easier to read code and ran very quickly and smoothly. The capability to mark multiple files was not supported, because it is not supported by a `SearchTable`.

The overloaded method `addSearchResults()` that accepted an array of `AbstactFile` objects was deleted. The `addSearchResults()` method that accepted a single `AbstractFile` object was renamed `addSearchResult()` to reflect its current responsibility.

The responsibility to create a `String` with a partial or full path and the name of the file was extracted from `FileTableCellRenderer` to the method `fillCellCachAtRow()` method. This method creates objects for `FileTableCellRenderer` to display. The responsibility to create this `String` did not fit with the responsibility of `FileTableCellRenderer`; however, `SearchTableModel` was already doing other simple data processing tasks, so moving it here made sense.

A.3.6.9 FileTableCellRenderer class

This class had 1 method that was very large, `getTableCellRenderComponent()`. This method formats an `AbstractFileTable` cell for display. It does all the tasks such as getting the `String` to display, setting the colors, fonts and the tool tip. During actualization if statements were added to change this behavior if its supplier class was a `SearchTable`. This just expanded the method and made the code smells even more pungent. The method had 6 methods, `getQuickSearch()`, `setMatches()`, `setLabel()`, `truncateText()`, `setbackGroundColor()`, and `setOutLine()` extracted from it. This not only made the code easier to read, but was done to make it easier for a class to override specific parts of the original method, without duplicating code.

The class field `tableModel` also changed type from `FileTable` to `AbstractFileTable`. The if statements that were added during actualization to create different functionality for the `SearchTable` were extracted from `setLabel()`, `setBackgroundColor()` and `getQuickSearch()`.

A.3.6.10 SearchTableCellRenderer

This class extends `FileTableCellRenderer`; it overrides the methods `setLabel()`, `setBackgroundColor()` and `getQuickSearch()`. The `setLabel()` overridden method calls the super, but sets the tool tip to the entire `AbstractFile` path and name displayed in the row. The `setBackgroundColor()` method does not call the super method, but rather removes functionality to shade the background color which is unsupported in a `SearchTable`. The `getQuickSearch()` method just returns `null`, because it too is unsupported in a `SearchTable()`.

A.3.6.11 FileTableHeader class

This class, like `FileTableCellRenderer` had 2 separate paths, 1 if it was a supplier to a `FileTable` and 1 if it was a supplier to a `SearchTable`. This also could easily be solved through inheritance. The class `SearchTableHeader` was extracted from it. This changed the class back into its state before the change started, except that its field table is now an `AbstractFileTable` instead of a `FileTable`.

To do this an if block was extracted from the `mouseClicked()` method and the `ActionListener` interface, its method `actionPerformed()` fields `checkBoxList` and `checkboxMenuItemExt` were moved to the `SearchTableHeader` class.

A.3.6.12 SearchTableHeader class

This class was extracted from the `FileTableHeader` class. It contains a method `mouseClicked()` that overrides the method in `FileTableHeader`. It creates a context menu that it listens to. The class also implements an `ActionListener` interface and the `actionPerformed()` method listens to the context menu created by the `mouseClicked()` method.

A.3.6.13 AbstractFileTable abstract class

This class had its `FileTableModel` field changed to an `AbstractFileTableModel`. The return type and parameter type for the getter and setter for this field also changed, which propagated to 7 other classes.

The calls to `setCellRenderer()` and `seTableHeader()` were removed from this class, so the implementing class could set their own. The constructor parameters were also changed. An `AbstractFileTableModel` was added, so that the implementing classes could set their own.

A.3.6.14 Classes impacted by the change of AbstractFileTable's fileTable field

These 7 code files were not part of the estimated impact set. `CompareFoldersAction`, `InvertSelectionAction`, `MarkAllAction`, `MarkExtensionAction`, `OpenInBothPanelsAction`, `FileDragSourceListener` and `StatusBar` were all affected by the type change of the field `tableModel` in the `AbstractFileTable` class. The class `OpenInBothPanelsAction`, required its call to the getter for this field to be cast to the type `FileTableModel`. The other classes all required their `FileTableModel` fields to be changed to the `AbstractFileTableModel` type.

A.3.6.15 FileTable class

This class now calls `setTableHeader()` and `setCellRenderer()` in its constructor to so that it `FileTableHeader` and `FileTableCellRenderer` supply those responsibilities. Likewise it added a `FileTableModel` to the super constructor call.

A.3.6.16 FlashLabel class

This class was moved to a new package `org.severe.ui.dialog.search.components`.

A.3.6.17 ButtonPanel class

This class was extracted from `SearchDialog`. It contains the south panel of `SearchDialog`. This includes a `JLabel` that displays the total results found during a search. It contains the objects of `JButton` to start, stop and cancel searches. It implements the `ActionListener` interface and listens to the 2 buttons. It also has a method that takes 2 integers as parameters and sets the text of the `JLabel` with these.

A.3.6.18 DirectoryPanel class

The method `actionPerformed()` had a temporary variable assignment changed to a call to the static `File.separator()` method. It was making a system call to determine the file separator path. The temporary variable was then inlined.

A.3.6.19 InputPanel class

Javadoc comments were clarified.

A.3.6.20 SearchDialogTest class

This class is the unit test suite for the `SearchDialog` class. It had 5 tests modified and 1 deleted. Three tests were moved to `ResultsPanelTest` 3 to `ButtonPanelTest`.

A.3.6.21 SearchThreadTest class

This class is the unit test suite for the `SearchThread` class. All 4 of its tests were modified. The objects of `AbstractFile` it used for testing were moved to `TestConstants`.

A.3.6.22 ResultsPanelTest class

This class is the unit test suite for the `ResutlsPanel` class. It had 4 test modified and 3 moved from `SearchDialogTest`.

A.3.6.23 SearchTableTest class

This class is the unit test suite for the `SearchTable` class. It was moved to the `org.severe.ui.dialog.table.tests` package. It had 5 tests modified, 1 added and 1 deleted.

A.3.6.24 SearchTableModelTest class

This class is the unit test suite for the `SearchTableModel` class. It was added and has 19 tests.

A.3.6.25 SearchTableHeaderTest class

This class is the unit test suite for the `SearchTableHeader` class. It was added and has 3 tests.

A.3.6.26 FlashLabelTest class

This class is the unit test suite for the `FlashLabel` class. It was moved to the `org.severe.ui.dialog.components` package.

A.3.6.27 ButtonPanelTest class

This class is the unit test suite for the `ButtonPanel` class. It was added and has 4 tests.

A.3.6.28 DirectoryPanelTest class

This class is the unit test suite for the `DirectoryPanel` class. It had 3 tests modified the `AbstractFile` constants they referred to were moved to `TestConstants`.

A.3.6.29 InputPanelTest class

This class had a Javadoc update.

A.3.6.30 SearchDialogTestSetUp abstract class

This class creates an instance of `SearchDialog` for testing by classes that extend it. The path to the test files defined as a `String` constant was moved to the `TestConstants` class. It also added a field of type `SearchTableModel` that can be used in tests.

A.3.6.31 BasicSearchFuncTest class

This is the functional test suite for the search functionality. It had 9 test tests modified. The objects of `AbstractFile` it uses for testing were moved to `TestConstants`. It added a new field of type `SearchTableModel` for use in tests.

A.3.6.32 TestConstants final class

This class was created to organize fields that are commonly referenced in tests. This includes 5 objects of `AbstractFile` and the test directory path `String`.

A.3.7 Verification

After prefactoring and postfactoring all the regression tests passed. No new regression tests were added for the abstract classes extracted from `FolderPanel`, `FileTable` and `FileTableModel`. The classes in the `org.severe.ui.dialog` packages now each have their own test class. All tests are in their own package, which has the same name as the package containing the class being tested plus *tests*. There is 1 functional test class, `BasicSearchFuncTest`. During verification 2 bugs were found, both in the new classes created during postfactoring. Coverage for each production code file is available in Table A.36.

The first bug was in `SearchTableModel`; when it was getting the path parent of the search result instead of the path search result in the `fillCellCacheAtRow()` method. The second bug was in `SearchTable`, in the `addSearchResultMethod()`. It needs to call `resizeAndRepaint()`, an inherited method after adding the first result, to allow the table to resize the columns to the Objects in them.

Table A.36 Change 3 Statement Verification

#	Code File	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	SearchDialog	43	42	97.7	0	0
2	SearchThread	27	25	92.6	0	0
3	SearchTableCellRenderer	10	10	100.0	0	0
4	SearchTableHeader	38	38	100.0	0	0
5	SearchTableModel	65	65	100.0	0	1
6	SearchTable	34	33	97.1	0	1
7	ButtonPanel	23	23	100.0	0	0
8	DirectoryPanel	51	42	82.4	0	0
9	InputPanel	29	29	100.0	0	0
10	ResultsPanel	26	25	96.2	0	0
11	FlashLabel	14	14	100.0	0	0
12	AbstractFileTable	274	195	71.2	0	0
13	AbstractFileTableModel	37	21	56.8	0	0
14	FileTable	331	89	26.9	0	0
15	FileTableCellRenderer	95	84	88.4	0	0
16	FileTableHeader	28	4	14.3	0	0
17	FileTableHeaderRenderer	18	18	100.0	0	0
18	FileTableModel	163	120	73.6	0	0
19	AbstractFolderPanel	60	35	58.3	0	0
20	FolderPanel	328	144	43.9	0	0
21	MainFrame	210	122	58.1	0	0
22	CompareFoldersAction	43	6	14	0	0
23	InvertSelectionAction	16	6	37.5	0	0
24	MarkAllAction	15	8	53.3	0	0
25	MarkExtensionAction	45	6	13.3	0	0
26	OpenInBothPanelsAction	34	9	26.5	0	0
27	FileDragSourceListener	27	3	11.1	0	0
28	StatusBar	207	151	72.9	0	0

A.3.8 Timing Data

Table A.37 contains the timing data for the change.

Table A.37 Change 3 Timing Totals

Phase	Time (hh:mm)
Concept Location	0:33
Impact Analysis	3:23
Prefactoring	2:11
Prefactoring Testing	0:07
Actualization	4:08
Actualization Testing	6:42
Postfactoring	15:49
Postfactoring Testing	5:34

A.3.9 Conclusions

This change could be described as an epic; however, it is difficult to see a logical way to divide it into smaller pieces. Adding the existing table from another part of the program is a do it or do not do it proposition. It would have been more difficult to add 1 column of the table at a time or some other piece of the table.

Alternately, a few parts of the change could have been left out; such as the ability to click on a file, which causes the search window to close and the file to be selected in muCommander's main window. The issue here is that again, it would have been more difficult to add later; but also this only required 2 methods, in already impacted classes. So the size of the change would have been only trivially affected.

Some of the postfactoring could have been skipped and added to the backlog; but the programmer already had the knowledge to do the postfactoring and was right there in the code. To delay the postfactoring to another change would have just made it more difficult. Most of the change was the refactoring of the code; the actualization itself was reasonable. That said the process worked very well; this change shows that SIP can handle a large change. The prefactoring phase made the actualization phase simpler. The postfactoring phase allowed the code to be improved in ways that were not apparent at the start of the change.

The changed set was only 11 compared to 21 code files in the estimated impact set. Of these 10 code files, 8 were impacted during postfactoring, 2 were not impacted. These 2 code files are suppliers to `FileTable` and the programmer assumed that a change this large would propagate to all of `FileTable`'s suppliers. An additional 7 code files were impacted during postfactoring. This is because the programmer changed the return type of a getter method that was extracted from `FileTable` to `AbstractFileTable`.

Table A.38 shows the total number of code files in each set of each phase of the change. Table A.39 is the current state of the product backlog. Figure A.22 to Figure A.25 show screen shots of muCommander before and after the change

Table A.38 Change 3 Code File Summary

#	Change	Number in Code Files						
		Visited Concept Location	Estimated Impact Set	Changed Set	Added during			Total Added
					Pre	Act	Post	
0	Original Baseline	N/A	N/A	N/A	N/A	N/A	N/A	1,070
1	Basic Search	5	3	4	0	4	0	1,074
2	Recursive search	0	3	4	4	0	5	1,083
3	Advanced Output	6	21	11	2	4	10	1,099

Table A.39 Change 3 Current Product Backlog

#	Title	Complete	User Story
1	Basic Search	x	Add a basic search function that allows a user to search in the current directory for all or part of the title of a folder or file, and return a list of the matching files and directories.
2	Recursive Search	x	Add the ability to search inside all directories.
3	Advanced Output	x	Change the output to a table similar to the main muCommander window.
4	Date Search		Allow the user search by a date of file's modification.
5	Case Sensitive Search		Add capability to search by case sensitive search terms.
6	Extension Search		Add the ability to search for files with specific extensions.
7	Properties Search		Add options to search for files based on their properties.
8	Size Search		Add the ability to search for a file by its size.
9	Regular Expression Search		Add capability to search by a regular expression.
10	Lucene Search		Incorporate the Apache Lucene search.

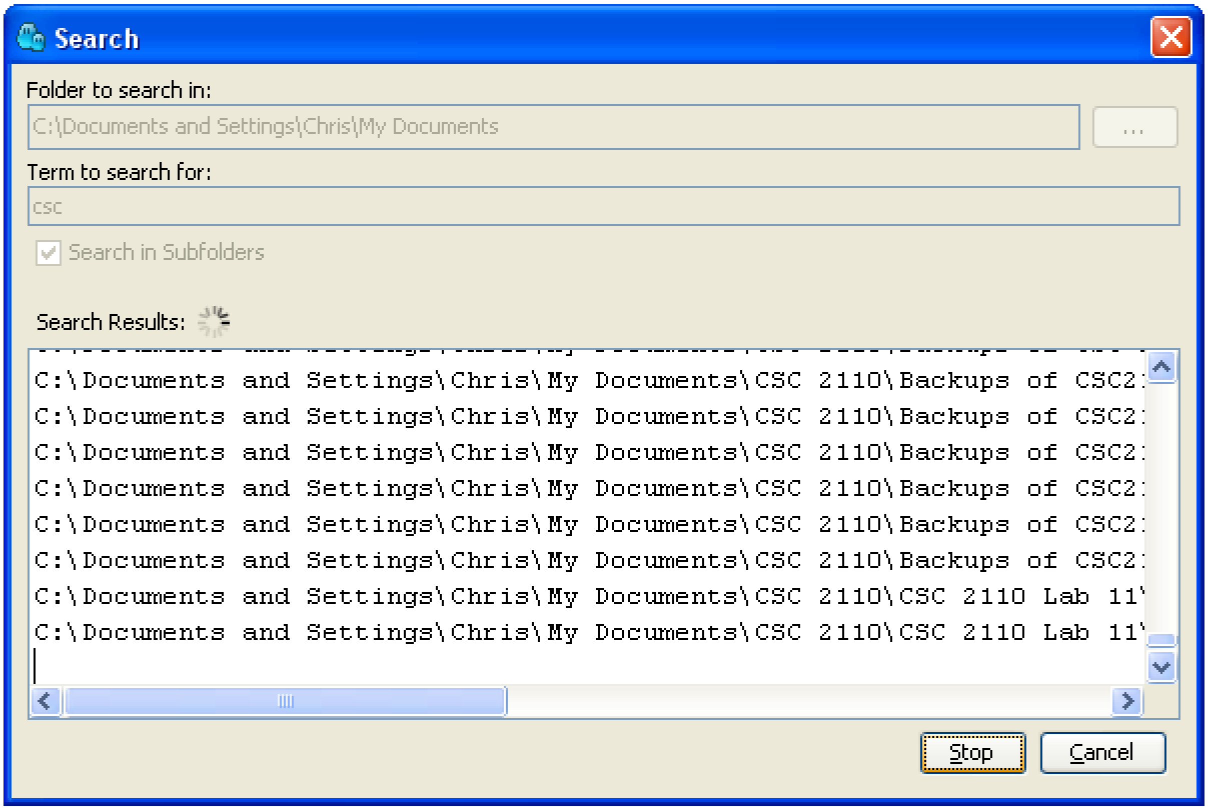


Figure A.22 Search window before Recursive search Change

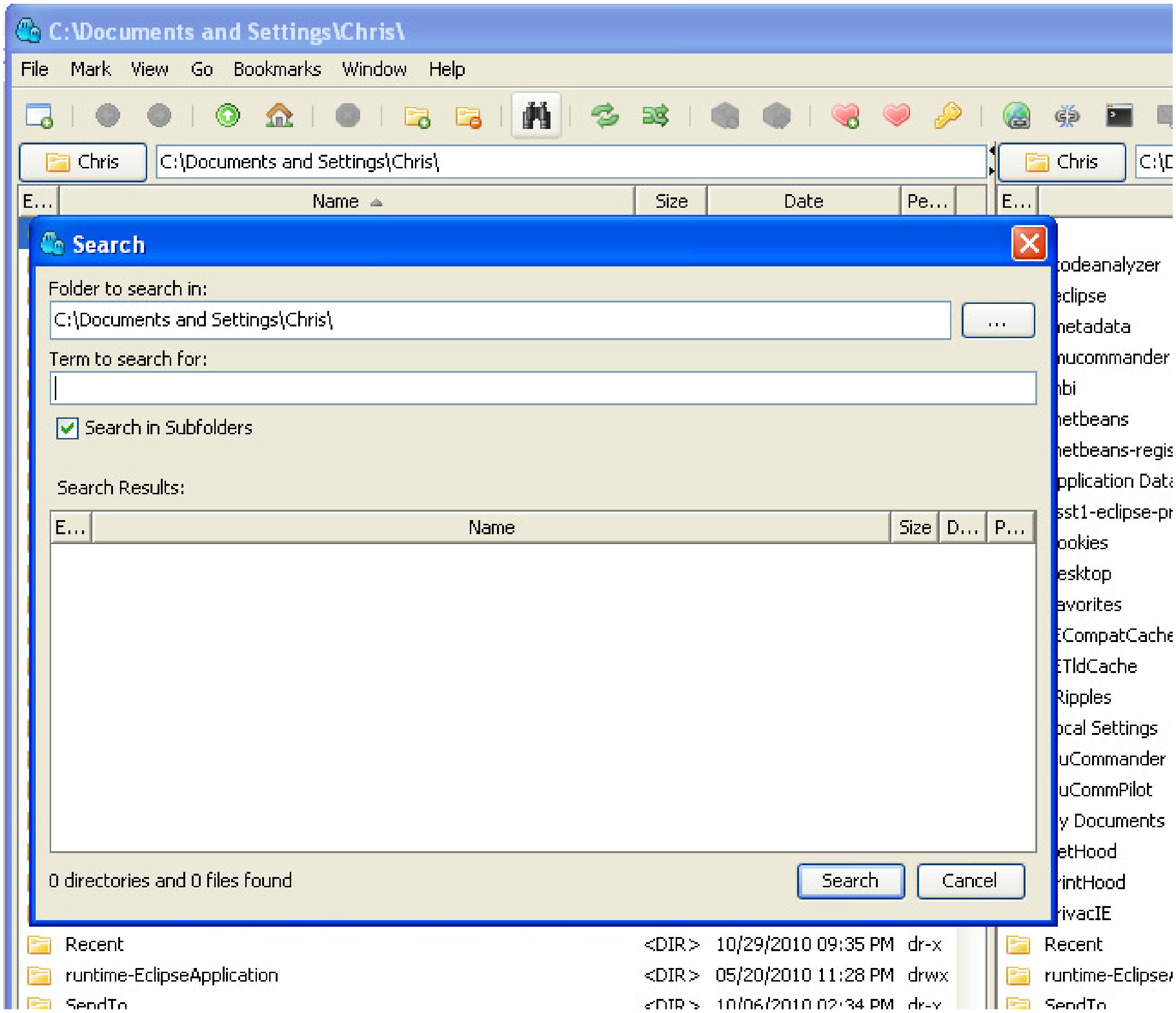


Figure A.23 Search window after Recursive search Change

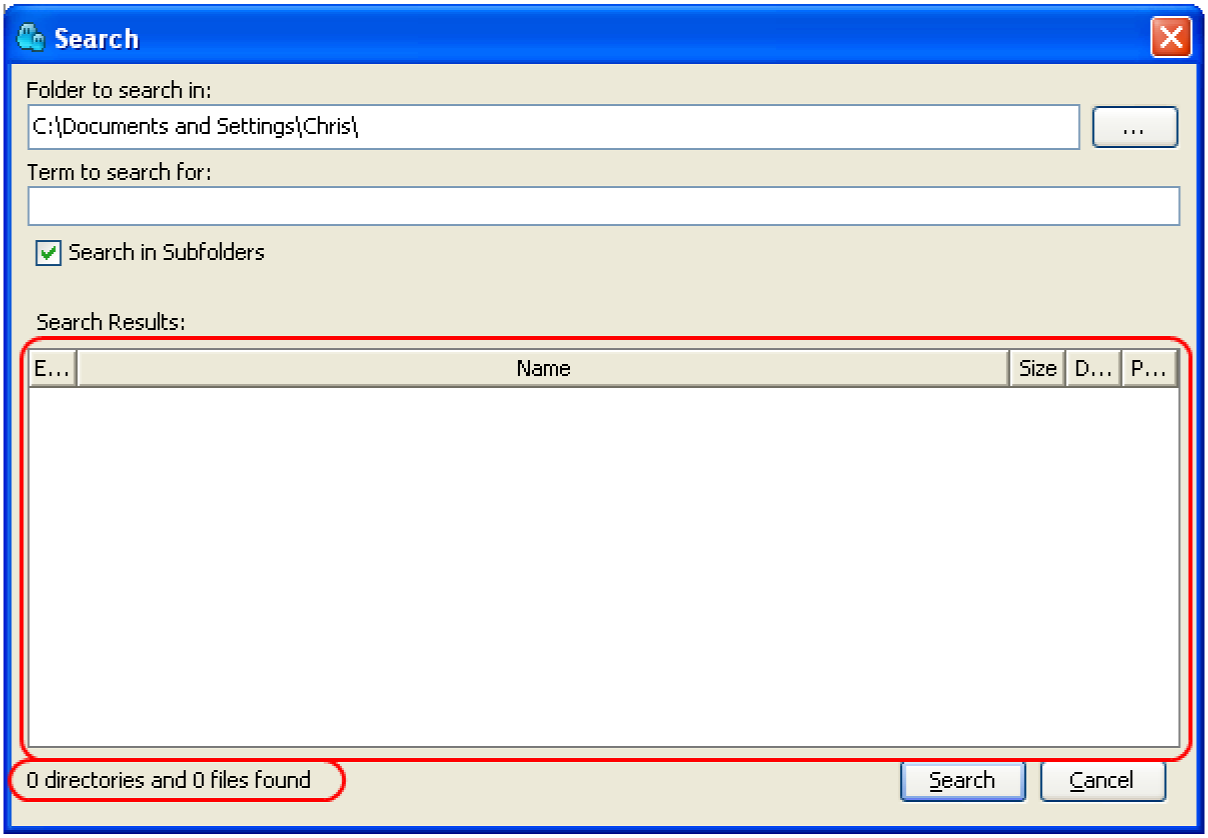


Figure A.24 Search window new input features circled

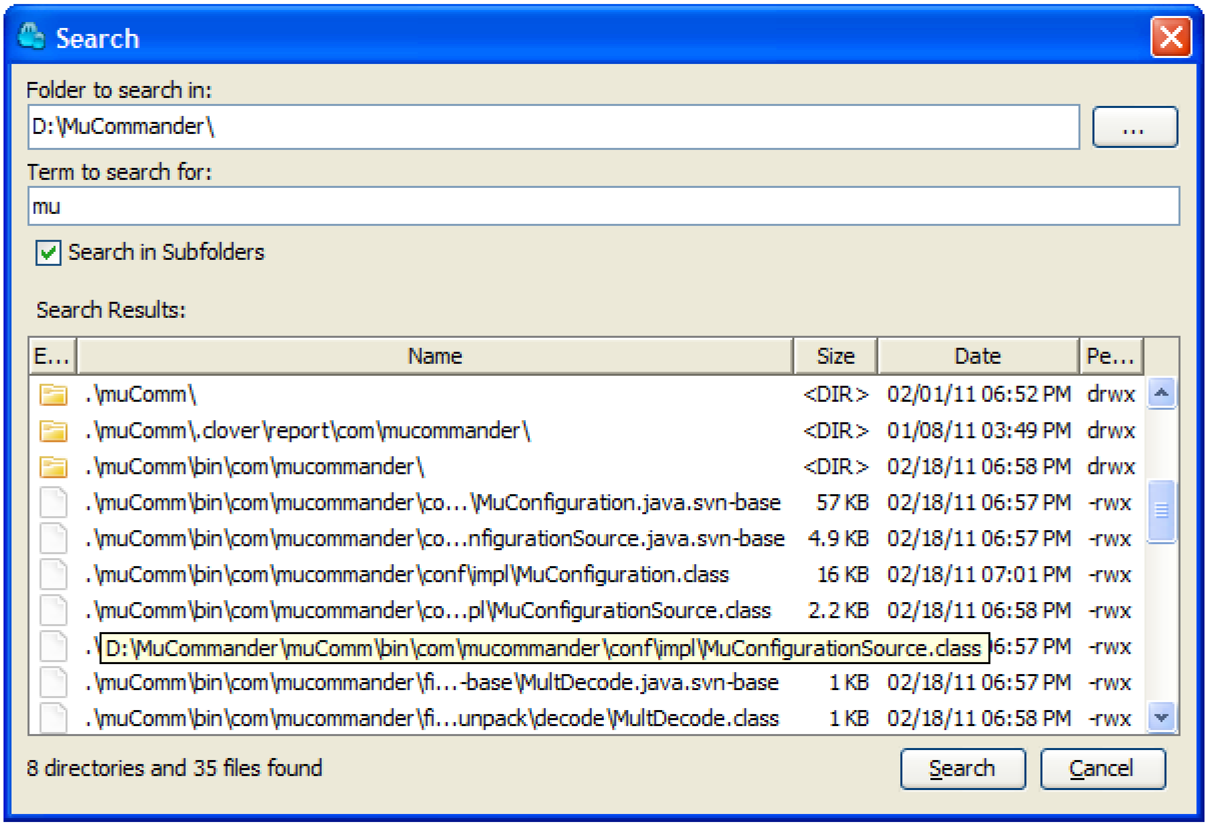


Figure A.25 Search window after search

SIP – Change 4 Date Search

A.4.1Initiation

Allow the user search by a date of file's modification to the Search Feature in muCommander. It is an application which enhances an operating systems file explorer. During the first 3 change requests, search capabilities were added, which helps a user find files in the file system.

This change request will add the capability to the search within a specified date range. The programmer will add 2 boxes to accept a minimum and a maximum date. The search results will include files modified between these 2 dates. Next to these boxes will be 2 icons that will open GUI calendars to select a date. A checkbox will be added to allow the user to choose to use or not use this functionality. The program will also validate the input dates.

A.4.2 Concept Location

No concept location was needed for this change request. Based on experience obtained during previous change requests the programmer knew the search is performed by the `SearchThread` class which was created during change 2.

A.4.3 Impact Analysis

The code file containing the concept location, `SearchThread` was marked as Impacted in JRipples, by the programmer. That caused JRipples to mark 7 code files as Next. From these code files, `SearchDialog` was marked as Impacted; it will need to change, because it creates an object of type `SearchThread`, which will change. This caused JRipples to mark 18 more code files as Next. The programmer then marked `InputPanel` as Impacted; it will hold the new GUI panel to choose a date range to search. JRipples added 4 code files to the set of Next code files for a current total of 27 code files.

The harness code files `BasicSearchFuncTest`, `InputPanelTest`, `SearchDialogTest` and `SearchThreadTest` were all marked as Impacted. There were now 39 code files marked as Next. The programmer visited `ButtonPanel` and marked it as Impacted; it will be responsible for checking to make sure there are no errors in the search criteria, before a search starts. The set of code files marked Next was now 40. `DirectoryPanel` was visited and marked as Impacted; it has the only error currently, now that multiple errors will be possible, there needs to be a central management location for errors. The set of code files marked as next was again 40. The harness code files `DirectoryPanelTest`, `ButtonPanelTest` and `TestConstants` were all marked as Impacted. This did not add any code files to the Next set, so the set of Next code files was now 37.

The programmer visited `AbstractFile`; it has a method, `getDate()`, that can be used to compare an `AbstractFile`'s date to a date range; therefore, it was marked Unchanged. This change request will require a date to be formatted, the same way it is in `AbstractFileTable`. `AbstractFile` was already marked as Next; therefore the programmer visited it. The class calls a static method in `CustomDateFormat`; therefore, `AbstractFileTable` was marked as Propagating. Then `CustomDateFormat` was visited; it has a method, `getDateFormatString()` that returns a date format `String` based on setting in the preference file. It would work, but it included the time, since usually users do not want to be that specific when searching, the programmer decided the day, month and year would be fine grained enough. Thus, `CustomDateFormat` was marked as Impacted; it will need a new method that returns a date format without the time. This left 112 code files in the Next set.

These code files were visited in a similar manner as in change 3. Code files such as `MarkForwardAction` were just marked as Unchanged based on their names. But, other code files, such as `ResultsPanel` that is part of the search dialog, were visited more closely. Ultimately, these code files were marked as Unchanged.

Table A.40 lists the totals of each type of code file visited. Table A.41 lists the code files visited during impact analysis, it leaves off the 112 code files marked

Unchanged at the end of impact analysis for clarity. A UML diagram of impact analysis is in Figure A.26.

Table A.40 Change 4 Impact Analysis Summary

Title	Code Files					Comments
	Visited	Impacted	Propagating	Unchanged	Not Visited	
Date Search	117	14	1	112	0	

Table A.41 Change 4 Impact Analysis Code Files Visited

#	Code File	Tool used	Impacted?	Comments
1	`SearchThread`	JRipples → Impacted	Impacted	Concept Location
2	`SearchDialog`	JRipples → Impacted	Impacted	Creates an instance of SearchThread
3	`InputPanel`	JRipples → Impacted	Impacted	Will hold a GUI date panel
4	`SearchDialogTest`	JRipples → Impacted	Impacted	
5	`SearchThreadTest`	JRipples → Impacted	Impacted	
6	`BasicSearchFuncTest`	JRipples → Impacted	Impacted	
7	`InputPanelTest`	JRipples → Impacted	Impacted	
8	`ButtonPanel`	JRipples → Impacted	Impacted	Needs to check for an error when search button pushed
9	`DirectoryPanel`	JRipples → Impacted	Impacted	Will need to move its error state to a central location
10	`DirectoryPanelTest`	JRipples → Impacted	Impacted	
11	`ButtonPanelTest`	JRipples → Impacted	Impacted	
12	`TestConstants`	JRipples → Impacted	Impacted	

13	AbstractFile	JRipples → Unchanged	Unchanged	Has a getDate() method, nothing else needed
14	AbstractFileTable	JRipples → Propagating	Propagating	Has table with formatted date.
15	CustomDateFormat	JRipples → Impacted	Impacted	Needs new method to create date format w/o time

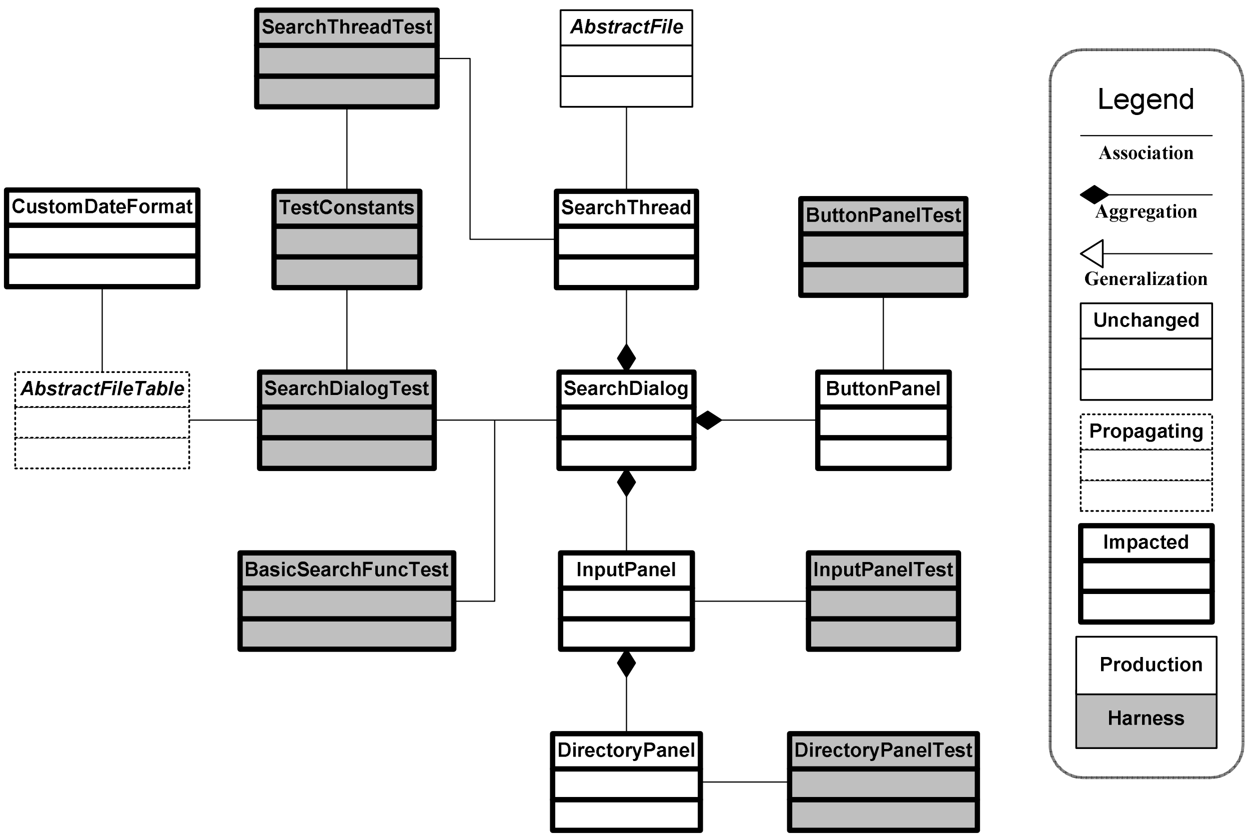


Figure A.26 Change 4 Impact Analysis UML

A.4.4 Prefactoring

To prepare for this change request the programmer extracted the class ErrorManager from DirectoryPanel. The programmer did this because the program will handle multiple types of errors; instead of having SearchDialog check each error to see if it is enabled before a search, it will just check with this new class. This new class will also blink all the enabled errors if the user tries to start a search with an error enabled. This refactoring was done to make the change request easier, not because of existing code smells.

Table A.42 lists the totals of each type of code file visited. Table A.43 lists the code files visited during prefactoring and the LOCs added and deleted in each. A UML diagram of prefactoring is in Figure A.27.

Table A.42 Change 4 Prefactoring Summary

Title	Code Files					
	Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
Date Search	8	8	2	0	0	0

Table A.43 Change 4 Prefactoring Code Files

#	Code File	Task	Lines of Code		
			Added	Deleted	Total
1	`ErrorManager`	Extracted class, added methods	32	0	32
2	`DirectoryPanel`	Extracted class from	10	13	23
3	`InputPanel`	Changed method	3	2	5
4	`SearchDialog`	Changed methods	8	5	13
5	`ButtonPanel`	Changed method	8	2	10
6	`ErrorManagerTest`	Extracted class, added methods	60	0	60
7	`DirectoryPanelTest`	Moved tests from, changed test	5	14	19
8	`InputPanelTest`	Changed method	2	1	3
9	`ButtonPanelTest`	Changed methods	10	0	10
10	`BasicSearchFuncTest`	Changed methods	3	3	6

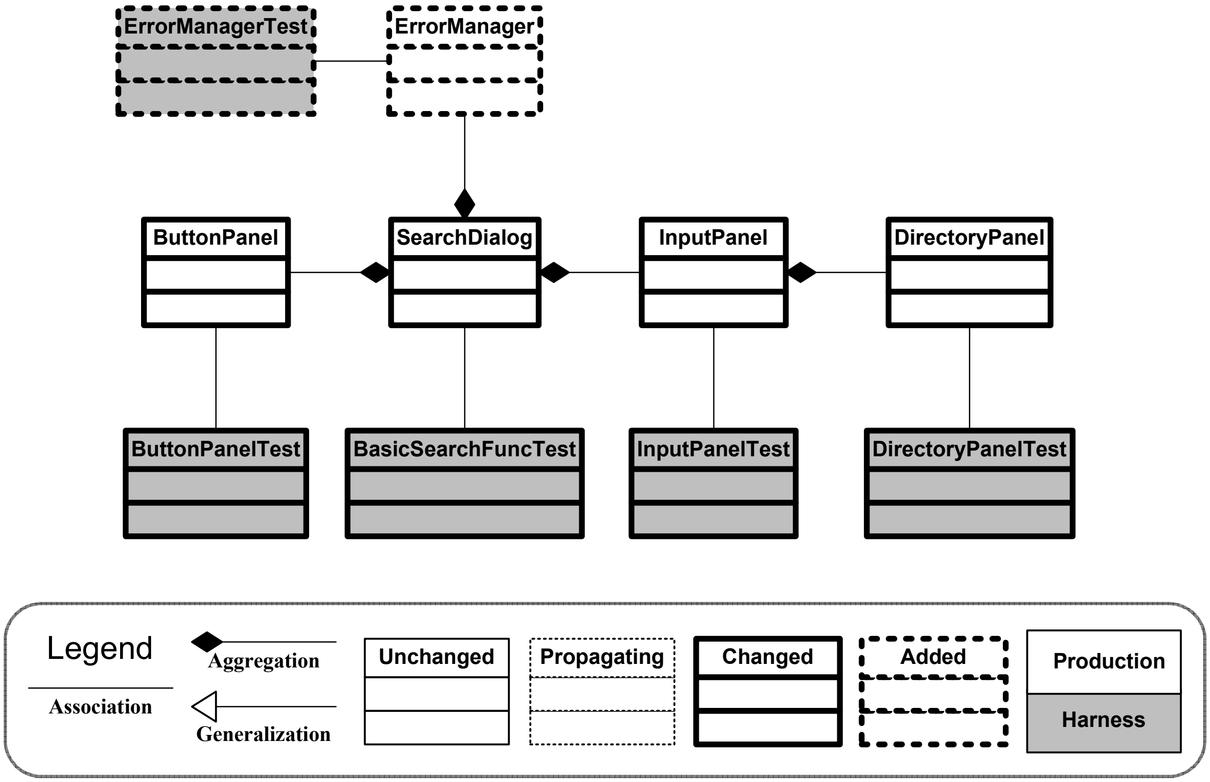


Figure A.27 Change 4 Prefactoring UML

A.4.4.1 ErrorManager class

The programmer extracted this class from `DirectoryPanel`. It has 1 field of type `HashSet` that holds objects of `FlashLabel`. There are 5 methods: `enableError()`, `disableError()`, `flashErrors()` and 2 `isErrorEnabled()` methods. One of the `isErrorEnabled()` methods takes no parameters, it returns true if any errors are enabled, while the other takes a parameter of type `FlashLabel` and it returns true if that error is enabled. The `enableError()` and `disableError()` methods also take a `FlashLabel` and add or remove it from the `HashSet`. The `flashErrors()` methods, just calls the `flash()` method in each enabled `FlashLabel`.

A.4.4.2 DirectoryPanel class

This class had the `ErrorManager` class extracted from it; this included 2 methods, `flashError()` and `isErrorEnabled()`. Three methods that called the method `setVisible()` on the field `directoryError`, now call `ErrorManager`'s `enableError()` and `disableError()` methods.

A field of type `ErrorManager` was added. A parameter of type `ErrorManager` was also added to the constructor, which sets the field to the parameter.

During refactoring the programmer noticed that the visibilities of this classes fields were all set to public. This probably was done by the Eclipse refactoring tool when the class was extracted from `InputPanel` and not noticed at the time. The visibilities were all changed to private, which did not propagate.

A.4.4.3 InputPanel class

This class's constructor changed; it added a parameter of type `ErrorManager`, which it passes to `DirectoryPanel`. This class creates an object of `ErrorManger`.

A.4.4.4 SearchDialog class

The programmer added a field of type `ErrorManager`. It creates an object of that type in the constructor and passes it to the `InputPanel` and `ButtonPanel` objects it creates. The if statement that called the methods `isErrorEnabled()` and `flashError()` in class `DirectoryPanel` was extracted from the method `searchStopButton()` to `ButtonPanel`.

A.4.4.5 ButtonPanel class

This class added a field of type `ErrorManager` and a parameter of the same type to its constructor, which it uses to set the field. An if statement extracted from

SearchDialog was added to the actionPerformed() method. It called a method isErrorEnabled() in DirectoryPanel to check if the error was enabled and if it was called flashError(). These methods were changed to call isErrorEnabled() and flashErrors() in ErrorManager.

A.4.4.6 ErrorManagerTest class

This class is the unit test suite for the ErrorManager class it was added during this change request. It has 5 tests, 2 of which, testFlashErrors() and testIsErrorEnabled() were moved from DirectoryPanelTest.

A.4.4.7 DirectoryPanelTest class

This class is the unit test suite for the DirectoryPanel class. It had 1 test changed and 2 test moved to ErrorManagerTest, testFlashError() and testIsErrorEnabled().

A.4.4.8 InputPanelTest class

This class is the unit test suite for the InputPanel class. It had its setUp() method changed, it had to add a parameter of type ErrorManager to the InputPanel constructor call it makes to create and object of type InputPanel.

A.4.4.9 ButtonPanelTest class

This class is the unit test suite for the ButtonPanel class. It had 1 test added.

A.4.4.10 BasicSearchFuncTest class

This class is a functional test suite. It had 3 tests changed.

A.4.5 Actualization

To actualize this change request, the programmer added a new class of type DatePanel that extends JPanel. This class contains all the GUI components of the change request description. This class gets dates from the user as text and creates Date objects from the text. It performs error checking to make sure that the user entered a valid date and checks to make sure that the minimum date is less than the maximum date. To create a border for the class that has a `JCheckBox` in it the programmer added a class that was provided by Kumar under a GNU License [43]. A test class for it was also added.

To add GUI calendars for the user to select a date, new classes were added by the programmer. These classes were taken from a program called `JCalendar` written by Toedter and available online under the GNU Lesser General Public License [44]. The program contained more functionality then needed so specific classes were chosen. These classes are `JCalendar`, `JDayChooser`, `JMonthChooser`, `JYearChooser` and `JSpinField`. These classes used together made up a very feature rich GUI calendar with a month drop down box and a year text box, both of which have buttons to increment or decrement their values. All of these classes were changed and added into muCommander. They were placed in a new package called `org.severe.ui.dialog.calendar`. A unit test class was added for each class taken from `JCalendar` and a functional test class was added that tests all the classes together. These test classes were all added to a new package, `org.severe.ui.dialog.calendar.tests`.

muCommander displays each file's modified date in the GUI with the time; entering the time when doing a date search seemed overly burdensome. The `CustomDateFormat` class had a static method `getDateNoTimeFormatString()` added that returns a `DateFormat String` based data from the applications preferences file, but without the time. This allows the user to choose a date in the same format as the application display, but without the time.

Table A.44 lists the totals of each type of code file visited. Table A.45 lists the code files visited during actualization and the LOCs added and deleted in each. A UML diagram of actualization is in Figure A.28.

Table A.44 Change 4 Actualization Summary

Title	Code Files					
	Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
Date Search	7	7	16	0	0	0

Table A.45 Change 4 Actualization Code Files

#	Code File	Task	Lines of Code			Comments
			Added	Deleted	Total	
1	`DatePanel`	Added class	308	0	308	
2	`ComponentTitledBorder`	Added class	40	5	45	Imported class started with 94 LOC
3	`CustomDateFormat`	Added method	5	0	5	
4	`JCalendar`	Added class	24	14	38	Imported class started with 147 LOC
5	`JDayChooser`	Added class	25	3	28	Imported class started with 274 LOC
6	`JMonthChooser`	Added class	19	1	20	Imported class started with 170 LOC
7	`JYearChooser`	Added class	6	4	10	Imported class started with 44 LOC
8	`JSpinField`	Added	8	3	11	Imported

		class				class started with 133 LOC
9	ErrorManager	Added, changed methods	5	1	6	
10	InputPanel	Changed methods	11	1	12	
11	SearchThread	Added, changed methods	19	2	21	
12	SearchDialog	Changed method	2	1	3	
13	DatePanelTest	Added class	213	0	213	
14	DateSearchFuncTest	Added class	181	0	181	
15	ComponentTitledBorderTest	Added class	123	0	123	
16	JCalendarTest	Added class	110	0	110	
17	JDayChooserTest	Added class	151	0	151	
18	JMonthChooserTest	Added class	95	0	95	
19	JYearChooserTest	Added class	71	0	71	
20	JSpinFieldTest	Added class	147	0	147	
21	JCalendarFuncTest	Added class	98	0	98	
22	SearchThreadTest	Changed tests	29	4	33	
23	TestConstants	Added field	1	0	1	

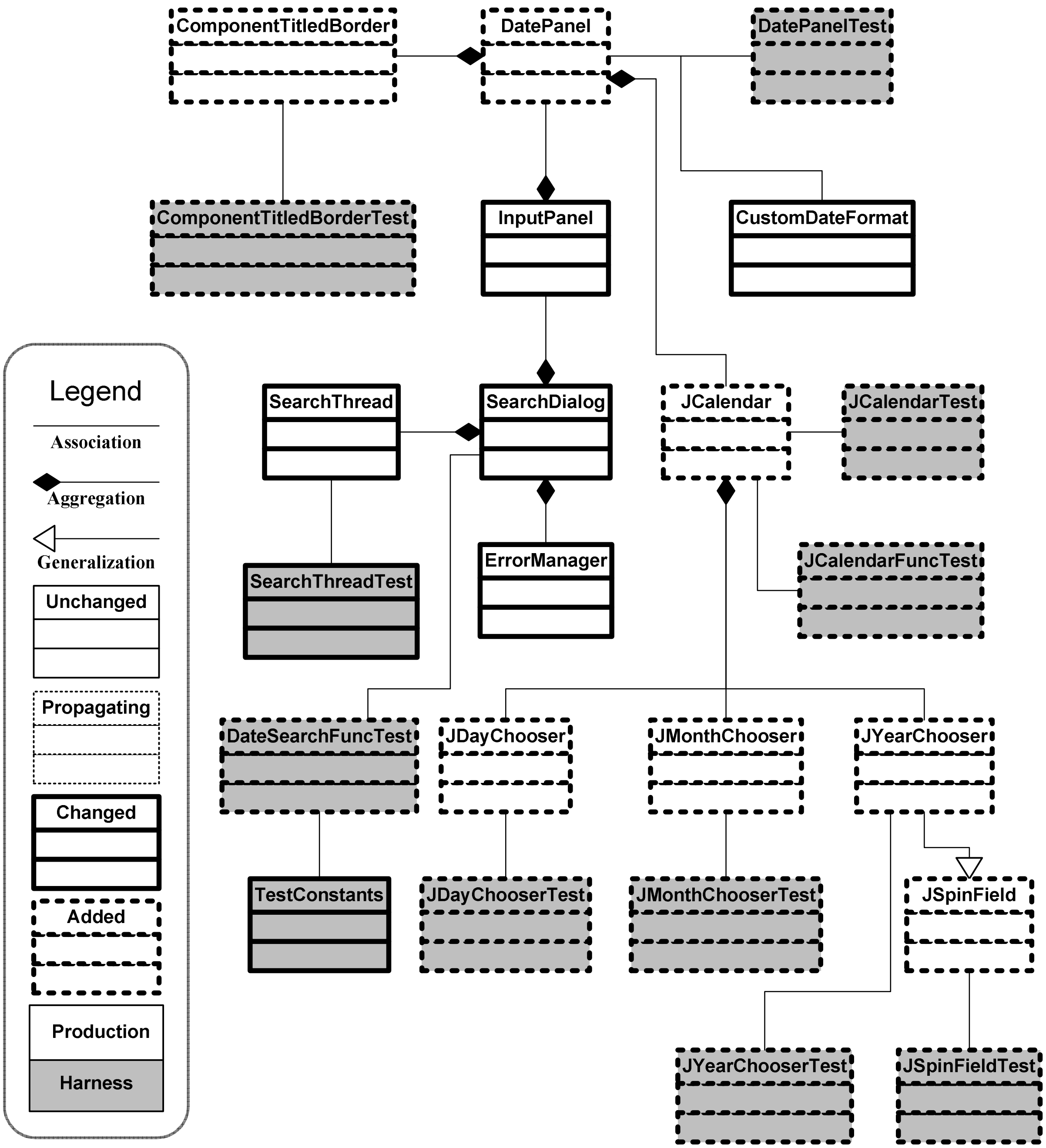


Figure A.28 Change 4 Actualization UML

A.4.5.1 DatePanel class

This class was created during actualization by the programmer. It contains a `JCheckBox` field that allows the user to enable and disable a date search. There are 2 `JTextField` objects for the user to enter dates in, 2 `JButton` objects that open `JCalendar` dialogs, 2 `JLabel` objects to describe the `JTextField` objects, 2 `Date` and 2 `boolean` error fields that can be set when an invalid date is entered. These fields all correspond to a minimum and maximum date range. There are also fields of type `DateFormat` for the date format `String`, a `FlashLabel` to display an error, an `ErrorManager` and a `boolean minGreatestError` that is true when the minimum date is greater than the maximum.

The border for this class was set to a `ComponentTitledBorder` this allows the `JCheckbox` to be added to the border. The methods include `createDateTextBox()`, which initializes the `JTextField` objects and `createCalendarButton()` that initializes the `JButton` objects. The `setEnabled()` method was overridden so that it only enables the `JCheckBox` unless the `JCheckbox` is selected, in which case it enables all the components. The method `datePanelSetSelected()` is called by `setEnabled()` to enable the components. The `actionPerformed()` method listens to the `JCheckBox` and `JButton` fields. The `focusLost()` method listens to the `JTextField` objects and sets the `Date` fields when they lose focus.

The `getErrorMessage()` method returns a `String` error message based upon which `boolean` errors are true. The `isError()` method returns true if any of the `boolean` errors are true. The `dateTextBoxCheck()` method tries to parse the text in the `JTextField` objects into a `Date`. The `checkMinLessThan()` method checks if the minimum `Date` is greater than the maximum `Date`. There are getters for the `Date` fields and an `isDateSearch()` that returns true if the `JCheckBox` is selected. The `keyReleased()` method calls the `dateTextboxCheck()` method if the text in one of the `JTextField` objects is updated and stops displaying the user error if the date has been changed to a valid one.

A.4.5.2 ComponentTitledBorder class

The `ComponentTitledBorder` class was added to the project by the programmer. It was written by Kumar and is available under the terms of the GNU Lesser General Public License [43]. The Java swing libraries do not have a way to add a check box to a panel's border that enables the inner components. This is a very popular way to organize a panel in many C++ frameworks. This class uses the `paintBorder()` method to draw a component such as a check box in the border. It then forwards `MouseEvent` objects that happen to that component to keep the components contracts with its suppliers. The only changes made to this class were to add getters for testing.

A.4.5.3 CustomDateFormat class

One static method was added to this class, `getDateNoTimeFormat()`. It returns a `DateFormat` string based upon the date string defined in the application's preferences file.

A.4.5.4 JCalendar class

This class was written by Toedter and is being used under the GNU Lesser Public License [44]. It and its suppliers, `JDayChooser`, `JMonthChooser`, `JYearChooser` and `JSpinField` create a GUI calendar that a user can select a date from. A GUI dialog calendar is not part of the Swing libraries, but has been done by others in many different ways, so one was selected add added instead of reinventing it.

The programmer made the following changes to this class; it was changed from extending `JPanel` to extending `JDialog`, so that it does not need to be added to a container to be displayed. The constructors were changed; one had an integer parameter removed and replaced with a parent of type `Component`. This allows the dialog to open near the `JButton` that creates an instance of it. The other constructor takes no parameters and opens the dialog in non-modal mode for testing. They both call a new `init()` method that initializes the dialog. This method is similar to the old constructor, but it also adds a `JLabel` to display the dialogs title. An if statement was added to the `propertyChange()` method that disposes of the dialog. Finally, the main method was removed because it will no longer work now that the `JCalendar` extends `JDialog`.

A.4.5.5 JDayChooser class

This class is a supplier to `JCalendar` class and was also written by Toedter [44]. The programmer added 2 fields, 1 a static field of type `int` that gets the system double click interval and the other of type long that records a click time to determine if it is within the double click interval. The constructor was changed to call `setRolloverEnabled()` to false for all of the `JButton` objects that represent the days. The `actionPerformed()` method was changed to listen for both single and double clicks on the `JButton` days. Now if the user double clicks a button, it will call `firePropertyChange()` to tell `JCalendar` to dispose itself. A bug was addressed here, that 2 `ActionEvent` objects can be created when a `JButton` is clicked on. One of these is created without a time and is now ignored. This allowed some commented code in the `keyPressed()` method that allows the user to traverse between days with the arrow keys.

A.4.5.6 JMonthChooser class

This class is a supplier to `JCalendar` class and was also written by Toedter [44]. The only changes made by the programmer were to add getters for testing.

A.4.5.7 JYearChooser class

This class is a supplier to `JCalendar` class and was also written by Toedter [44]. The `setValue()` method's visibility was changed by the programmer from protected to public, so it can be called by `DatePanel`. A getter was added for testing.

A.4.5.8 JSpinField class

This class is a supplier to `JCalendar` class and was also written by Toedter [44]. The programmer made the following changes, the `setValue()` method no longer calls `firePropertyChange()` and the `setValue()` method's visibility was lowered to public from protected for testing. Two getters were added for testing.

A.4.5.9 ErrorManager class

This class had an overloaded method added by the programmer, `enableError()`, with an additional `boolean` parameter. When it is set to false the error is added so the `isErrorEnabled()` method will return true, but the `FlashLabel` will not be set to visible. This was done to make the state of errors is current, but the user can be given time to correct it on their own without having an error displayed until appropriate. The `disableError()` method also added a call to `FlashLabel repaint()` to make sure a disabled error is removed from the GUI.

A.4.5.10 InputPanel class

The programmer added a `DatePanel` to the constructor of this `JPanel` class and a `FlashLabel` error message from `DatePanel`'s `getErrorLabel()` method to its inner panel. This location will be a good place to show errors to the user without crowding the panel where they choose the search criteria. A call to `DatePanel`'s `setEnabled()` method was added to the `switchToSearchState()` method. A getter for the `DatePanel` object was also added.

A.4.5.11SearchThread class

The programmer added a `boolean` field to enable a date search. The constructor added a `boolean` parameter that sets the new field. The `recursiveSearch()` method now calls `isDateInRange()` for each `AbstractFile` to check if it is in the date range specified, if the date search is enabled. The `isDateInRange()` method was added. It takes an `AbstractFile` as a parameter and checks to make sure it is in the date range entered in the `DatePanel`.

A.4.5.12 SearchDialog class

The call in the `searchCommand()` method that creates an object of type `SearchThread` had a parameter added to match the new `SearchThread` constructor. The parameter is set by a call to `DatePanel`'s `isDateSearch()` method.

A.4.5.13 DatePanelTest class

This class was added, it is the unit test suite for the `DatePanel` class; it has 17 tests.

A.4.5.14 DateSearchFuncTest class

This class was added it is a functional test suite for the `DatePanel` class and its suppliers; it has 6 tests.

A.4.5.15 ComponentTitledBorderTest class

This class was added, it is the unit test suite for the `ComponentTitledBorder` class; it has 12 tests.

A.4.5.16 JCalendarTest class

This class was added, it is the unit test suite for the `JCalendar` class; it has 11 tests.

A.4.5.17 JDayChooserTest class

This class was added, it is the unit test suite for the `JDayChooser` class; it has 12 tests.

A.4.5.18 JMonthChooserTest class

This class was added, it is the unit test suite for the `JMonthChooser` class; it has 11 tests.

A.4.5.19 JYearChooserTest class

This class was added, it is the unit test suite for the `JYearChooser` class; it has 5 tests.

A.4.5.20 JSpinFieldTest class

This class was added, it is the unit test suite for the `JSpinField` class; it has 14 tests.

A.4.5.21 JCalendarFuncTest class

This class was added it is a functional test suite for the `JCalendar` class and its suppliers; it has 6 tests.

A.4.5.22 SearchThreadTest class

This class is the unit test suite for the `SearchThread` class. It had 4 test changed and 2 tests added.

A.4.5.23 TestConstants class

This class contains static final fields used by the test suite. It added a field of type long that is set to the length of a day in milliseconds.

A.4.6 Postfactoring

The `DatePanel` class that was created during actualization by the programmer was too large and had too much responsibility. So the class `DateField` was extracted from it. It extends the `JTextField` class; it adds methods to customize the class to only accept `Date` objects. The class handles the parsing of text to `Date` objects.

The programmer extracted an abstract class, `SearchFuncTestSetUp` from `BasicSearchFuncTest` and `DateSearchFuncTest` that has `setUp()` and `tearDown()` methods. It is similar to the class `SearchDialogTestSetUp` that was extracted during change 2. All 3 of these classes were put in a new package `org.severe.ui.dialog.search.functional.tests`. These functional tests take significantly longer to run than unit test; having them in their own package makes it easier to run them separately. The programmer did this extraction because the functional tests expanded to 2 classes with similar `setUp()` and `tearDown()` methods during actualization.

The other classes changed during postfactoring were cleaned up; for example, unused methods were removed, fields were moved to the beginning of the class as other classes in muCommander and the Javadoc was updated. In the classes added from other sources, `JCalendar`, its suppliers and `ComponentTitledBorder` this was necessary because these classes were intended for general use. There were some parts that did not match the code style of muCommander and were not needed. In the case of existing classes such as `SearchThread`, the cleanup was made necessary because of actualization.

Table A.46 lists the totals of each type of class visited. Table A.47 lists the classes visited during postfactoring and the LOCs added and deleted in each. A UML diagram of postfactoring is in Figure A.29.

Table A.46 Change 4 Postfactoring Summary

Title	**Code Files**					
	Visited	**Changed**	**Added**	**Propagating**	**Unchanged**	**Added to Changed Set**
Date Search	20	19	3	1	0	0

Table A.47 Change 4 Postfactoring Code Files

#	**Code File**	**Task**	**Lines of Code**			**Comments**
			Added	**Deleted**	**Total**	
1	`DatePanel`	Extracted class from, extracted methods	58	180	238	
2	`DateField`	Extracted class	121	0	121	
3	`ComponentTitledBorder`	Javadoc	0	0	0	
4	`JCalendar`	Removed field, changed methods	7	25	32	
5	`JDayChooser`	Moved fields, methods	14	33	47	
6	`JMonthChooser`	Moved fields methods	10	29	39	
7	`JYearChooser`	Moved fields, methods	4	15	19	
8	`JSpinField`	Moved fields	7	10	17	

9	SearchThread	Javadoc	0	0	0	
10	SearchFuncTestSetUp	Class extracted	71	0	71	
11	BasicSearchFuncTest	Class extracted from	2	66	68	
12	DateSearchFuncTest	Class extracted from	25	78	103	
13	ComponentTitledBorderTest	Javadoc	0	0	0	
14	DateFieldTest	Extracted class	115	0	115	
15	DatePanelTest	Extracted class from	20	102	122	
16	JCalendarTest	Javadoc	0	1	1	
17	JDayChooserTest	Javadoc	4	3	7	
18	JMonthChooserTest	Javadoc	0	0	0	
19	JYearChooserTest	Method removed	4	3	7	
20	JSpinFieldTest	Javadoc	0	2	2	
21	JCalendarFuncTest	Method removed	1	9	10	
22	SearchThreadTest	Javadoc	5	4	9	

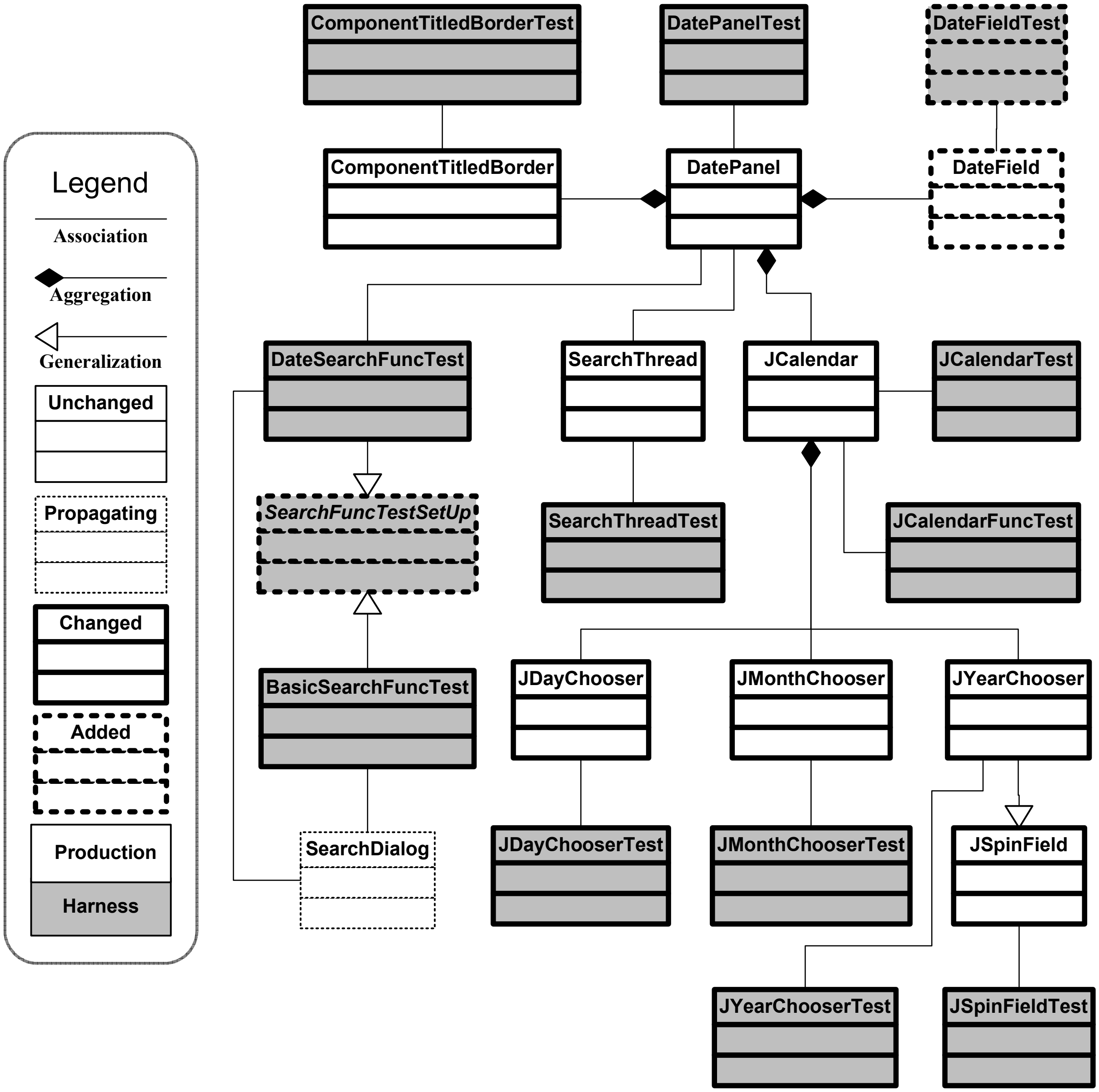


Figure A.29 Change 4 Postfactoring UML

A.4.6.1 DatePanel class

This class had a class, `DateField`, extracted from it. Two `Date` fields and 2 `boolean error` fields were extracted, along with parts of the methods `createDateTextBox()`, `actionPerformed()` and all of `focusLost()`, `focusGained()`, `dateTextBoxCheck()`, `keyPressed()`, `keyReleased()`, `keyTyped()` and `checkYear()`. This included the responsibility for initializing the

JTextField objects that the user can enter dates in and parsing the text to create Date objects. The JTextField fields changed their types to DateField objects.

A PropertyChangeListener interface was added; it listens for PropertyChangeEvent objects from the 2 DateField objects. A new showError() method was extracted from actionPerformed(), datePanelSetEnabled() and propertyChanged().

A.4.6.2 DateField class

This class was extracted from DatePanel. It extends JTextField; it adds fields of type Date, DateFormat, SimpleDateFormat, a boolean for errors and 3 static final String objects used to identify PropertyChangeEvent objects it fires. The class implements the KeyListener and FocusListener interfaces. There is a setter for the Date, which will also call firePropertyEvent() to notify listeners that the date has changed. A setText() method that takes a Date as a parameter, an isError() method that returns true if an invalid date is entered in the field.

The dateTextBoxCheck() method was extracted from DatePanel, but it was simplified; before it had a JTextField parameter, but now since it can only check its JTextField, it was removed. The error message responsibility was also removed from this method. The checkYear() method was extracted from DatePanel. The only change was to make its temporary variable of type SimpleDateFormat a class field.

The focusLost() method now calls setText() with the Date field and firePropertyChange() to inform listeners they should now display an error message, if appropriate. The keyReleased() method was extracted from

`DatePanel`; it could be reduced because it does not have to have different paths for 2 `JTextField` objects. It now just handles its own `KeyEvent` objects.

A.4.6.3 ComponentTitledBorder class

This class had its Javadoc updated.

A.4.6.4 JCalendar class

This class had its unused Locale field removed, along with its getters and setters. The method `setCalendar()` called `firePropertyChange()` but there are no listeners for it, so it was removed. The fields were moved from the end of the class file to the beginning to match the rest of muCommander and Javadoc added.

A.4.6.5 JDayChooser and JMonthChooser class

These classes had the getter and setter for `Locale` removed, they will only use the system `Locale`. Their `main()` methods were removed, they are not needed. The fields were moved from the end of the class file to the beginning to match the rest of muCommander and Javadoc added.

A.4.6.6 JYearChooser class

This class had its unneeded `main()` method removed. The fields were moved from the end of the class file to the beginning to match the rest of muCommander and Javadoc added.

A.4.6.7 JSpinField class

The fields were moved from the end of the class file to the beginning to match the rest of muCommander and Javadoc added.

A.4.6.8 SearchThread class

This class had its Javadoc updated.

A.4.6.9 SearchFuncTestSetUp abstract class

This class was extracted from `BasicSearchFuncTest` and `DateSearchFuncTest`. It contains the `setUp()` and `tearDown()` methods that create an instance of `SearchDialog` for testing. It has 8 fields corresponding to regularly used components of the `SearchDialog` for the test to use. It also has 3 tester fields that are part of Abbot that the tests can use.

A.4.6.10 BasicSearchFuncTest class

This class is a functional test suite. It had its `setUp()` and `tearDown()` methods extracted to `SearchFuncTestSetUp`, along with all of its fields.

A.4.6.11 DateSearchFuncTest class

This class is a functional test suite. It had its `setUp()` and `tearDown()` methods extracted to `SearchFuncTestSetUp`, it still has a `setUp()` method call that calls the super method and initializes its 2 remaining fields, 9 were extracted. It had 2 tests and 1 test helper method changed.

A.4.6.12 ComponentTitledBorderTest class

This class had its Javadoc updated.

A.4.6.13 DateFieldTest class

This class is the test suite for the `DateField` class. Seven tests were moved from `DatePanelTest` then they were combined into 3 tests. There are 8 total tests.

A.4.6.14 DatePanelTest class

This class is the test suite for the `DatePanel` class. Seven tests were moved from `DatePanelTest`. Three tests were changed, there are 10 remaining tests.

A.4.6.15 JCalendarTest class

This class is the test suite for the `JCalendar` class; its Javadoc was updated and used imports removed.

A.4.6.16 JDayChooserTest class

This class is the test suite for the `JDayChooser` class; its Javadoc was updated and used imports removed.

A.4.6.17 JMonthChooserTest class

This class is the test suite for the `JMonthChooser` class; its Javadoc was updated.

A.4.6.18 JYearChooserTest class

This class is the test suite for the `JYearChooser` class; its Javadoc was updated, used imports and before class was removed.

A.4.6.19 JSpinFieldTest class

This class is the test suite for the `JSpinField` class; its Javadoc was updated and used imports removed.

A.4.6.20 JCalendarFuncTest class

This class is the functional test suite for the `JCalendar` class and its suppliers; its Javadoc was updated, used imports and `tearDown()` method was removed.

A.4.6.21 SearchThreadTest class

This class is the test suite for the `SearchThread` class; its Javadoc was updated and used imports removed.

A.4.7 Verification

After prefactoring and postfactoring all the regression tests passed. No new regression tests were added. All tests are in their own package, which has the same name as the package containing the code file being tested plus tests. There are 3 functional test classes. During verification 2 bugs were found, both in the new classes created during actualization.Table A.48 lists the coverage of each production code file added during the SIP and its statement coverage by the test suite.

Table A.48 Change 4 Statement Verification

#	Code file	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	`SearchDialog`	43	42	97.7	0	0
2	`SearchThread`	40	38	95.0	0	0
3	`ErrorManager`	13	13	100.0	0	0
4	`ComponentTitledBorder`	35	35	100.0	0	0
5	`DateField`	55	54	98.2	0	0
6	`ButtonPanel`	26	26	100.0	0	0
7	`DatePanel`	89	86	96.6	0	2
8	`DirectoryPanel`	50	41	82.0	0	0
9	`InputPanel`	36	36	100.0	0	0
10	`JCalendar`	75	60	80.0	0	0
11	`JDayChooser`	142	133	93.7	0	0
12	`JMonthChooser`	76	63	82.9	0	0
13	`JYearChooser`	15	15	100.0	0	0
14	`JSpinField`	64	54	84.4	0	0
15	`CustomDateFormat`	22	13	59.1	0	0

The first bug was in `DatePanel`; if the user types a date with a 2 digit year, the `Date` was parsed as 1st century year. A new method was added to parse the `Date` into a user expected date. The second bug was that a the `Date` objects were not being read in before a search started, which could cause a search without a `Date`, even though a date was displayed to the user. Adding a `KeyListener` to parse the `Date` after each keystroke solved the problem.

A.4.8 Timing Data

Table A.49 contains the timing data for the change request.

Table A.49 Change 4 Timing Totals

Phase	Time (hh:mm)
Concept Location	0:00
Impact Analysis	1:26
Prefactoring	1:41
Prefactoring Testing	0:41
Actualization	4:42
Actualization Testing	3:34
Postfactoring	4:46
Postfactoring Testing	1:28

A.4.9 Conclusions

This change request added a significant number of code files to muCommander, but the change request required less effort than change 3. This is because the programmer reused 6 code files from outside sources that just needed slight modifications to be added to the project. These code files provided functionality that is missing from the Swing libraries, but are available in many other language libraries and frameworks. For example, the `ComponentTitledBorder` is a popular feature in many C++ frameworks. This is why there was no real reason to write these classes again, many others have already solved these problems and made them available for use.

The impact set was 1 code file smaller than the estimated impact set. The `SearchDialogTest` code file did not need to be changed. It is difficult to determine how the test code files will change. In this case, the programmer assumed that since `SearchDialog` needed to change, then its test would change. However, only the call to `SearchThread`'s constructor needed to change. This did not require any additional testing.

This change request presented a challenge to coordinate the date parsing and error messages. Making sure a search cannot happen with an invalid date, but not displaying the date so frequently, is complicated. The quirks of the Gregorian calendar are broad; the programmer believes that there is a high probability of bugs appearing at certain dates. Looking at the code after postfactoring, it is clear that having the `Date` parsing done in 1 code file and another code file handle the responsibility of when to display the date was much simpler. An easier solution would have been to create the `DateField` code file first, but that design was not apparent to the programmer at the time.

The prefactoring of extracting a class to manage the errors will make future change requests that require displaying an error easier with a smaller impact set. For instance, the `ButtonPanel` now checks with the `ErrorManager` class when the `JTextField startStopButton` is pressed; so if a new error is needed, so long as it uses the `ErrorManager` class, `ButtonPanel` will not be impacted, but it will still know if an error is enabled or not.

Table A.50 shows the total number of code files in each set of each phase of the change request. Table A.51 is the current state of the product backlog. Figure A.30 to Figure A.33 show screen shots of muCommander before and after the change request.

Table A.50 Change 4 Code File Summary

#	Change	Number in Code Files						
		Visited Concept Location	Estimated Impact Set	Impact Set	Added during			Total Added
					Pre	Act	Post	
0	Original Baseline	N/A	N/A	N/A	N/A	N/A	N/A	1,070
1	Basic Search	5	3	4	0	4	0	1,074
2	Recursive search	0	3	4	4	0	5	1,083
3	Advanced Output	6	21	11	2	4	10	1,099
4	Date Search	0	13	12	2	16	3	1,120

Table A.51 Change 4 Current Product Backlog

#	Title	Complete	User Story
1	Basic Search	x	Add a basic search function that allows a user to search in the current directory for all or part of the title of a folder or file, and return a list of the matching files and directories.
2	Recursive Search	x	Add the ability to search inside all directories.
3	Advanced Output	x	Change the output to a table similar to the main muCommander window.
4	Date Search	x	Allow the user search by a date of file's modification.
5	Case Sensitive Search		Add capability to search by case sensitive search terms.
6	Extension Search		Add the ability to search for files with specific extensions.
7	Properties Search		Add options to search for files based on their properties.
8	Size Search		Add the ability to search for a file by its size.
9	Regular Expression Search		Add capability to search by a regular expression.
10	Lucene Search		Incorporate the Apache Lucene search.

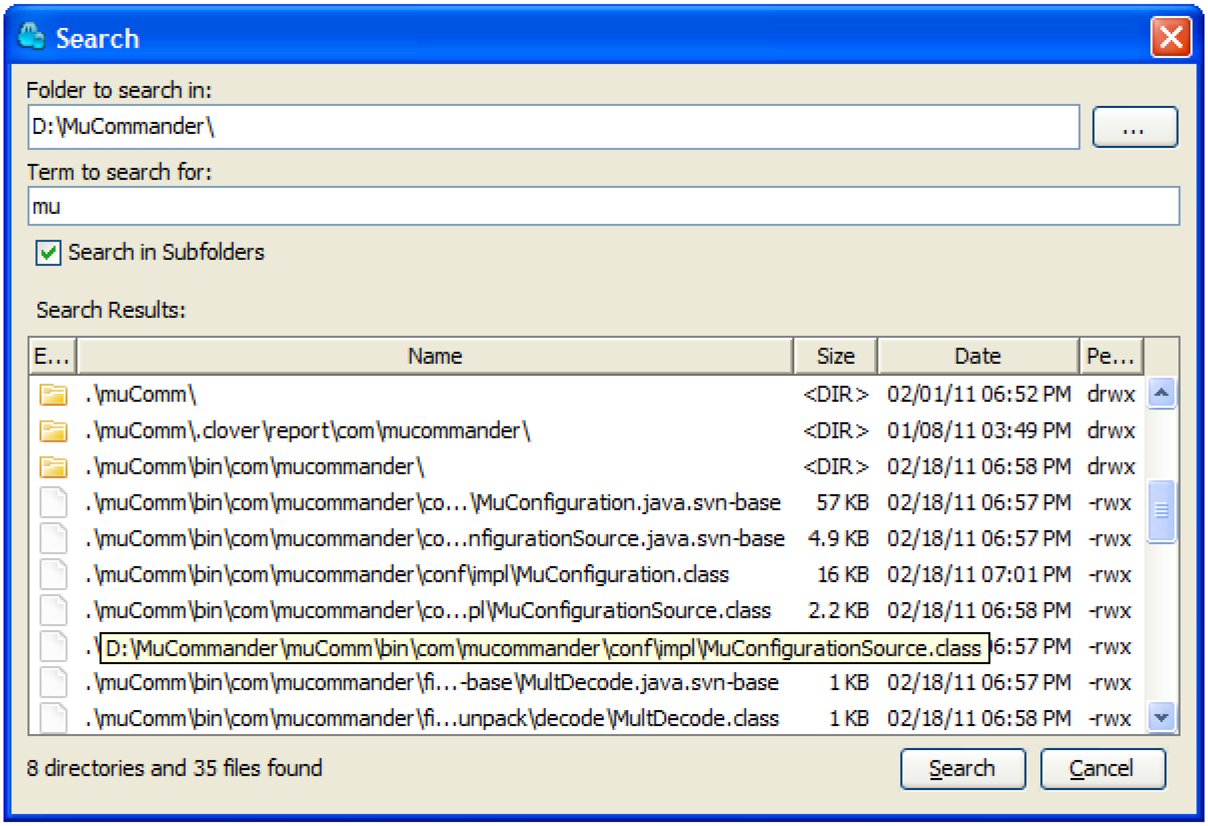


Figure A.30 Search window before Date Search Change

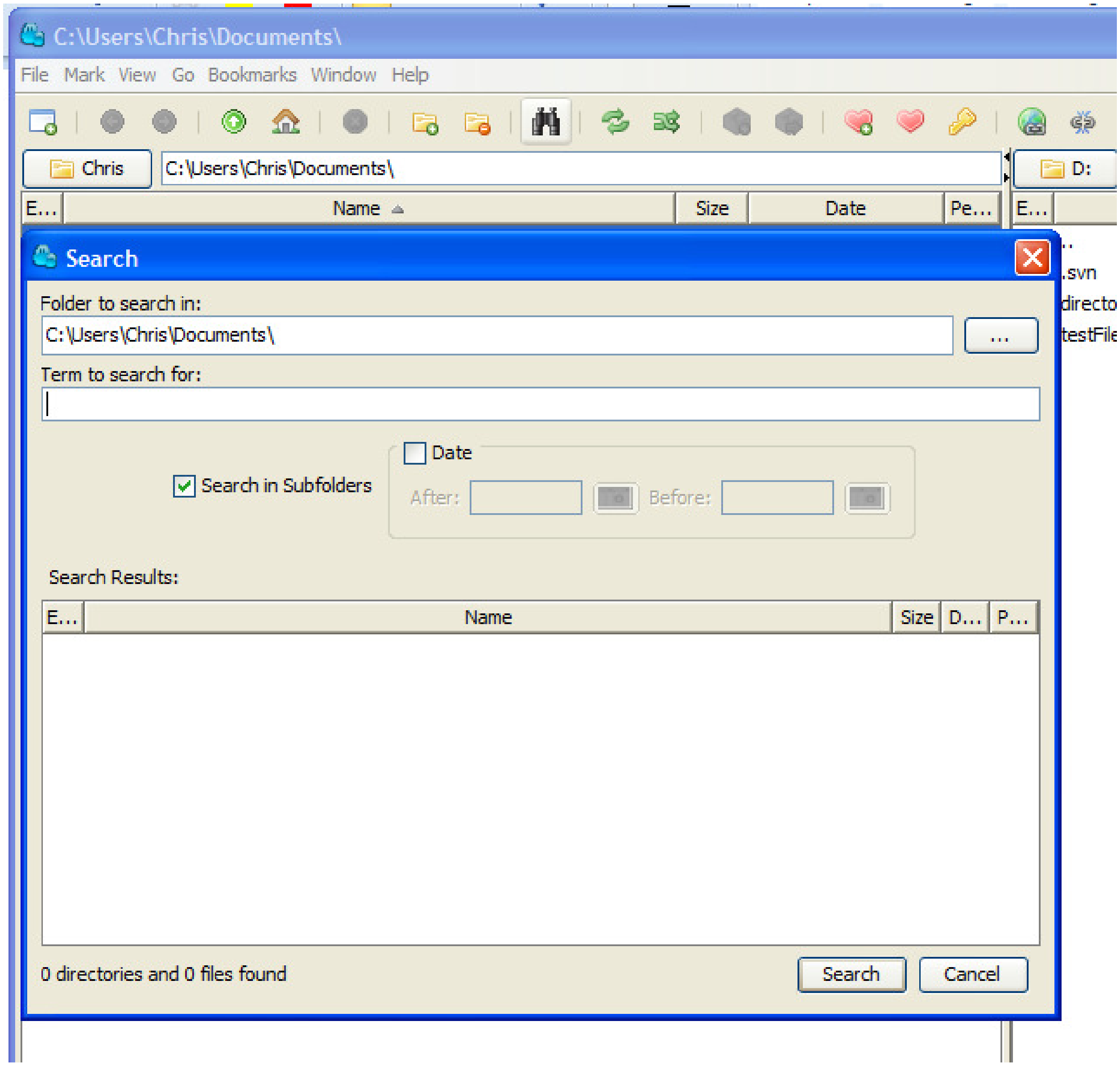


Figure A.31 Search window after the Date Search Change

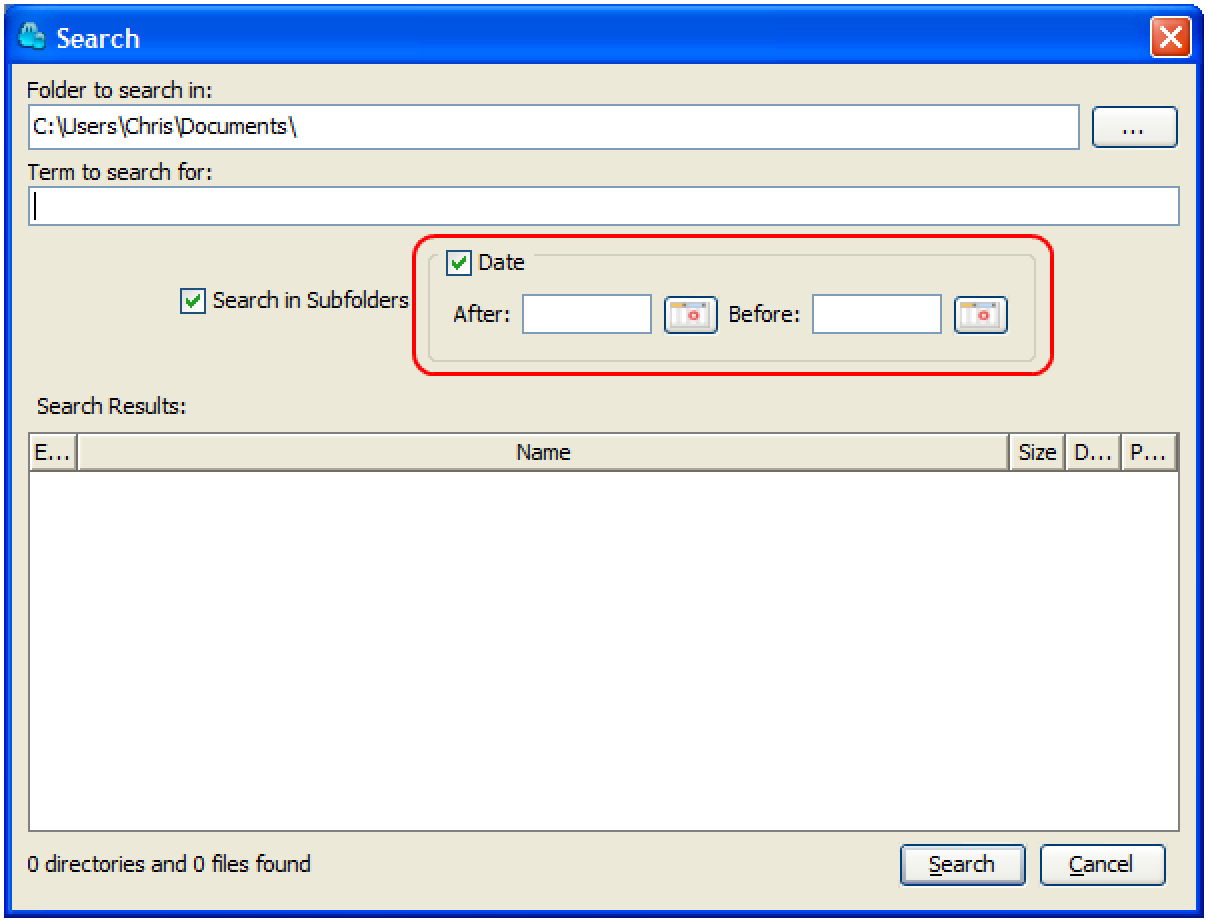


Figure A.32 Search window with date search circled

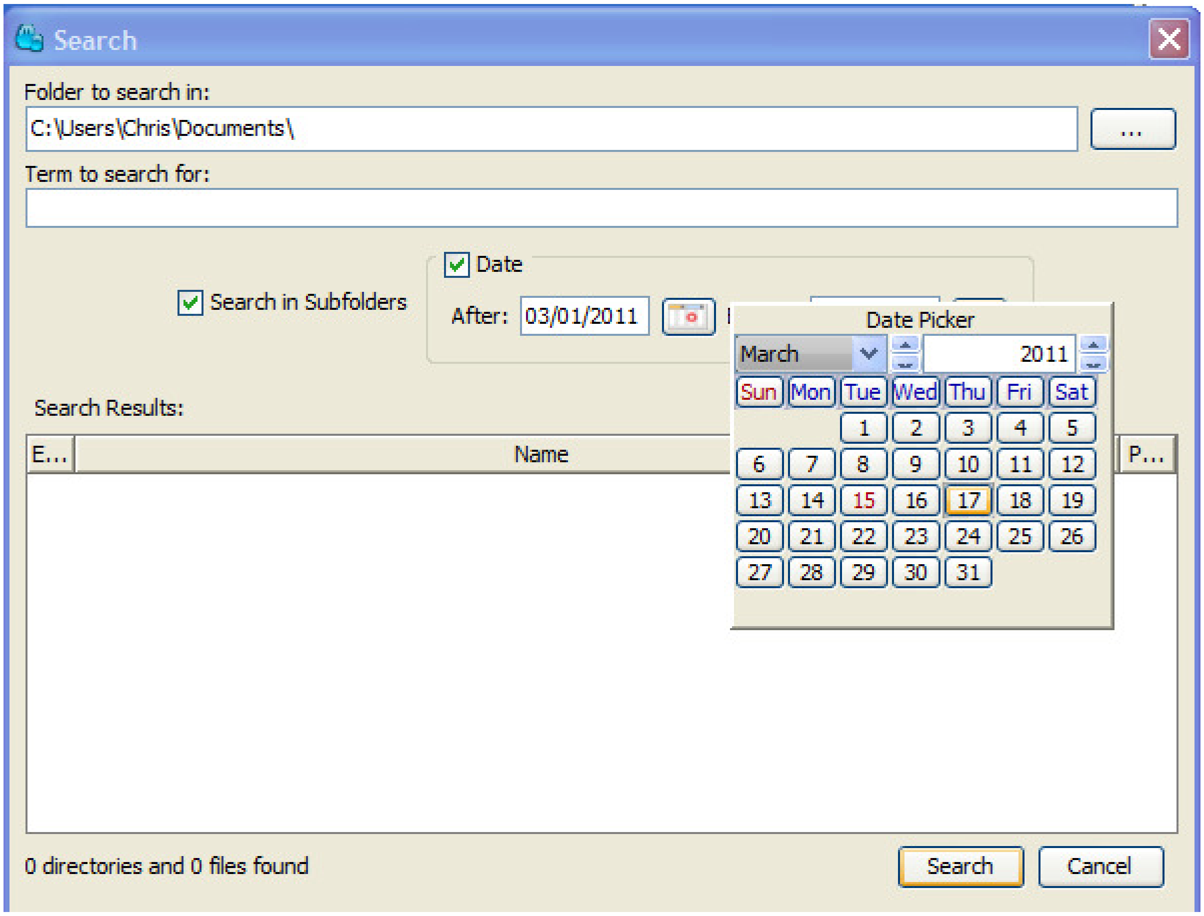


Figure A.33 Search window with date search calendar

SIP – Change 5 Case Sensitive Search

A.5.1Initialization

The program, muCommander enhances an operating systems file explorer. During the first 4 change requests, search capabilities were added; which include: searching for a term, searching in any file system directory, recursively searching in subfolders, displaying results in a GUI table with the look and feel of the muCommander application and searching within a specified date range.

This change request is: “Add capability to search by case sensitive search terms.” A check box will be added to the GUI display that will allow the user to turn this capability on and off. To organize the GUI better a border will be added around the new check box and the recursive search check box.

A.5.2 Concept Location

No concept location was needed for this change. Based on experience obtained during previous changes the programmer knew the search is performed by the `SearchThread` class which was created during change 2.

A.5.3 Impact Analysis

To start impact analysis the programmer marked `SearchThread` as Impacted in JRipples. This marked 9 classes as Next. During the visit the programmer realized that `SearchThread` had 2 responsibilities, 1 to iterate through the files of the file system and 1 to check if each file met the search criteria. This made sense at first, because there was only 1 search criterion, the file name. However, a second, date search criteria was added in the last change and a third was going to be added this change. The programmer decided to refactor this responsibility from `SearchThread` during prefactoring. This requirement influenced the programmer's decisions during impact analysis.

The class `InputPanel` was visited and marked as Impacted because it contains the GUI panel that the case sensitive check box will be added to; JRipples added 12 classes to the Next set. The programmer then visited `SearchDialog`, which was marked as Impacted because a new class created during this change that holds all the search criteria would be instantiated there. JRipples increased the Next set to 30 classes. The programmer then visited `DatePanel`, which was marked as Impacted because it would be affected by the prefactoring. JRipples increased the Next set to 36 classes. To make the prefactoring already mentioned easier, the responsibility for the buttons that open the date picker would be moved from `DatePanel` to `DateField`; therefore `DateField` was also marked as Impacted. JRipples added 1 class to the Next set, so it was still 36 classes. The programmer visited `ButtonPanel` and did not see any reason it would be impacted, it was marked Unchanged. `DirectoryPanel` was visited next; the user chooses the directory to search through this class, which is related to the search criteria, so it was marked as Impacted. JRipples added 3 classes to the Next set; a subset of the Next set, the 21 classes that are not test classes were visited by the programmer and marked Unchanged. These classes did not have any responsibility related to the search criteria.

The programmer then visited the test classes and marked `SearchThreadTest`, `InputPanelTest`, `SearchDialogTest`, `DatePanelTest`, `DateFieldTest`, `DirectoryPanelTest`, `ButtonPanelTest`, `BasicSearchFuncTest`, `DateSearchFuncTest` and `SearchFuncTestSetUp` as impacted. These are the test classes for classes in the Impact set already, except for `ButtonPanelTest`; it is the test for, `ButtonPanel`, which is not in the impact set. It is impacted, because one of its tests calls a method, `searchCommand()` in `SearchDialog` that will be modified. The remaining 5 test classes were marked as Unchanged. After the programmer marked all these classes, JRipples added 13 classes as Next. The programmer marked these classes as Unchanged. They are all required by the various impacted test classes to set up the tests and would not be modified.

The total classes of each mark are listed in Table A.52 and the classes visited during impact analysis are listed in Table A.53. A UML diagram of impact analysis is shown in Figure A.34.

Table A.52 Change 5 Impact Analysis Summary

Title	Code Files					Comments
	Visited	Impacted	Propagating	Unchanged	Not Visited	
Case Sensitive Search	57	16	0	41	0	

Table A.53 Change 5 Impact Analysis Code Files Visited

#	Code File	Tool used	Impacted?	Comments
1	`SearchThread`	JRipples → Impacted	Impacted	Concept Location
2	`InputPanel`	JRipples → Impacted	Impacted	Case sensitive check box to be added here
3	`SearchDialog`	JRipples → Impacted	Impacted	Will add new class object to manage search criteria
4	`DatePanel`	JRipples → Impacted	Impacted	Extract responsibility to DateField
5	`DateField`	JRipples → Impacted	Impacted	Receive responsibility from DatePanel
6	`ButtonPanel`	JRipples → Unchanged	Unchanged	
7	`DirectoryPanel`	JRipples → Impacted	Impacted	Will be impacted by search criteria prefactoring
8	`AbstractFile`	JRipples → Unchanged	Unchanged	Already returns file’s name with case
9	`ActionProperties`	JRipples → Unchanged	Unchanged	
10	`AppLogger`	JRipples → Unchanged	Unchanged	

11	AuthException	JRipples → Unchanged	Unchanged	
12	ComponentTitledBorder	JRipples → Unchanged	Unchanged	
13	CustomDateFormat	JRipples → Unchanged	Unchanged	
14	ErrorManager	JRipples → Unchanged	Unchanged	
15	FileFactory	JRipples → Unchanged	Unchanged	
16	FlashLabel	JRipples → Unchanged	Unchanged	
17	FocusDialog	JRipples → Unchanged	Unchanged	
18	FolderPanel	JRipples → Unchanged	Unchanged	
19	IconManager	JRipples → Unchanged	Unchanged	
20	JCalendar	JRipples → Unchanged	Unchanged	
21	MainFrame	JRipples → Unchanged	Unchanged	
22	ResultsPanel	JRipples → Unchanged	Unchanged	
23	SearchAction	JRipples → Unchanged	Unchanged	
24	SearchTable	JRipples → Unchanged	Unchanged	
25	SearchTableModel	JRipples → Unchanged	Unchanged	
26	SpinningDial	JRipples → Unchanged	Unchanged	
27	Translator	JRipples → Unchanged	Unchanged	
28	YBoxPanel	JRipples → Unchanged	Unchanged	
29	SearchThreadTest	JRipples →	Impacted	

		Impacted		
30	InputPanelTest	JRipples → Impacted	Impacted	
31	SearchDialogTest	JRipples → Impacted	Impacted	
32	DatePanelTest	JRipples → Impacted	Impacted	
33	DateFieldTest	JRipples → Impacted	Impacted	
34	DirectoryPanelTest	JRipples → Impacted	Impacted	
35	ButtonPanelTest	JRipples → Impacted	Impacted	
36	BasicSearchFuncTest	JRipples → Impacted	Impacted	
37	DateSearchFuncTest	JRipples → Impacted	Impacted	
38	SearchFuncTestSetUp	JRipples → Impacted	Impacted	
39	SearchTableTest	JRipples → Unchanged	Unchanged	
40	SearchTableCellRendererTest	JRipples → Unchanged	Unchanged	
41	SearchTableHeaderTest	JRipples → Unchanged	Unchanged	
42	ResultsPanelTest	JRipples → Unchanged	Unchanged	
43	SearchTableModelTest	JRipples → Unchanged	Unchanged	
44	SearchDialogTestSetUp	JRipples → Unchanged	Unchanged	
45	AbstractFileTable	JRipples → Unchanged	Unchanged	
46	AbstractFileTableModel	JRipples → Unchanged	Unchanged	
47	AbstractFolderPanel	JRipples → Unchanged	Unchanged	

48	ActionKeymapIO	JRipples → Unchanged	Unchanged	
49	ActionManager	JRipples → Unchanged	Unchanged	
50	AnimatedIcon	JRipples → Unchanged	Unchanged	
51	Column	JRipples → Unchanged	Unchanged	
52	CommandBarIO	JRipples → Unchanged	Unchanged	
53	DesktopManager	JRipples → Unchanged	Unchanged	
54	ShutdownHook	JRipples → Unchanged	Unchanged	
55	TestConstants	JRipples → Unchanged	Unchanged	
56	ThemeManager	JRipples → Unchanged	Unchanged	
57	WindowManager	JRipples → Unchanged	Unchanged	

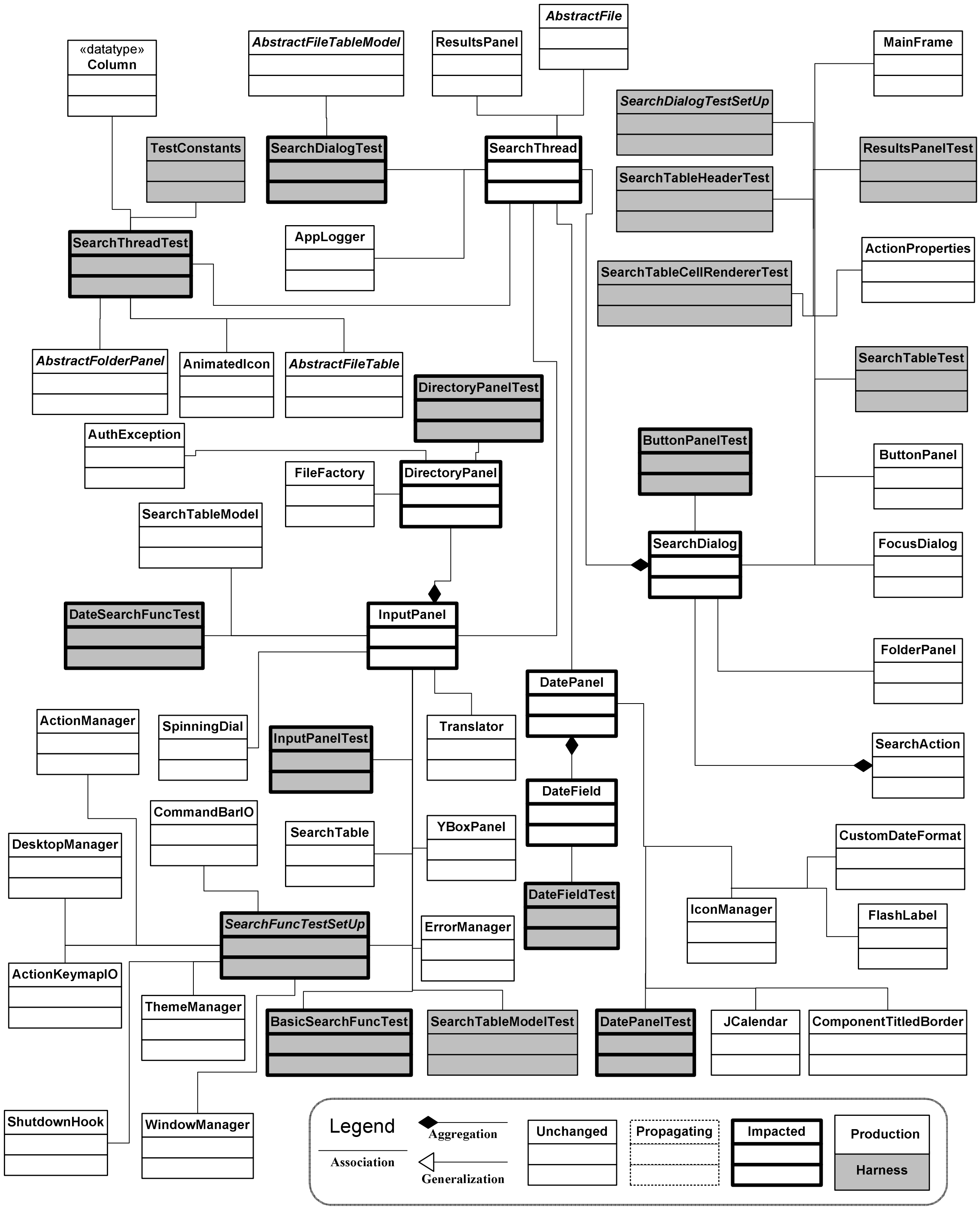


Figure A.34 Change 5 Impact Analysis UML

A.5.4 Prefactoring

The `SearchThread` class searches the file system and contains the logic that decides if a file matches the search criteria and should be added to the set of results or not. During the last change a method was added to it that checks if a file's modified date is within a user specified date. The current structure encourages any new change that adds a search criterion to add a new method with logic that checks the specific criteria. Then the `recursiveSearch()` method, will call this method to see if a file meets the criteria. This will make `SearchThread` a very large class, with a wide variety of responsibilities. To stop this from occurring a Strategy design pattern was implemented [42]. A new class was created to manage the search criteria responsibility, `SearchManager`. An interface, `SearchOption`, was also created. Classes that implement this interface can be added to a list in `SearchManager` dynamically. These classes contain their own algorithms to decide if a file meets their responsibility of the search criteria. When a search is executed, `SearchManager` will check with all the classes on its list to decide if a file meets all the search criteria. The class `SearchThread` had the responsibility to check the date of a file extracted from it to a new class, `DateOption` that implements `SearchOption`; `SearchThread` then had just its original responsibility, of recursively finding the files in the file system.

This prefactoring moved the concept location from `SearchThread` to `SearchManager`. This was done to make actualization simpler and to make future changes easier. It is now possible to add many different search criteria to the program with a small impact set. This prefactoring also meant that the class that contains the concept location, `SearchManager`, would not need to be changed during actualization.

After, the new SearchManager and DateOption classes were created, it became apparent that some of the responsibility left in DatePanel during the last change, should be moved to DateField; namely the JButton that opens a dialog that allows the user to select a date from a calendar. The DateField class was extracted from DatePanel because it had enough responsibility to warrant its own class. However, now either DatePanel or DateField must create an object of a new class, DateOption that will implement the date checking algorithm. Instead of DatePanel creating 2 objects of this new class, each DateField will implement its own object of DateOption. This left 2 objects of type JButton in DatePanel that could be moved to DateField. This refactoring could have been done during the postfactoring phase of change 4, but it was not clear to the programmer at that time. The necessity of adding the new DateOption object, made this refactoring clear.

The other classes that have responsibility to match the search criteria were also changed. The responsibility for matching the search term to the file's name was moved from the InputPanel class to a new class SearchTermOption, which implements SearchOption.

The recursive search and start directory responsibility were also moved to SearchManager, so that all of the search logic would be in 1 class. However, these criteria were given their own methods in SearchManager, because they are not compared against a file's criteria, but rather they set up the search.

The total of each class by type of visit is listed in Table A.54. Table A.55 is a summary of the refactoring type and LOC added and deleted during prefactoring. Figure A.35 is a UML of prefactoring.

Table A.54 Change 5 Prefactoring Summary

Title	Code Files					
	Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
Case Sensitive Search	15	15	8	0	0	0

Table A.55 Change 5 Prefactoring Code Files

#	Code File	Task	Lines of Code		
			Added	Deleted	Total
1	`SearchThread`	Extracted class from	11	32	43
2	`SearchDialog`	Added field, modified method	10	8	18
3	`SearchManager`	Extracted class	49	0	49
4	`SearchOption`	Created interface	6	0	6
5	`DatePanel`	Extracted fields, methods from	28	88	116
6	`DateField`	Extracted fields, methods	71	33	104
7	`DateOption`	Extracted class	58	0	58
8	`InputPanel`	Added field, modified methods	42	19	61
9	`DirectoryPanel`	Added field, modified methods	8	3	11
10	`SearchTermOption`	Extracted class	37	0	37
11	`SearchThreadTest`	Modified method, tests	20	13	33
12	`SearchDialogTest`	Modified method, test	8	9	17
13	`SearchManagerTest`	Modified tests	92	0	92
14	`DatePanelTest`	Modified method, tests	3	25	28
15	`DateFieldTest`	Added method, modified tests	55	12	67
16	`DateOptionTest`	Modified tests	75	0	75
17	`InputPanelTest`	Modified method	3	2	5
18	`DirectoryPanelTest`	Modified methods	3	6	9
19	`SearchTermOptionTest`	Added test class	56	0	56
20	`ButtonPanelTest`	Modified test	4	1	5
21	`BasicSearchFuncTest`	Modified tests	4	4	8
22	`DateSearchFuncTest`	Modified tests	7	12	14
23	`SearchOptionTestClass`	Added class for tests	14	0	14

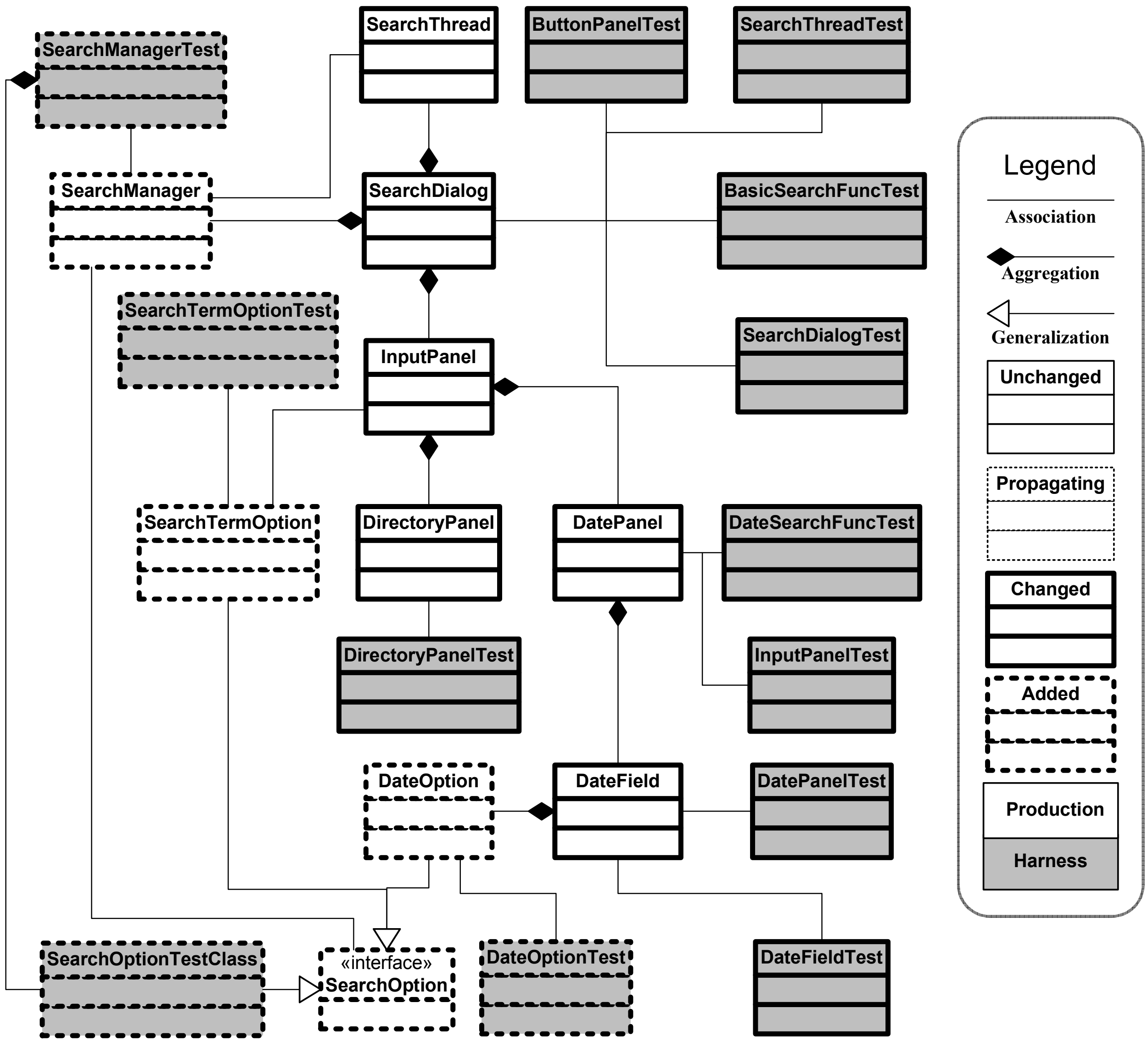


Figure A.35 Change 5 Prefactoring UML

A.5.4.1 SearchThread class

This class had the fields `searchDirectory` and `recusiveSearch` extracted to `SearchManager`. The field `searchTerm` was extracted to `SearchTermOption` and `dateSearch` was extracted to `DatePanel`. The method, `isDateInRange()` was moved to `DateOption`. The constructor now only receives 2 parameters of type `SearchDialog` and `SearchManager`.

The method, `recursiveSearch()` now just checks a file by calling `isFileValid()` in `SearchManager`, to see if it meets the search criteria. It also calls the methods `isRecursiveSearch()` and `getSearchDirectory()` in `SearchManager` to get the parameters removed from the `SearchThread` constructor. The method had the `String` parameter removed.

A.5.4.2 SearchDialog class

This class added a field of type `SearchManger`. It creates an instance of it in its constructor, passes it to `InputPanel` and `SearchThread` when it creates an instance of each. The method `searchCommand()` was merged with `searchStopButton()`, because it was now only 2 lines. This method merge could have been done during change 4, but was missed.

A.5.4.3 SearchManager class

This class was created; it manages the criteria for a search. It contains an `ArrayList` of `SearchOptions`, a `boolean isRecursive` and an `AbstractFile searchDirectory`. The last 2 were extracted from `SearchThread`.

There are methods, `addOption()`, `removeOption()` and `containsOptions()` to add and remove `SearchOption` objects from the `ArrayList`. The method `isFileValid()` is called by `SearchThread` to see if a file meets the searches criteria. This method iterates through the `ArrayList` and calls the `meetsCriteria()` of each `SearchOption`. If they all return true the method returns true; if one returns false, it returns false.

A.5.4.4 SearchOption interface

This interface needs to be implemented by classes that need to have criteria added to the search. It contains 1 method, `meetsCriteria()` that takes an `AbstractFile` as a parameter and should return true if the file meets the criteria and false if not.

A.5.4.5 DatePanel class

This class had the `JButton` fields that open a dialog calendar extracted to the `DateField` class and the parts of the `actionPerformed()` method that listened for them. The `createCalendarButton()` method was also moved to `DateField`. The 2 getters `getMinDate()` and `getMaxdate()` that called the `getDate()` method the appropriate `DateField` objects, were removed.

A.5.4.6 DateField class

This class added a field of type `DateOption`, which is initialized from a parameter passed to the constructor. The field of type `DateFormat` was only read once, so it was inlined. The class extended `JTextField`, but this was changed to `JPanel` and a field of type `JTextField` was added to the class.

The methods `createTextField()` was extracted from the constructor and it now initializes the field of type `JTextField` instead of the base type of the class. The method `createCalendarButton()` was moved from `DatePanel`. The method `setEnabled()` was overridden to enable and disable all the `Component` objects.

The `KeyListener` interface was changed for a `DocumentListener`. This made the code simpler; the `KeyListener` differentiates between different types of `KeyEvent` objects, while the `DocumentListener` differentiates between adding and removing text. The method `keyReleased()` from `KeyListener` had a workaround added to check if it was an event that added or removed text. Now with the `DocumentListener`, the code was divided between the `insertUpdate()` and `removeUpdate()` methods. This also allowed `null` checks to be removed from `setText()`. Finally, a call to `DateOption setDate()` was added to the `setDate()` method, so that the `DateOption` object would always have the most recently entered date.

A.5.4.7 DateOption class

This class implements the `SearchOption` interface. It has an abstract nested class and 2 nested classes that implement it. These classes all have 1 method, `compare()` which takes 2 longs as parameters. This was done so that the `meetsCriteria()` method could use polymorphism. The classes were nested because they are very small, 1 method with 1 LOC. This kept all the logic of the date search criteria in 1 file. This could be seen as a workaround for Java's lack of polymorphism at the method level.

One of the nested class's implementation returns true if the first parameter is greater and the second if the second parameter is greater. These classes allow the logic of the `meetsCriteria()` method from the `SearchOption` interface to be changed through polymorphism; this allows the same `DateOption` class to be used for both the minimum date and maximum date. The logic is set by a `boolean` parameter in the constructor.

A.5.4.8 InputPanel class

This class added the `ActionListener` interface; it listens to the `recursiveBox` field and calls the `setRecursive()` method in the `SearchManager`. Fields of type `SearchManager` and `SearchTermOption` were added. The `SearchTermOption` is added to the `SearchManager`'s list of search criteria by default in the constructor. It is never removed. The methods `createInputBox()`, `createLabelPanel()` and `createOptionsPanel()` were extracted from the constructor.

A.5.4.9 DirectoryPanel class

This class added a field of type `SearchManager`. It now updates the directory by calling `setSearchDirectory()` in `SearchManager`, from its constructor with the start directory and from `keyReleased()` when one is entered.

A.5.4.10 SearchTermOption class

This class implements the `SearchOption` interface; its `meetsCriteria()` method returns true if the search term is in any part of the file name regardless of case. It has 1 field of type `String` that stores the search term. It also implements a `DocumentListener` that listens to the document in the `JTextField` field in `InputPanel`. When the `Document` of the `JTextField` is updated, the `String` is updated.

A.5.4.11 SearchThreadTest class

This class is the unit test suite for the `SearchThread` class. It had its `setUp()` method modified and its `tearDown()` method, which was empty removed. All 6 of its tests were modified.

A.5.4.12 SearchDialogTest class

This class is the unit test suite for the `SearchDialog` class. It had 1 test and its `setUp()` method modified.

A.5.4.13 SearchManagerTest class

This class was added, it is the unit test suite for the `SearchManager` class; it has 9 tests.

A.5.4.14 DatePanelTest class

This class is the unit test suite for the `DatePanel` class. It had 1 test and its `setUp()` method modified and 3 tests added.

A.5.4.15 DateFieldTest class

This class is the unit test suite for the `DateField` class. It added a `setUpOneTime()` method had 2 tests and its `setUp()` method modified. One test was deleted and 2 added.

A.5.4.16 DateOptionTest class

This class was added, it is the unit test suite for the `DateOption` class; it has 5 tests.

A.5.4.17 InputPanelTest class

This class is the unit test suite for the `InputPanel` class. It had its `setUp()` method modified.

A.5.4.18 DirectoryPanelTest class

This class is the unit test suite for the `DirectoryPanel` class. It had its `setUp()` method modified and its `tearDown()` method, which was empty removed.

A.5.4.19 SearchTermOptionTest class

This class was added, it is the unit test suite for the `SearchTermOption` class; it has 3 tests.

A.5.4.20 ButtonPanelTest class

This class is the unit test suite for the `ButtonPanel` class. It had one test modified.

A.5.4.21BasicSearchFuncTest class

This class is a functional test suite. It had 4 tests modified.

A.5.4.22 DateSearchFuncTest class

This class is a functional test suite. It had 3 tests modified.

A.5.4.23 SearchOptionTestClass class

This class is an implementation of the `SearchOption` interface for use in tests. It has a constructor that sets a `boolean` field, which the `meetsCriteria()` method returns. There is no logic.

A.5.5 Actualization

The prefactoring prepared the code for the change very well. One class, `InputPanel` was modified and one class `CaseSensitiveOption` was added. `InputPanel` added a check box to turn case sensitive searching on and off. It does this by swapping its `SearchTermOption` field for the `CaseSensitiveOption` field. It also added a border around the recursive check box and the case sensitive check box in the GUI to organize it.

The added `CaseSensitiveOption` class is very similar to the `SearchTermOption` class, but it uses logic that includes the case of the search term and the file's name.

The total of each class by type of visit is listed in Table A.56. Table A.57 is a summary of the changes made to each class during actualization and the LOC added and deleted. Figure A.36 is a UML of actualization.

Table A.56 Change 5 Actualization Summary

Title	Code Files					
	Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
Case Sensitive Search	3	3	2	0	0	0

Table A.57 Change 5 Actualization Code Files

#	Code File	Task	Lines of Code		
			Added	Deleted	Total
1	`InputPanel`	Added, modified methods	53	5	58
2	`CaseSensitiveOption`	Added Class	37	0	37
3	`InputPanelTest`	Added, modified tests	68	3	71
4	`CaseSensitiveOptionTest`	Added class	53	0	53
5	`BasicSearchFuncTest`	Added tests	22	0	22

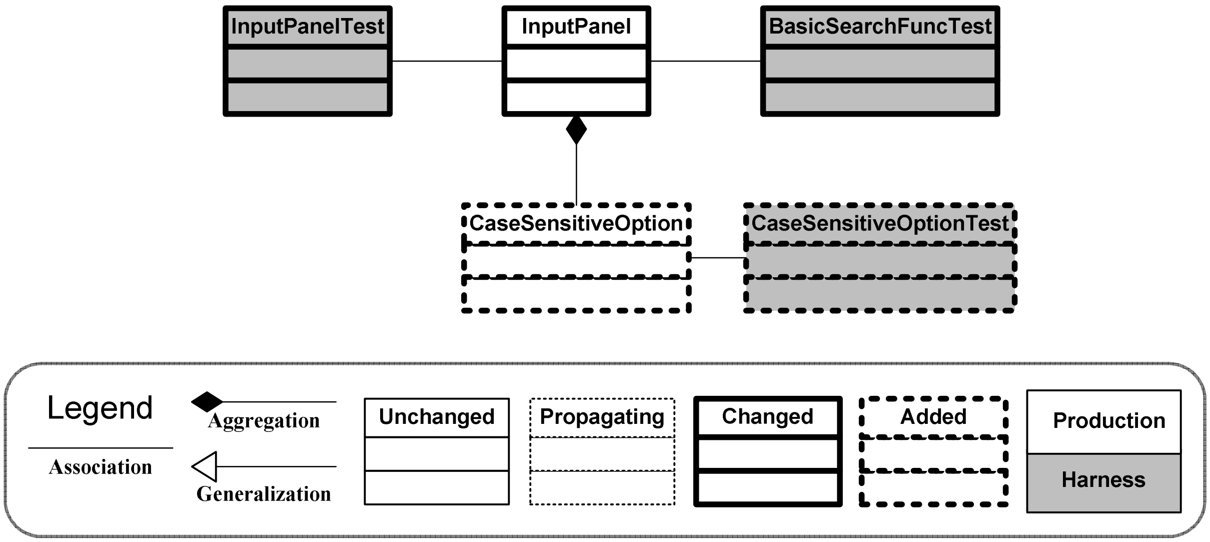


Figure A.36 Change 5 Actualization UML

A.5.5.1 InputPanel class

This class added fields of type `JCheckBox` and `CaseSensitiveOption`. The `JCheckBox` was added to the GUI in the `createOptionsPanel()` method. It along with the existing `JCheckBox` for recursive searches were both placed in their own `YBoxPanel` and a border was put around them.

The `CaseSensitiveOption` field is initialized in the constructor, but not added to the `SearchManager`. Logic was added in the `actionPerformed()` method to call a new `swapSearchTermOptions()` method that changes out the `SearchTermOption` for the `CaseSensitiveOption`. This causes the search to use the case sensitive logic. If the user unchecks the `JCheckBox`, the 2 will be swapped again.

A.5.5.2 CaseSensitiveOption class

This class implements the `SearchOption` interface; this allows it to be added to the `SearchManager`. It is very similar to `SearchTermOption`; its `meetsCriteria()` method returns true if the search term is in any part of the file name, but it includes case. It has 1 field of type `String` that stores the search term. It also implements a `DocumentListener` that listens to the `Document` in the `JTextField` field in `InputPanel`. When the `Document` of the `JTextField` is updated, the `String` is updated.

A.5.5.3 InputPanelTest class

This class is the unit test suite for the `InputPanel` class. It had its `setUp()` method modified, 3 test were added, 2 modified and 1 extracted from another.

A.5.5.4 CaseSensitiveOptionTest class

This class was added, it is the unit test suite for the `CaseSensitiveOption` class; it has 3 tests.

A.5.5.5 BasicSearchFuncTest class

This class is a functional test suite. It had 2 tests added.

A.5.6 Postfactoring

When the class `InputPanel` was extracted from `SearchDialog` during change 2, it held all the input fields. During the changes since then, `DirectoryPanel` was extracted from it and `DatePanel` was added to it. It now both holds other panels and creates panels. To clarify its responsibility, `BasicOptionsPanels` was extracted from it. `InputPanel` still creates a small panel that has 2 `JLabel` objects and an `AnimatedIcon`, because this panel has a mixture of `Component` objects that do not belong to any one group. The only other responsibility `InputPanel` has for this panel is to turn the `AnimatedIcon` on and off when a search starts or stops. This small responsibility does not belong to any of the supplier classes of `InputPanel`, so it was left there.

The classes `SearchTermOption` and `CaseSensitiveOption` had the same methods, but used a different logic in 3 of them. A super class was extracted from them; this also allowed them to be swapped more easily using their abstract class type. This super class extraction was necessary because of the change and could have been done during prefactoring to prepare for the change. This may have been slightly easier overall, but the change is the same in the end.

The total of each class by type of visit is listed in Table A.58. Table A.59 is a summary of the refactoring type and LOC added and deleted during postfactoring. Figure A.37 is a UML of postfactoring.

Table A.58 Change 5 Postfactoring Summary

Title	**Code Files**					
	Visited	**Changed**	**Added**	**Propagating**	**Unchanged**	**Added to Changed Set**
Case Sensitive Search	11	11	3	0	0	0

Table A.59 Change 5 Postfactoring Code Files

#	Code File	Task	Lines of Code		
			Added	Deleted	Total
1	`InputPanel`	Extracted class from	19	90	109
2	`BasicOptionsPanels`	Extracted class	97	0	97
3	`AbstractTermOption`	Extracted super class	30	0	30
4	`SearchTermOption`	Extracted super class from	2	24	26
5	`CaseSensitiveOption`	Extracted super class from	2	24	26
6	`SearchDialog`	Modified method	1	1	2
7	`InputPanelTest`	Modified, moved tests from	5	75	80
8	`BasicOptionsPanelsTest`	Added, moved tests	111	0	111
9	`SearchDialogTest`	Modified test	2	2	4
10	`SearchThreadTest`	Modified tests	5	6	11
11	`ButtonPanelTest`	Modified tests	1	1	2
12	`SearchFuncTestSetUp`	Modified method	4	3	7
13	`DateSearchFuncTest`	Modified tests, method	5	7	12
14	`BasicSearchFuncTest`	Modified tests	55	50	105

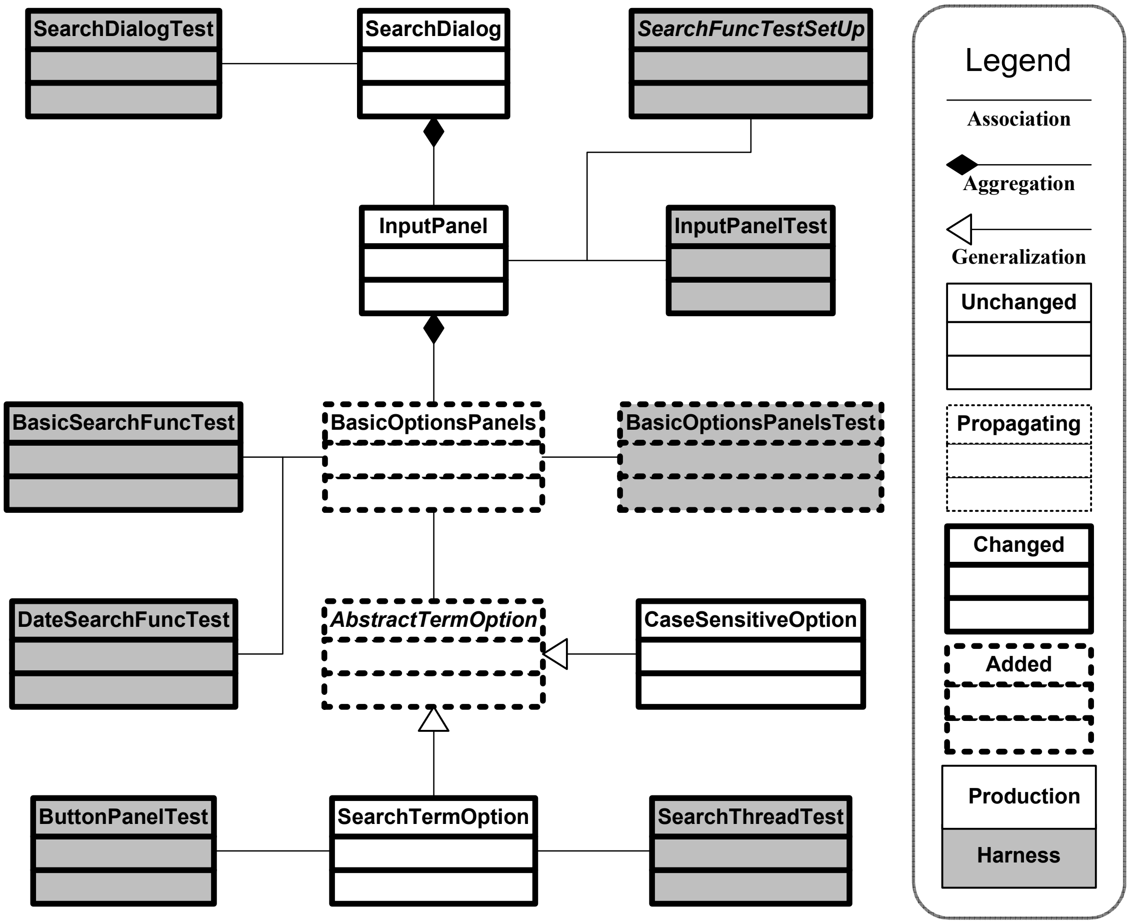


Figure A.37 Change 5 Postfactoring UML

A.5.6.1 InputPanel class

The class `BasicOptionsPanels` was extracted from this class. The extraction included the fields of type `JTextField` that holds the search term, the `JCheckBox` objects that turn the recursive and case sensitive search on and off, the `SearchTermOption`, `CaseSensitiveOption` and `SearchManager`. The methods `createInputBox()`, `swapSearchTermOptions()` and `actionPerformed()` were also extracted. A portion of `createOptionsPanel` that made a `YBoxPanel` was also extracted. Now this method just combines the `DatePanel` and a `YBoxPanel` from a call to `getBasicOptionsPanel()` in `BasicOptionsPanels`.

A.5.6.2 BasicOptionsPanels class

This class was extracted from `InputPanel`. It creates 2 `YBoxPanel` objects, 1 contains 2 `JCheckBox` objects, 1 `JCheckbox`, is listened to by the `actionPerformed()` method and calls the `setRecursive()` method in `SearchManager` when its selected. The other `JCheckBox` is also listened to by `actionPerformed()` and swaps the between array index zero and 1, when it is selected. This array is of type `AbstractTermOption` and contains objects of type `SearchTermOption` and `CaseSensitiveOption` objects.

The other `YBoxPanel` contains a `JLabel` and a `JTextField` that contains the search term. The `JTextField` is listened to by the `SearchTermOption` and `CaseSensitiveOption`. Since these fields all have an association, they were placed in the same class. However, they are not in the same `YBoxPanel` in the GUI, so there are 2 methods, `getInputFieldPanel()` and `getBasicOptionsPanel()` that return the `YBoxPanel` objects to be added in the appropriate place by `InputPanel`.

Finally, to make the swapping between the object at index 1 and 2 of the array of type `AbstractTermOption`, a nested `enum` was created. The values are `INSENSITIVE` and `SENSITIVE` and there is a method `getOpposite()` that returns the other value.

A.5.6.3 AbstractTermOption abstract class

This class was extracted from the `SearchTermOption` and `CaseSensitiveOption` classes. It contains the field of type `String` that holds the search term. The constructor and methods, `changedUpdate()`, `insertUpdate()` and `removeUpdate()` were also extracted. The method `setSearchTerm()` is different in each class, but needed to be referenced from a reference of `AbstractTermOption`, so it was added as an abstract method.

A.5.6.4 SearchTermOption class

This class had the `AbstractTermOption` super class extracted from it. It lost the field and methods described in `AbstractTermOption`.

A.5.6.5 CaseSensitiveOption class

This class had the `AbstractTermOption` super class extracted from it. It lost the field and methods described in `AbstractTermOption`.

A.5.6.6 SearchDialog class

A chained method call to get the parameter for `setInitialFocusComponent()` in the constructor had to add an extra call; because the `getInputBox()` method was extracted from `InputPanel` to `BasicSearchOptionsPanels`.

A5.6.7 InputPanelTest class

This class is the unit test suite for the `InputPanel` class. It had 5 tests moved to `BasicOptionsPanelsTest` and 3 modified.

A.5.6.8 BasicOptionsPanelsTest class

This class was added, it is the unit test suite for the `BasicOptionsPanels` class; it has 9 tests, 5 were moved from `InputPanelTest`.

A.5.6.9 SearchDialogTest class

This class is the unit test suite for the `SearchDialog` class. It had 1 test modified.

A.5.6.10 SearchThreadTest class

This class is the unit test suite for the `SearchThread` class. It had 5 tests modified.

A.5.6.11 ButtonPanelTest class

This class is the unit test suite for the `ButtonPanel` class. It had 1 test modified.

A.5.6.12 SearchFuncTestSetUp abstract class

This is a class that is extended by test classes that need a `SearchDialog` object for testing. It added a field of type `JCheckBox` and modified its `setUp()` method.

A.5.6.13 DateSearchFuncTest class

This class is a functional test suite. It had 2 tests and a test helper method modified.

A.5.6.14 BasicSearchFuncTest class

This class is a functional test suite. It had 11 tests modified.

A.5.7 Verification

After prefactoring and postfactoring all the regression tests passed. No new regression tests were added. All new tests passed; no bugs were identified in this change. Coverage for each production code file is available in Table A.60.

Table A.60 Change 5 Statement Verification

#	Code File	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	SearchDialog	44	43	97.7	0	0
2	SearchThread	25	21	84.0	0	0
3	SearchManager	17	17	100.0	0	0
4	DateField	69	64	92.8	0	0
5	BasicOptionsPanels	45	45	100.0	0	0
6	DatePanel	58	57	98.3	0	0
7	DirectoryPanel	53	44	83.0	0	0
8	InputPanel	36	36	100.0	0	0
9	SearchTermOption	4	4	100.0	0	0
10	DateOption	20	20	100.0	0	0
11	CaseSensitiveOption	4	4	100.0	0	0
12	AbstractTermOption	7	6	85.7	0	0

A.5.8 Timing Data

Table A.61 contains the timing data for the change request.

Table A.61 Change 5 Timing Totals

Phase	Time (hh:mm)
Concept Location	0:00
Impact Analysis	1:02
Prefactoring	9:32
Prefactoring Testing	2:53
Actualization	1:36
Actualization Testing	0:49
Postfactoring	2:35
Postfactoring Testing	1:19

A.5.9 Conclusions

This change had a large prefactoring, that directly impacted the size change set of actualization. It moved the concept location from a dual responsibility class to its own class. After the prefactoring, actualization was much simpler. It required 1 class to be modified and 1 to be created along with 2 test classes modified and 1 created. The prefactoring organized the criteria for a search; the logic for each criterion is now in its own class. It also meant that the class that contained the concept location did not need to be modified during actualization. In general, the impact set to add a criterion should be much smaller.

Additionally, because of the use of inheritance and polymorphism a search criterion is only added when it has been enabled. This will allow many different criteria options without slowing simple searches. Before the change, there was procedural checking to see if a criteria was enabled for each file checked; had this pattern continued, a search done with only a term would have had to check all the criteria for each file, even if the criteria was not enabled. This would have made for a slow search; now only the enabled criteria will be checked. The Strategy design pattern organizes the source code for future changes and should provide good performance even if a large number of search criteria are added.

One harness code file was in the estimated impact set called `SearchFuncTestSetUp` but was not changed during prefactoring or actualization. it was changed during postfactoring. Table A.62 lists the totals for each set of code files for each change of this iteration to date. Table A.63 is the current state of the product backlog. Figure A.38 to Figure A.41 are screen shots of muCommander showing the change.

Table A.62 Change 5 Code File Summary

#	Change	Number in Code Files						
		Visited Concept Location	Estimated Impact Set	Changed Set	Added during			Total Project
					Pre	Act	Post	
0	Original Baseline	N/A	N/A	N/A	N/A	N/A	N/A	1,070
1	Basic Search	5	3	4	0	4	0	1,074
2	Recursive search	0	3	4	4	0	5	1,083
3	Advanced Output	6	21	11	2	4	10	1,099
4	Date Search	0	13	12	2	16	3	1,120
5	Case Sensitive	0	16	15	8	2	3	1,133

Table A.63 Change 5 Current Product Backlog

#	Title	Complete	User Story
1	Basic Search	x	Add a basic search function that allows a user to search in the current directory for all or part of the title of a folder or file, and return a list of the matching files and directories.
2	Recursive Search	x	Add the ability to search inside all directories.
3	Advanced Output	x	Change the output to a table similar to the main muCommander window.
4	Date Search	x	Allow the user search by a date of file's modification.
5	Case Sensitive Search	x	Add capability to search by case sensitive search terms.
6	Extension Search		Add the ability to search for files with specific extensions.
7	Properties Search		Add options to search for files based on their properties.
8	Size Search		Add the ability to search for a file by its size.

9	Regular Expression Search	Add capability to search by a regular expression.
10	Lucene Search	Incorporate the Apache Lucene search.

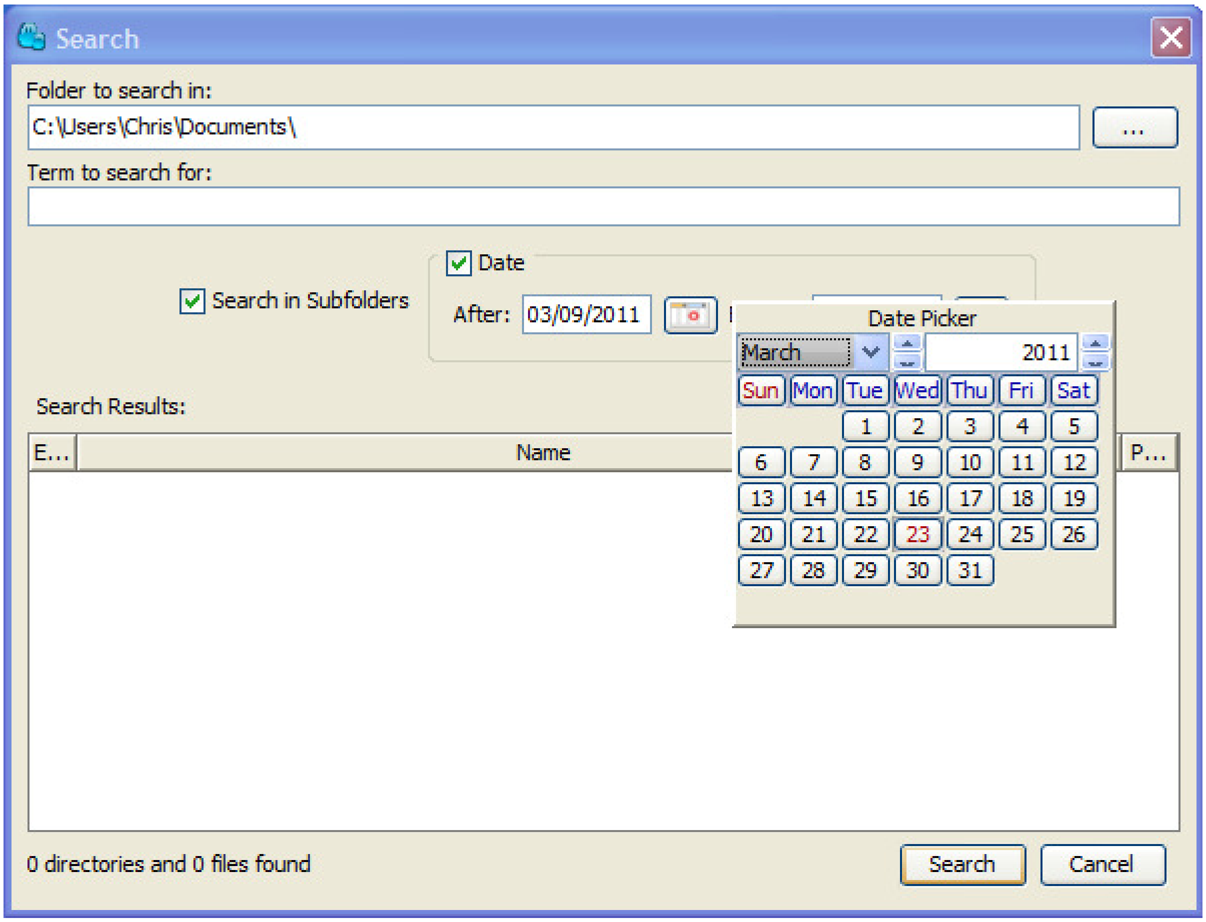


Figure A.38 Search window before Case Sensitive Change

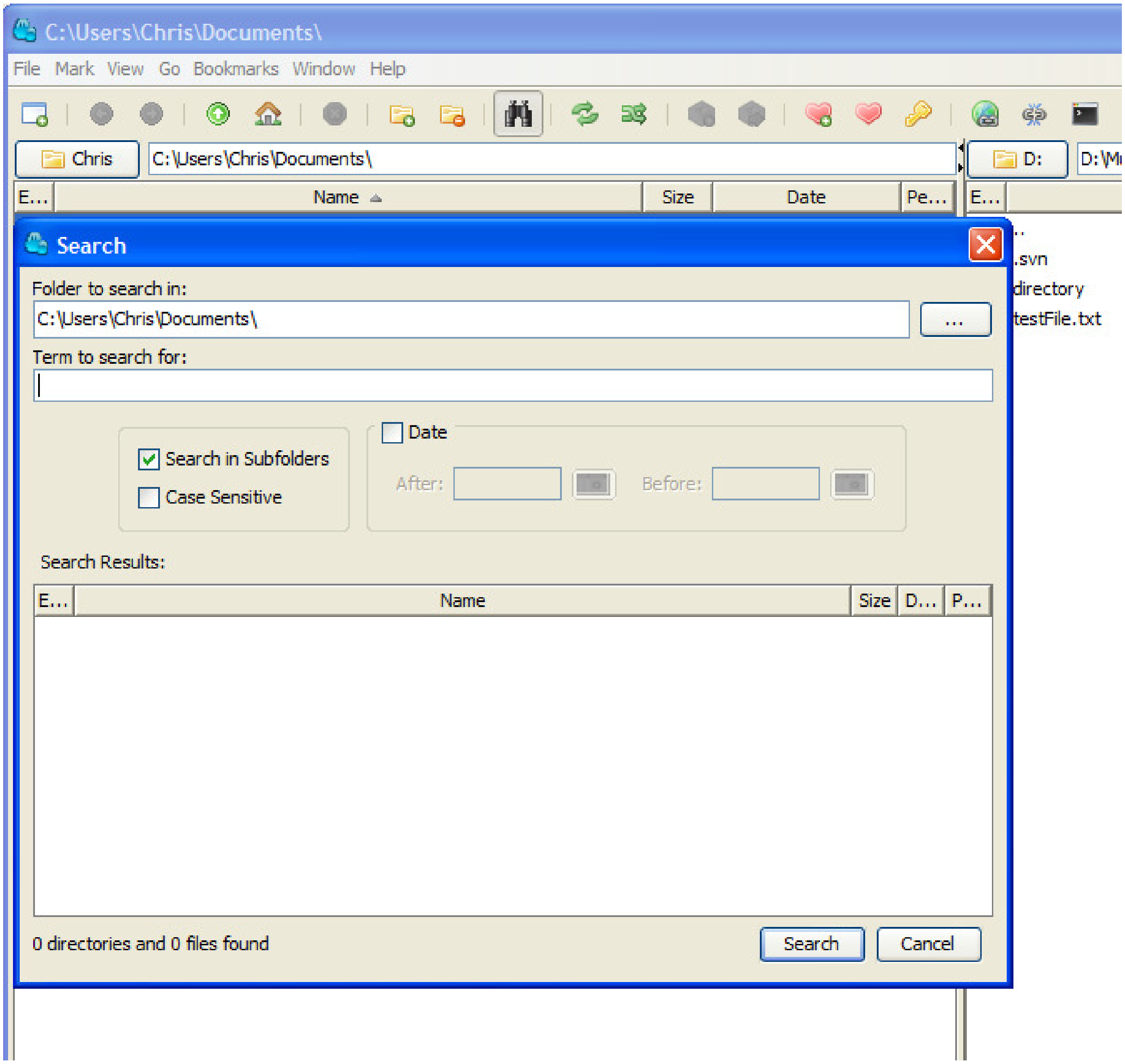


Figure A.39 Search window after Case Sensitive Change

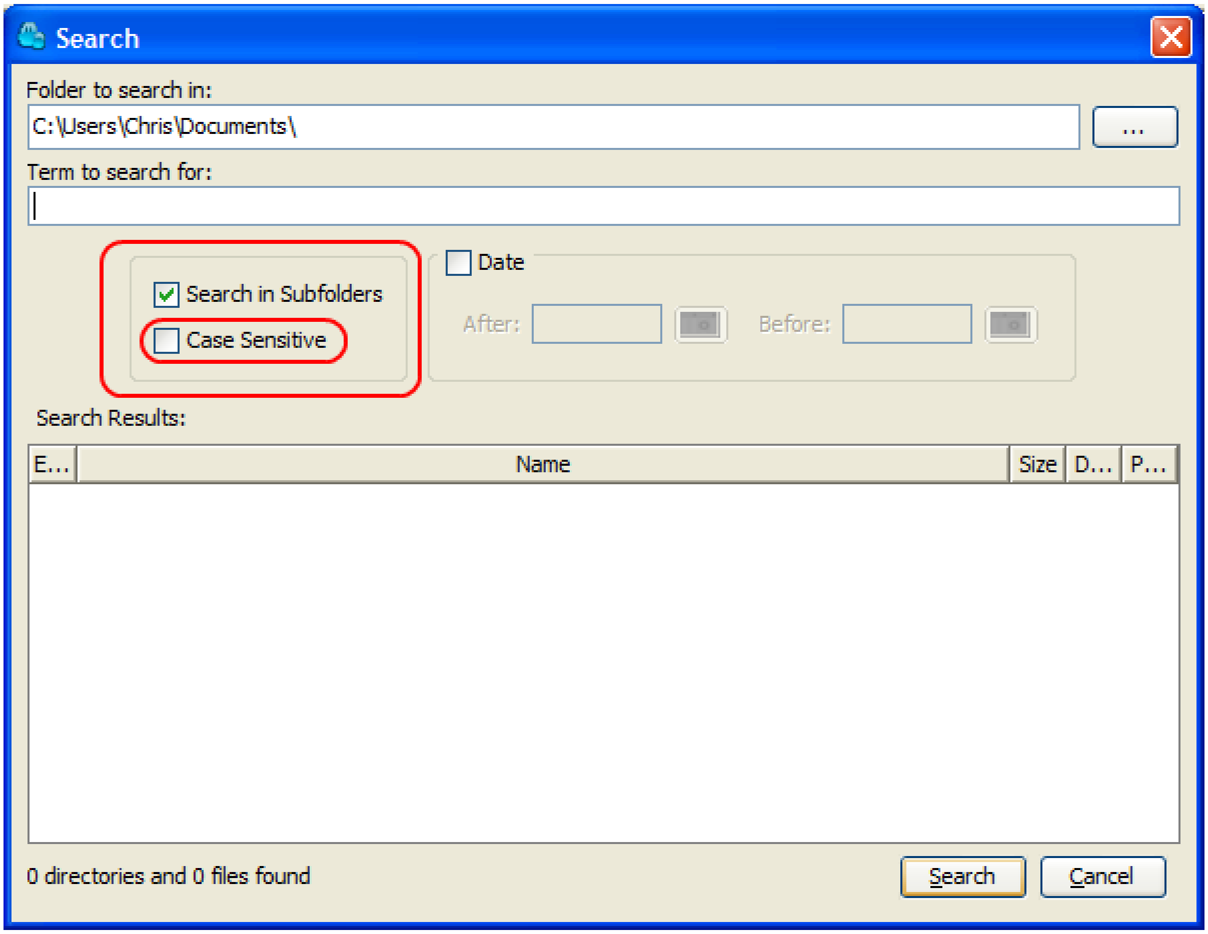


Figure A.40 Search window case sensitive search feature circled

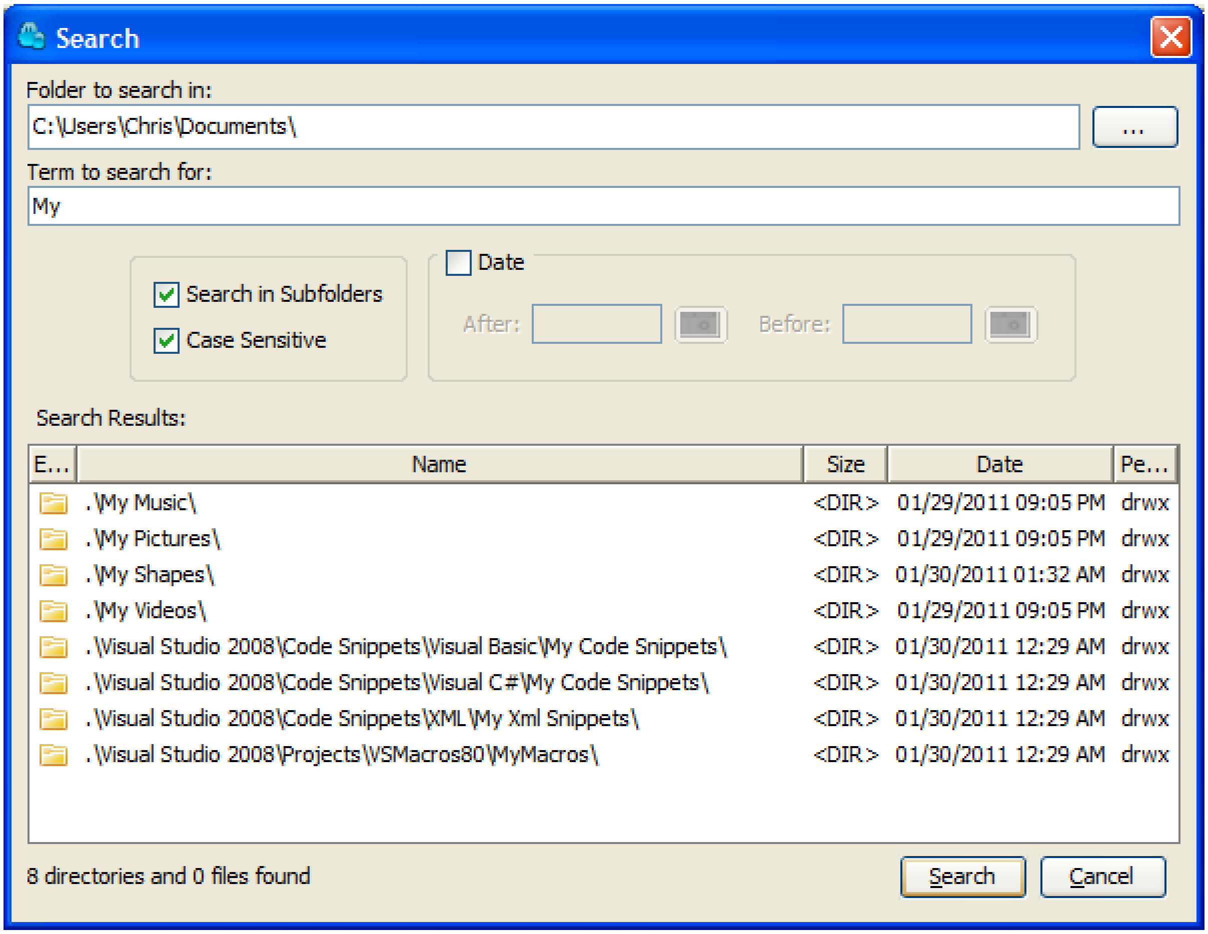


Figure A.41 Search window after a case sensitive search has finished

SIP – Change 6 Extension Search

A.6.1 Initialization

Add the ability to search for files with specific extensions to the search feature in muCommander. It is an application which enhances an operating system's file explorer. During the first 5 change requests, search capabilities were added which include:

- searching for a file whose name contains a certain term, both case sensitive and insensitive
- searching in any file system directory
- recursively searching in subfolders

- displaying results in a GUI table with the look and feel of the muCommander application
- searching within a specified date range

This change request will add the capability to search for files with a specific extension. A check box will be added to the GUI display that will allow the user to turn this capability on and off. A text box will also be added that will allow the user to enter one or more file extensions, separated by a semicolon, to search for.

Finally, when the extension search is enabled, the user entered search term will not be compared against the file's extension. This will give the search more capability. For example, if the search term is "txt" and the extension is "log", the search will only return results such as "Some txt file.log", but not all files with a txt extension.

A.6.2 Concept Location

No concept location was needed for this change. Based on experience obtained during previous changes the programmer knew the concept was located in the `BasicOptionsPanels` class which was created during change 5.

A.6.3 Impact Analysis

The programmer started impact analysis by marking the code file containing the concept location, `BasicOptionsPanels`, Impacted in JRipples; this marked 14 code files as Next. `AbstractTermOption` was visited and marked as Impacted because this change request will modify how a file's name is compared to the search term. For the same reason, the programmer marked `SearchTermOption` and `CaseSensitiveOption`, which inherit from `AbstractTermOption` as Impacted. The Next set now contained 15 code files. The programmer then visited `AbstractFile`; it contained methods `getFileNameWithoutExtension()` and `getExtension()`. These methods are all the change request requires from `AbstractFile`, so it was marked Unchanged.

The programmer then visited `InputPanel`; which was marked as Impacted because it contains the panel that errors are displayed in and this change request will need to display an error. The Next set of code files was now 22. `DatePanel` was then visited and marked as Propagating because the programmer will use the test field from the date picker added during change 4 in this change request. Following this path, the programmer marked `DateField` then `JCalendar` then `JYearChooser` as Propagating. Then `JSpinField` was visited and marked as Impacted because it only accepts integers, this change request would require it to also accept alphabetic characters. The Next set created by JRipples was now 35 code files. The programmer then visited the other code files that are related to the date picker and their test classes, `JDayChooser`, `JMonthChooser`, `JCalendarFuncTest`, `JCalendarTest`, `JMonthChooserTest`, `JSpinFieldTest` and `JYearChooserTest`. All were marked Unchanged; except `JSpinFieldTest`, which will need to be changed with `JSpinField`. The Next set was now 28 code files.

The programmer then visited and marked the test classes `BasicOptionsPanelsTest`, `CaseSensitiveOptionTest` and `SearchTermOptionTest` as Impacted; these will need to change to test the new functionality in the classes they are directed at. The Next set was now 26 code files. The programmer visited the 15 production code files in the Next set and marked them Unchanged. The harness code files were then visited, 10 were marked Unchanged;

TestConstants was marked Impacted because new AbstractFile objects would be added to test the extension search. This added 7 code files to the Next set. The programmer visited these and marked them Unchanged to end impact analysis. Table A.64 shows the code file totals for impact analysis and Table A.65 lists each code file visited. Figure A.42 is a UML of visited code files.

Table A.64 Change 6 Impact Analysis Summary

Title	Code Files					Comments
	Visited	Impacted	Propagating	Unchanged	Not Visited	
Extension Search	54	11	4	39	0	

Table A.65 Change 6 Impact Analysis Code Files Visited

#	Code File	Tool used	Impacted?	Comments
1	BasicOptionsPanels	JRipples → Impacted	Impacted	Concept Location
2	AbstractTermOption	JRipples → Impacted	Impacted	File name comparison will change
3	SearchTermOption	JRipples → Impacted	Impacted	File name comparison will change
4	CaseSensitiveOption	JRipples → Impacted	Impacted	File name comparison will change
5	AbstractFile	JRipples → Unchanged	Unchanged	Has needed methods
6	InputPanel	JRipples → Impacted	Impacted	Contains error panel
7	DatePanel	JRipples → Propagating	Propagating	Propagates to JSpinField
8	DateField	JRipples → Propagating	Propagating	Propagates to JSpinField
9	JCalendar	JRipples →	Propagating	Propagates to

		Propagating		JSpinField
10	JYearChooser	JRipples → Propagating	Propagating	Propagates to JSpinField
11	JSpinField	JRipples → Impacted	Impacted	Contains field that changes color on invalid input
12	JDayChooser	JRipples → Unchanged	Unchanged	
13	JMonthChooser	JRipples → Unchanged	Unchanged	
14	JCalendarFuncTest	JRipples → Unchanged	Unchanged	
15	JCalendarTest	JRipples → Unchanged	Unchanged	
16	JMonthChooserTest	JRipples → Unchanged	Unchanged	
17	JSpinFieldTest	JRipples → Impacted	Impacted	Code file test directed at Impacted
18	JYearChooserTest	JRipples → Unchanged	Unchanged	
19	BasicOptionsPanelsTest	JRipples → Impacted	Impacted	Code file test directed at Impacted
20	CaseSensitiveOptionTest	JRipples → Impacted	Impacted	Code file test directed at Impacted
21	SearchTermOptionTest	JRipples → Impacted	Impacted	Code file test directed at Impacted
22	ComponentTitledBorder	JRipples → Unchanged	Unchanged	
23	CustomDateFormat	JRipples → Unchanged	Unchanged	
24	DateOption	JRipples → Unchanged	Unchanged	

25	DirectoryPanel	JRipples → Unchanged	Unchanged	
26	ErrorManager	JRipples → Unchanged	Unchanged	
27	FlashLabel	JRipples → Unchanged	Unchanged	
28	IconManager	JRipples → Unchanged	Unchanged	
29	SearchDialog	JRipples → Unchanged	Unchanged	
30	SearchManager	JRipples → Unchanged	Unchanged	
31	SearchOption	JRipples → Unchanged	Unchanged	
32	SearchTable	JRipples → Unchanged	Unchanged	
33	SearchTableModel	JRipples → Unchanged	Unchanged	
34	SpinningDial	JRipples → Unchanged	Unchanged	
35	Translator	JRipples → Unchanged	Unchanged	
36	YBoxPanel	JRipples → Unchanged	Unchanged	
37	BasicSearchFuncTest	JRipples → Unchanged	Unchanged	
38	ButtonPanelTest	JRipples → Unchanged	Unchanged	
39	DateFieldTest	JRipples → Unchanged	Unchanged	
40	DatePanelTest	JRipples → Unchanged	Unchanged	
41	DateSearchFuncTest	JRipples → Unchanged	Unchanged	
42	InputPanelTest	JRipples → Unchanged	Unchanged	

43	`SearchDialogTest`	JRipples → Unchanged	Unchanged	
44	`SearchFuncTestSetUp`	JRipples → Unchanged	Unchanged	
45	`SearchTableModelTest`	JRipples → Unchanged	Unchanged	
46	`SearchThreadTest`	JRipples → Unchanged	Unchanged	
47	`TestConstants`	JRipples → Impacted	Impacted	Need to add fields
48	`DateOptionTest`	JRipples → Unchanged	Unchanged	
49	`DirectoryPanelTest`	JRipples → Unchanged	Unchanged	
50	`SearchManager`	JRipples → Unchanged	Unchanged	
51	`SearchTableTest`	JRipples → Unchanged	Unchanged	
52	`FileFactory`	JRipples → Unchanged	Unchanged	
53	`SearchTableCellRendererTest`	JRipples → Unchanged	Unchanged	
54	`ResultsPanelTest`	JRipples → Unchanged	Unchanged	

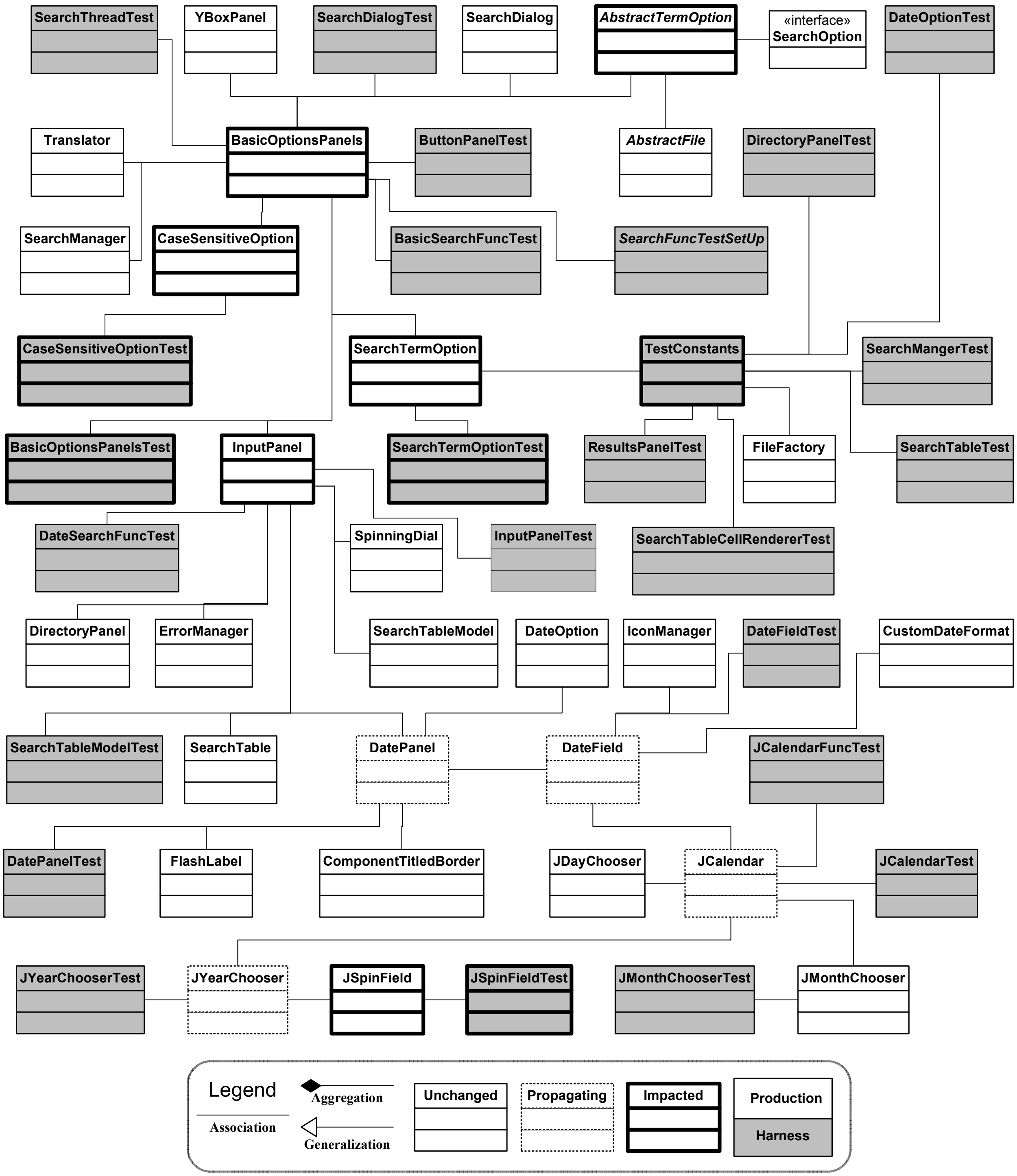


Figure A.42 Change 6 Impact Analysis UML

A.6.4 Prefactoring

The programmer added the class `JSpinField` as part of the date chooser that opens when the user clicks on a calendar icon. This field colors the text green if the user input is valid and red if the user input is invalid as the user types. However, the JSpinField only accepts integer values. To make it easier to add the coloring feature for alphabetical values to this change request, a new class, FeedbackField was extracted from JSpinField. It extends JTextField and is only responsible for changing the color of the text, depending if it is valid or invalid. To make FeedbackField work in general cases; the programmer added a nested interface, InputListener. InputListener has 1 method, isInputValid() that allows implementing classes to define what is valid and invalid input.

This refactoring removed responsibility from JSpinField, but it did not significantly change the size of JSpinField, 54 LOC were deleted, but 46 were added to JSpinField. JSpinField's JTextField was replaced with FeedbackField and the CaretListener interface was replaced with InputListener. However, the code file FeedbackField is 97 LOC, so the production code was increased by 89 LOC. This is because to give FeedbackField sufficient generality to be used multiple cases, it has 3 constructors, 12 getters and setters for its colors and 3 new methods for its interface. If this feature had not been desired for use in another class, it would not have been necessary to do this refactoring.

A test class FeedbackField was extracted from JSpinFieldTest. It also had tests added for the new methods. Table A.66 shows the code file visited and Table A.67 summarizes the changes to each code file. Figure A.43 is a UML of the code files visited.

Table A.66 Change 6 Prefactoring Summary

Title	Code Files					
	Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
Extension Search	2	2	2	0	0	0

Table A.67 Change 6 Prefactoring Code Files

#	Code File	Task	Lines of Code		
			Added	Deleted	Total
1	`JSpinField`	Extracted class from	46	54	100
2	`FeedbackField`	Extracted class	97	0	97
3	`JSpinFieldTest`	Extracted test class from	2	13	15
4	`FeedbackFieldTest`	Extracted test class	132	0	132

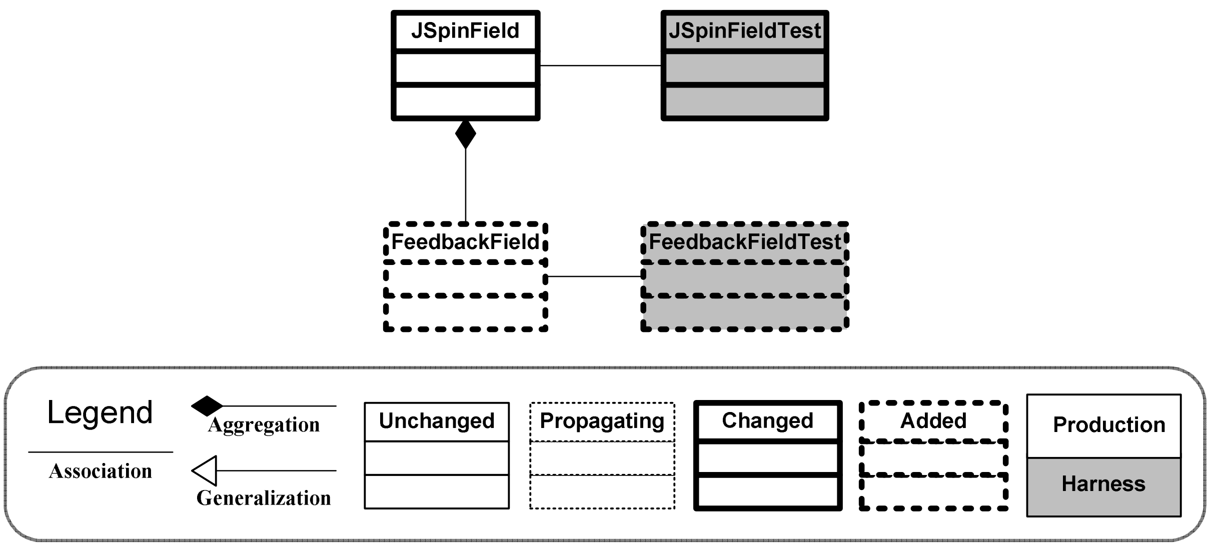


Figure A.43 Change 6 Prefactoring UML

A.6.4.1 JSpinField class

The programmer extracted `FeedbackField` from this class. The field of type `Color` was extracted. The field of type `JTextField` was changed to type `FeedbackField` and its name was changed from `textField` to `feedbackField`. The renaming modified the constructor and methods `setValue()`, `setMaximum()`, setHorizontalAlignment(), setFont(), setForeground(), setEnabled(), actionPerformed() and getTextField().

The constructor and the method setValue() had their responsibility for color moved to FeedbackField. The interface CaretListener and its method caretUpdate() were also extracted to FeedbackField. The interface InputListener and its method isValidInput() were added. The method listens to input in the FeedbackField and returns true if it is valid. It also updates an int field if the input is valid.

The programmer deleted the main() method that is unneeded, but was missed in previous refactoring.

A.6.4.2 FeedbackField code file

The programmer extracted the Feedback class from JSpinField. It extends JTextField and adds responsibility to color the text inside the JTextField a valid color or invalid color depending on input. It also has a default color for when it is not in focus. There is a constructor with these colors as parameters and getters and setters that allow them to be customized.

The interface CaretListener and its method caretUpdate() were extracted from JSpinField. The method calls a new method checkValidUpdate() and sets the color to valid if it returns true, invalid if false.

To allow classes that create an object of this class to define what is valid and invalid text, it has a nested interface InputListener, with 1 method, isValidInput() that should return true if the input is true. The instantiating class can add or remove itself as a listener through the addInputListener() and `removeInputListener()` methods. These methods add or remove the listener from a field of type `HashSet`. The method `checkValidUpdate()` iterates through the listeners in the `HashSet` and calls their `isValidInput()` method; if any returns false, it returns false, if all return true, it returns true.

A.6.4.3 JSpinFieldTest class

This is the test class for the `JSpinField` class. The programmer extracted the `FeedbackFieldTest` class from this test class. The extraction included the test, `testCaretUpdate()`. One test was modified.

A.6.4.4 FeedbackFieldTest class

This is the test class for the `FeedbackField` code file. The programmer extracted it from `JSpinFieldTest`. One test, `testCaretUpdate()` was extracted and 14 tests were added.

A.6.5 Actualization

To actualize the change request, the programmer created a new class that extends `YBoxPanel` called `ExtensionPanel`. The class contains a `JCheckBox`, `FeedbackField` and `FlashLabel`. It is a supplier to `BasicOptionsPanels` and was incorporated as a component. This class adds the components to the GUI for the user to enter extensions.

The programmer also added a class that implements the `SearchOption` interface, `ExtensionOption` that is added to the list of `SearchOption` objects in the `SearchManager` when an extension search is enabled. `ExtensionOption`'s primary responsibility is to check an `AbstractFile`'s extension against the set of user entered extensions and return true if it is.

The programmer added the responsibility of changing between classes that extend `AbstractTermOption` to compare an `AbstractFile`'s name to a search term to `BasicOptionsPanels`. When an extension search is enabled, `BasicOptionsPanels` will change between 4 different implementations of the `AbstractTermOption` class. There were 2 classes to do this at the beginning of this change request, which compare the search term to the file's name including the extension. The programmer created 2 new classes that compare the file's name without the extension to the search term, `SearchTermWithoutExtensionOption` and `CaseSensitiveWithoutExtensionOption` that extend `AbstractTermOption`. Additionally, the programmer added a `FocusListener` to `FeedbackField` to change the text color to the default when the field has lost focus.

The test classes, `ExtensionSearchFuncTest`, `ExtensionOptionTest` and `ExtensionPanelTest` were added by the programmer. `FeedbackFieldTest` and `BasicOptionsPanelsTest` were changed. Two new files to be used with the extension tests were added, testFile.log and testFile.test that are the same as testFile.txt added in change 2, but with different extensions. Final `AbstractFiles` corresponding to these files were added to the class `TestConstants`. Table A.68 shows the code files visited and Table A.69 lists the code files changed. Figure A.44 is a UML of code files visited.

Table A.68 Change 6 Actualization Summary

Title	Code Files					
	Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
Extension Search	6	6	7	0	0	0

Table A.69 Change 6 Actualization Code Files

#	Code File	Task	Lines of Code		
			Added	Deleted	Total
1	ExtensionPanel	Added class	88	0	88
2	BasicOptionsPanels	Changed methods	58	17	75
3	ExtensionOption	Added class	34	0	34
4	SearchTermWithoutExtensionOption	Added class	14	0	14
5	CaseSensitiveWithoutExtensionOption	Added class	14	0	14
6	FeedbackField	Added method	14	3	17
7	InputPanel	Changed methods	3	2	5
8	ExtensionPanelTest	Added test class	71	0	71
9	BasicOptionsPanelsTest	Changed method, tests	16	11	27
10	ExtensionOptionTest	Added test class	27	0	27
11	FeedbackFieldTest	Added methods	11	2	13
12	ExtensionSearchFuncTest	Added test class	103	0	103
13	TestConstants	Added fields	4	0	4

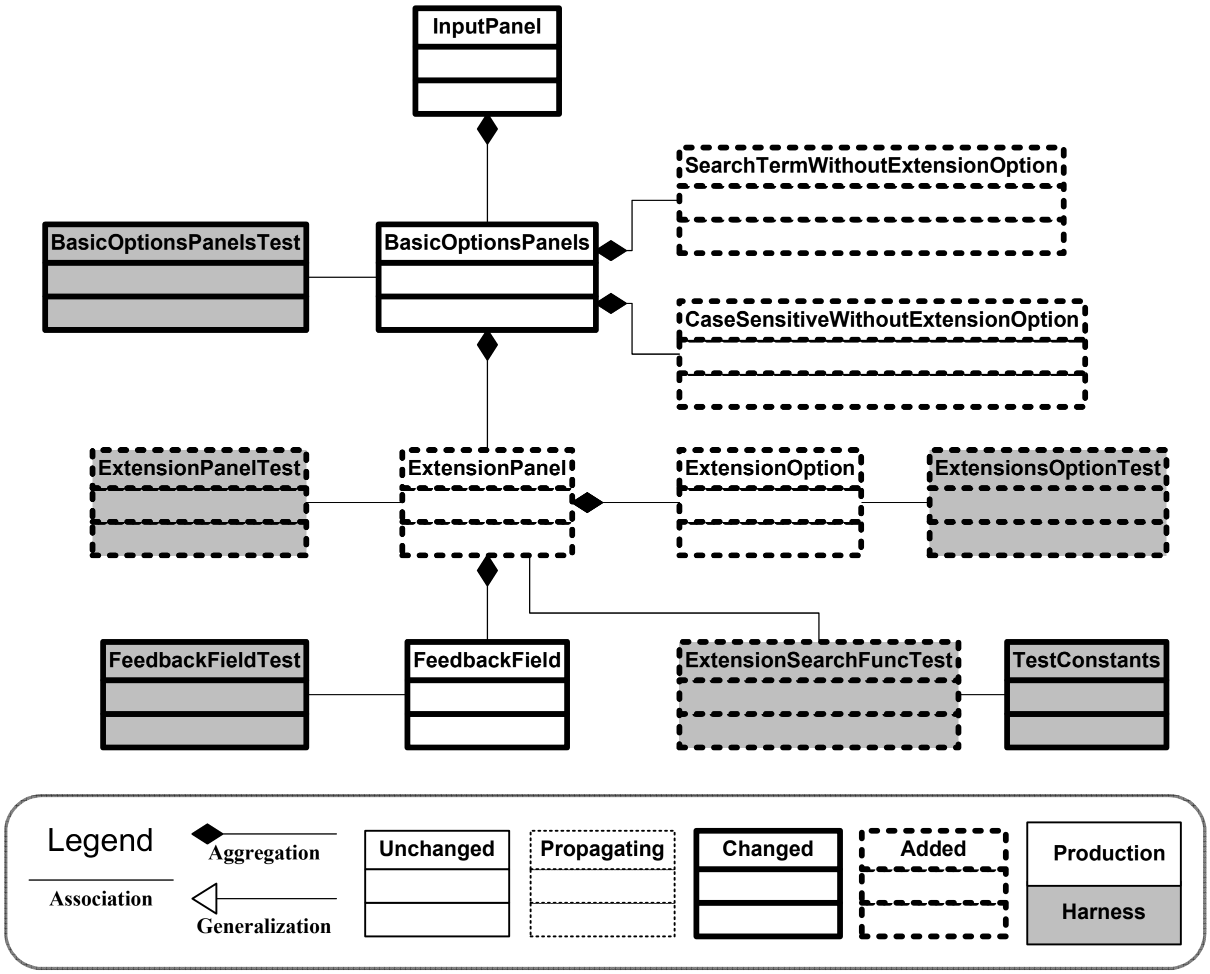


Figure A.44 Change 6 Actualization UML

A.6.5.1 ExtensionPanel class

The programmer added this class to the project. It has fields of type `JCheckBox`, `FeedbackField`, `FlashLabel`, `SearchManager`, `ErrorManager`, `ExtensionOption`, `BasicOptionsPanel`, `Pattern` and a static final `String`. The `JCheckBox` and `FeedbackField` get the user input. The `FlashLabel` displays errors to the user when added to the `ErrorManager`. The `ExtensionOption` is added to the `SearchManager` when the extension search is enabled. `BasicOptionsPanels` is a client of this class, one of its methods is called when the extension search is modified.

The Pattern and String are used to check if the user has input any characters into the FeedbackField that are invalid in a file extension.

The class implements the InputListener interface. The isValidInput() method uses the Pattern field to check the text entered by the user into the FeedbackField is valid. It also adds the FlashLabel to the ErrorManager, if the input is invalid.

The class also implements the ActionListener interface. The actionPerformed() method listens for the JCheckBox. It enables the FeedbackField and adds the ExtensionOption to the SearchManager. It also calls the method swapSearchTerms() in basicOptionsPanels.

A.6.5.2 BasicOptionsPanels class

The programmer added a field of type ExtensionPanel to this class. The method getBasicOptionsPanel() return type was changed to a JPanel. A temporary variable of type JPanel was added and the ExtensionPanel along with the YBoxPanel already created in the method, then the JPanel is returned.

Another field of type AbstractTermOption was also added. The method swapSearchTermOptions() was changed. It had taken a parameter of type Case; it then removed the opposite AbstractTermOption of the parameter from the SearchManager and added the AbstractTermOption corresponding to the Case. This would no longer work, because now there are 4 AbstractTermOption objects and the caller of this method may not know which AbstractTermOption to switch to. The parameter was changed to a boolean type. If set to true it will change to the AbstractTermOption that is used with an extension search; if false it switches between the case sensitivity `AbstractTermOption` objects. Since the Swing libraries are not thread safe, the modifier synchronized was added to the method.

The array field of type `AbstractTermOption` was expanded from size 2 to 4. The nested `enum`, `Case` added 2 values `INSENSITIVE_WO_EXT` and `SENSITIVE_WO_EXT` along with a method `switchExtension()` that returns the `Case` value with the same case sensitivity, but opposite extension concept. The `getOpposite()` method was changed to add the 2 new values.

A.6.5.3 ExtensionOption class

The programmer added this class to handle the responsibility of checking if an `AbstractFile`'s extension matches any of the search criteria extensions. It has 1 array field of type `String` that holds the search extensions. It implements the `SearchOption` interface; the method `meetsCriteria()` from the interface gets an `AbstractFile`'s extension and compares it to each of the extensions in the array of extensions; if any of the extensions match it returns true.

The `getExtensions()` method returns the array of `String` extensions, but it also initializes the array if it is `null` so it never returns `null`. The `setExtensions()` methods takes a single `String` and parses it into an array and assigns it to the array field of `String` objects.

A.6.5.4 SearchTermWithoutExtensionOption class

The programmer added this class to enable extension searches to not compare an `AbstractFile`'s extension with the search term. It extends `AbstractTermOption`. Its `meetsCriteria()` method returns true if the `AbstractFile`'s name without the extension contains the search term, ignoring case.

A.6.5.5 CaseSensitiveWithoutExtensionOption class

The programmer added this class to enable extension searches to not compare an `AbstractFile`'s extension with the search term, but include case. It extends `AbstractTermOption`. Its `meetsCriteria()` method returns true if the `AbstractFile`'s name without the extension contains the search term, including case.

A.6.5.6 FeedbackField code file

The programmer added the `FocusListener` interface to this code file. The interface's `focusLost()` method changes the fields text color to the default color if the current color is valid. Also, the default color is only initialized to black if a `null` color is passed to the constructor.

A.6.5.7 InputPanel class

This class had to add its `ErrorManager` object to the `BasicOptionsPanels` object creation call. It also adds the `FlashLabel` that displays an extension error to the same location as the date error.

A.6.5.8 ExtensionPanelTest class

This class was added, it is the unit test suite for the `ExtensionPanel` class; it has 5 tests.

A.6.5.9 BasicOptionsPanelsTest class

This class is the unit test suite for the `BasicOptionsPanels` class. It had its `setUp()` method and 5 tests changed.

A.6.5.10 ExtensionsOptionTest class

This class was added, it is the unit test suite for the `ExtensionOption` class; it has 2 tests.

A.6.5.11 FeedbackFieldTest class

This class is the unit test suite for the `FeedbackField` class. It had 2 tests changed and 1 added.

A.6.5.12 ExtensionSearchFuncTest class

This class is a functional test suite for extension searches. It extends `SearchFuncTestSetUp` and 6 has tests.

A.6.5.13 TestConstants class

This class holds public static final fields used by the unit and functional tests. It added 2 fields of type `AbstractFile` corresponding to 2 new files added to the project with log and test extensions.

A.6.6 Postfactoring

After actualization the change request functionality worked, but the method in `BasicOptionsPanels` that switched between the 4 classes that extend `AbstractTermOption` was confusing and would be difficult to change in the future. The responsibility to listen to 1 `JCheckBox` and switch between 2 classes had grown and was spread across 2 classes, `BasicOptionsPanels` and `ExtensionPanel`. Further, 2 of these classes created during actualization, `SearchTermWithoutExtensionOption` and `CaseSensitiveWithoutExtensionOption`, had long and confusing names and very similar responsibility. The programmer decided that instead of having 4 different `AbstractTermOption` objects, there should be 1 class that listens to the 2 `JCheckBox` objects and uses polymorphism to switch between the compare criteria. The programmer decided to simplify this responsibility and combine it into 1 code file, SearchTermOption. The super class and 3 other AbstractTermOption classes would all be merged into it. Additionally, ActionListener objects would be extracted from BasicOptionsPanels and ExtensionPanel to this code file.

The programmer changed the ExtensionOption's method, setExtensions(), which parses the user entered String into an array of String extensions, to a regular expression algorithm. The rest of the refactoring was renaming fields in FeedbackField and updating Javadoc in TestConstants. Table A.70 shows the code files visited and Table A.71 lists the changed code files. Figure A.45 is a UML of code files visited.

Table A.70 Change 6 Postfactoring Summary

Title	Code Files					
	Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
Extension Search	17	12	(5)	0	0	0

Table A.71 Change 6 Postfactoring Code Files

#	Code File	Task	Lines of Code		
			Added	Deleted	Total
1	SearchTermOption	Merged classes to, added interfaces, classes, methods	104	6	110
2	AbstractTermOption	Merged class	0	30	30
3	CaseSensitiveOption	Merged class	0	15	15
4	SearchTermWithoutExtensionOption	Merged class	0	14	14
5	CaseSensitiveWithoutExtensionOption	Merged class	0	14	14
6	ExtensionOption	Changed methods	12	7	19

7	`BasicOptionsPanels`	Extracted, moved method	8	62	70
8	`ExtensionPanel`	Extracted method	18	18	36
9	`FeedbackField`	Renamed field	10	10	20
10	`SearchTermOptionTest`	Merged class to, added, changed method, added tests	44	1	45
11	`CaseSensitiveOptionTest`	Merged class	0	53	53
12	`ExtensionsOptionTest`	Added method, added, changed tests	45	3	48
13	`BasicOptionsPanelsTest`	Added, changed tests	37	59	96
14	`ExtensionPanelTest`	Changed method, tests	17	9	26
15	`FeedbackFieldTest`	Changed tests	7	7	14
16	`ExtensionSearchFuncTest`	Changed method, tests	24	18	42
17	`TestConstants`	Javadoc	0	0	0

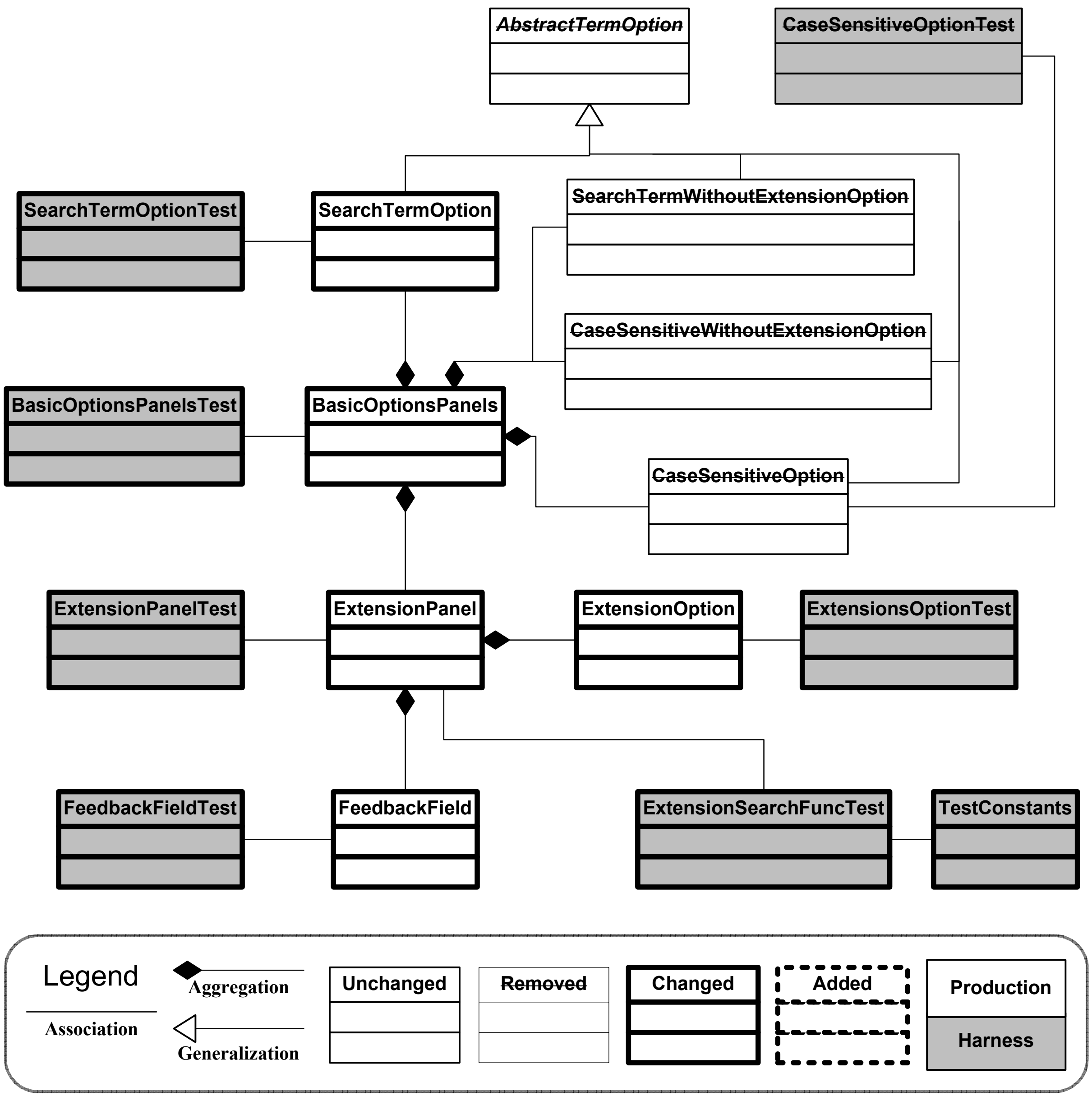


Figure A.45 Change 6 Postfactoring UML

A.6.6.1 SearchTermOption code file

The programmer merged the `AbstractTermListener` super class with this class. This added a field of type `String` and the 3 `DocumentListener` methods `changedUpdate()`, `insertUpdate()` and `removeUpdate()`.

The programmer also merged the responsibility from the classes `CaseSensitiveOption`, `SearchTermWithoutExtensionOption` and

CaseSensitiveWithoutExtensionOption into this code file. This was done by adding 2 nested interfaces, FileNameChooser and CaseSensitiveChooser, with 2 nested classes for each interface.

The FileNameChooser interface is implemented by the nested classes FileNameWithoutExt and FileNameWithExt. Both of these classes have a single method, chooseFileName(), which takes a parameter of type AbstractFile and returns its name as a String. The difference is that the method in FileNameWithoutExt returns the name without the extension and FileNameWithExt returns the name with the extension.

The CaseSensitiveChooser interface is implemented by the nested classes CaseSensitive and CaseInsensitive. Both of these classes have a single method, chooseCase() that takes a String as a parameter and returns a String. The difference is that the CaseInsensitive implementation converts the String to lower case before returning it, while the CaseSensitive implementation just returns the original String.

The nested classes are used by the meetsCriteria() method from the SearchOption interface. The FileNameChooser method chooseFilename() is passed the AbstractFile to get the appropriate file name. Then the name is passed to the CaseSensitiveChooser method chooseCase() that returns the name as a String in the proper case. That String is compared to the search term String and meetsCriteria() finally returns true, if the search term is contained in the String. The CaseSensitiveChooser method chooseCase() also is used by the setSearchTerm() method to set the search term to the proper case for the search.

The ActionListener for the case sensitive JCheckBox was extracted from BasicOptionsPanels. The portion of the actionPerformed() method that listens for the case sensitive JCheckBox was extracted from the method with the same name. It now calls a new method setCaseSensitve(), which switches between the classes that implement the CaseSensitiveChooser.

The ActionListener for the extension JCheckBox was extracted from the ExtensionPanel class. The portion of the actionPerformed() method that listens for the extension JCheckBox was extracted from the method with the same name. It now calls a new method setFileNameChooser(), which switches between the classes that implement the FileNameChooser.

This would appear to make this code file large and have diverse responsibility; however after the change request the code file has 112 LOC as measured by Clover. Its responsibility is also clear, to compare the search term to a file's name.

A.6.6.2 Deleted classes

The AbstractTermOption abstract class, CaseSensitiveOption, SearchTermWithoutExtensionOption and CaseSensitiveWithoutExtensionOption classes all were merged with SearchTermOption and removed from the project.

A.6.6.3 ExtensionOption class

The programmer changed the setExtensions() method. The method parses a String into a String array of extensions. The parsing removes leading white space, semicolons, periods and commas. This was done with a loop that used 4 calls to the String startsWith() method. This was replaced with a regular expression algorithm. To do this 2 fields, one of type `String` containing the characters and one of type `Pattern` were added to the class.

The method `meetsCriteria()` was changed so that a `null` check of its parameter of type `AbstractFile` is done first.

A.6.6.4 BasicOptionsPanels class

The programmer extracted the responsibility of switching between the different search term search options from this class to `SearchTermOption`. The array field of type `AbstractTermOption` was deleted along with the nested `enum Case` and the field of the same type. The `swapSearchTerms()` method also extracted to `SearchTermOption` along with the portion of `actionPerformed()` that listened to the case sensitive `JCheckBox`.

A new field of type `SearchTermOption` was added. It was added as a `DocumentListener` to the field of type `JTextField` that the user enters a search term in and as an `ActionListener` to the case sensitive `JCheckBox` field.

A.6.6.5 ExtensionPanel class

The programmer extracted the portion of the `actionPerformed()` method that listens to the extension `JCheckBox` field and called `swapSearchTermOptions()` in `BasicOptionsPanels` to `SearchTermOption`. This required the `BasicOptionsPanels` parameter in the constructor to be replaced with a parameter of type `SearchTermOption`. The object received from this parameter, was added to the extension `JCheckBox` as an `ActionListener`.

A.6.6.6 FeedbackField code file

The programmer renamed the field of type `HashSet` that contains the `InputListeners` from update to listeners to better describe what it holds. The method `checkValidUpdate()` was also renamed to `checkInputListeners()`.

A.6.6.7 SearchTermOptionTest class

This class is the unit test suite for the `SearchTermOption` class. It added a `setUpBeforeClass()` method, had its `setUp()` method changed and added 2 tests.

A.6.6.8 CaseSensitiveOptionTest class

This class is unit test suite for the `CaseSensitiveOption` class. Since the `CaseSensitiveOption` class was merged with the `SearchTermOption` class, this test class was removed from the project.

A.6.6.9 ExtensionOptionTest class

This is the unit test suite for the `ExtensionOption` class. It added a `setUpBeforeClass()` method, 2 tests were changed and 4 tests were added.

A.6.6.10 BasicOptionsPanelsTest class

This is the unit test suite for the `BasicOptionsPanels` class. It had a field renamed, 7 tests were changed and 2 tests were added.

A.6.6.11 ExtensionPanelTest class

This class is the unit test suite for the `ExtensionPanel` class. It had its `setUp()` method changed and 4 tests were changed.

A.6.6.12 FeedbackFieldTest class

This class is the unit test suite for the `FeedbackField` class. It had 5 tests changed.

A.6.6.13 ExtensionSearchFuncTest class

This class is a functional test suite for extension searches. It had its `setUp()` method changed and 7 tests were changed.

A.6.6.14 TestConstants class

This class holds public static final fields used by the unit and functional tests. It had its Javadoc updated.

A.6.7 Verification

The test suite exposed 3 bugs during the change request, a forth bug was discovered through code inspection. Two of these bugs were part of the current change request and were fixed; the other 2 were added to the backlog.

After prefactoring all the regression tests passed. During postfactoring 1 test, `testSetMonth()` from `JDayChooserTest`, failed. The programmer investigated this further and discovered the test will fail if run on the last day of any month if the next month has fewer days. The programmer did a test through user intervention and found that the bug did not affect the program's functionality. Therefore, a priority 4, minor problem not involving primary functionality, change request was added to the backlog to fix this bug. No new regression tests were added.

During impact analysis the programmer visited the `DatePanel` class; during this visit the programmer realized that the `datePanelSetEnabled()` method did not remove the `DateOption` object from the `SearchManager`. This means that if a date is entered and the date `JCheckBox` is unchecked, a date search will still be performed. This is the opposite of what a user would expect, but a there is an easy workaround;

just delete the date. This bug was given a priority 3, some functionality is impaired, but a workaround can be found, therefore a change request was added to the backlog.

While writing the test class for the `SearchTermOption` code file during postfactoring, the programmer found a bug in the `insertUpdate()` method. The bug was found by running `testInsertUpdate()` from the `SearchTermOptionTest` class. An exception was thrown by `insertUpdate()` if an empty `String` was input in the `Document` it listens to. This was resolved by adding a check for an empty `String`.

The programmer found a second bug in `SearchTermOption`, with the test, `testActionPerformedCaseSensitiveBox()` from the `SearchTermOptionTest` class. If a case sensitive search was enabled, disabled and enabled, without changing the search term, the case of the search term would be lost. The programmer added a field to `SearchTermOption` to fix the bug. The new field stores the term with case, so the case can be recovered when switching between case sensitive searches. Coverage for each production code file is available in Table A.72.

Table A.72 Change 6 Statement Verification

#	Code File	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	`FeedbackField`	42	42	100.0	0	0
2	`BasicOptionsPanels`	38	38	100.0	0	0
3	`ExtensionPanel`	36	36	100.0	0	0
4	`InputPanel`	37	37	100.0	0	0
5	`JSpinField`	61	51	83.6	0	0
6	`SearchTermOption`	38	37	97.4	0	2
7	`ExtensionOption`	20	20	100.0	0	0

A.6.8 Timing Data

Table A.73 contains the timing data for the change request.

Table A.73 Change 6 Timing Totals

Phase	Time (hh:mm)
Concept Location	0:00
Impact Analysis	0:55
Prefactoring	3:06
Prefactoring Testing	0:55
Actualization	2:20
Actualization Testing	2:36
Postfactoring	3:18
Postfactoring Testing	2:08

A.6.9 Conclusions

Prefactoring extracted 1 production code file, `FeedbackField` and made it much more useful for general use by other classes. This made it simpler to use in this change request, which extended the look and feel of a previous change into this change request.

The actualization was more difficult for the programmer. The design used by `BasicOptionsPanels` to switch between 2 classes that extend `AbstractTermOption` was difficult to extend to 4 classes that extend `AbstractTermOption` without bugs. This was not apparent to the programmer at the beginning of the change request otherwise he would have refactored these classes during prefactoring. Because of this difficulty the programmer knew he would delete the 2 new classes that extend `AbstractTermOption` during postfactoring, therefore he did not write a test class for these classes. The classes were also very simple, so there was not a large concern of bugs in the classes themselves. During postfactoring, the functionality was tested by new tests added to the `SearchTermOptionTest` class.

The strategy pattern [42] used to add and remove search criteria worked well. The programmer believes using this pattern has greatly reduced the changed set from the procedural pattern that was in `SearchThread` until change 5.

The changed set was 5 code files less than the estimated impact set. The 5 code files were changed during postfactoring. The change was complex and the programmer found it easier to allow code smells to develop during actualization and address them in postfactoring. Table A.74 lists the totals for each set of code files for each change request of this iteration to date. Table A.75 is the current state of the product backlog. Figure A.46 to Figure A.51 are screen shots of muCommander showing the change request functionality.

Table A.74 Change 6 Code File Summary

#	Change	Number in Code Files						
		Visited Concept Location	Estimated Impact Set	Changed Set	Added during			Total Project
					Pre	Act	Post	
0	Original Baseline	N/A	N/A	N/A	N/A	N/A	N/A	1,070
1	Basic Search	5	3	4	0	4	0	1,074
2	Recursive search	0	3	4	4	0	5	1,083
3	Advanced Output	6	21	11	2	4	10	1,099
4	Date Search	0	13	12	2	16	3	1,120
5	Case Sensitive	0	16	15	8	2	3	1,133
6	Extension Search	0	11	6	2	7	(5)	1,137

Table A.75 Change 6 Current Product Backlog

#	Title	Complete	User Story
1	Basic Search	x	Add a basic search function that allows a user to search in the current directory for all or part of the title of a folder or file, and return a list of the matching files and directories.
2	Recursive Search	x	Add the ability to search inside all directories.
3	Advanced Output	x	Change the output to a table similar to the main muCommander window.
4	Date Search	x	Allow the user search by a date of file's modification.
5	Case Sensitive Search	x	Add capability to search by case sensitive search terms.
6	Extension Search	x	Add the ability to search for files with specific extensions.
7	Properties Search		Add options to search for files based on their properties.
8	Date Bug		DateOption is not removed when disabled.
9	Size Search		Add the ability to search for a file by its size.
10	Regular Expression Search		Add capability to search by a regular expression.
11	Lucene Search		Incorporate the Apache Lucene search.
12	JDayChooserTest Bug		The test testSetMonth() fails on last day of month, if next month has fewer days

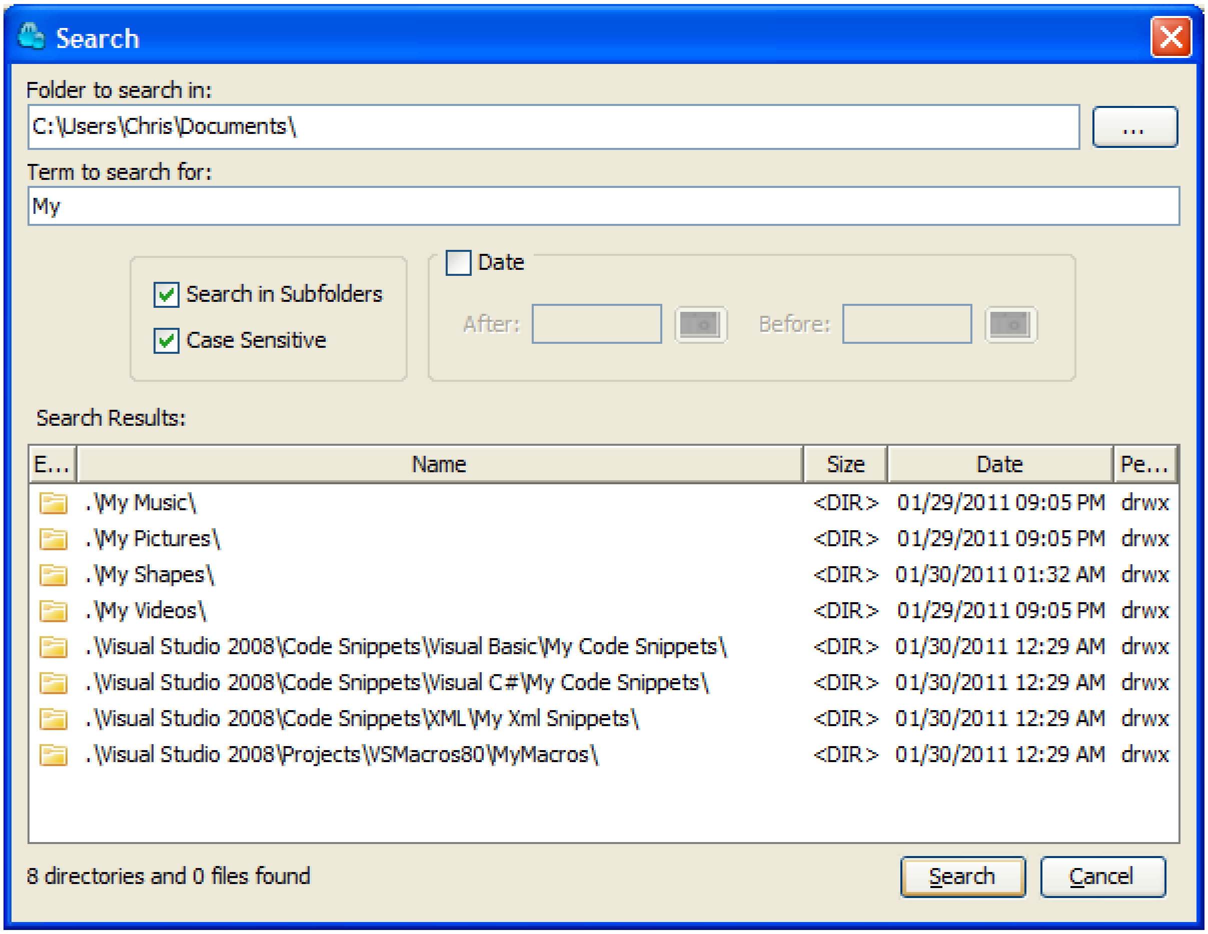


Figure A.46 Search window before the Extension Search Change

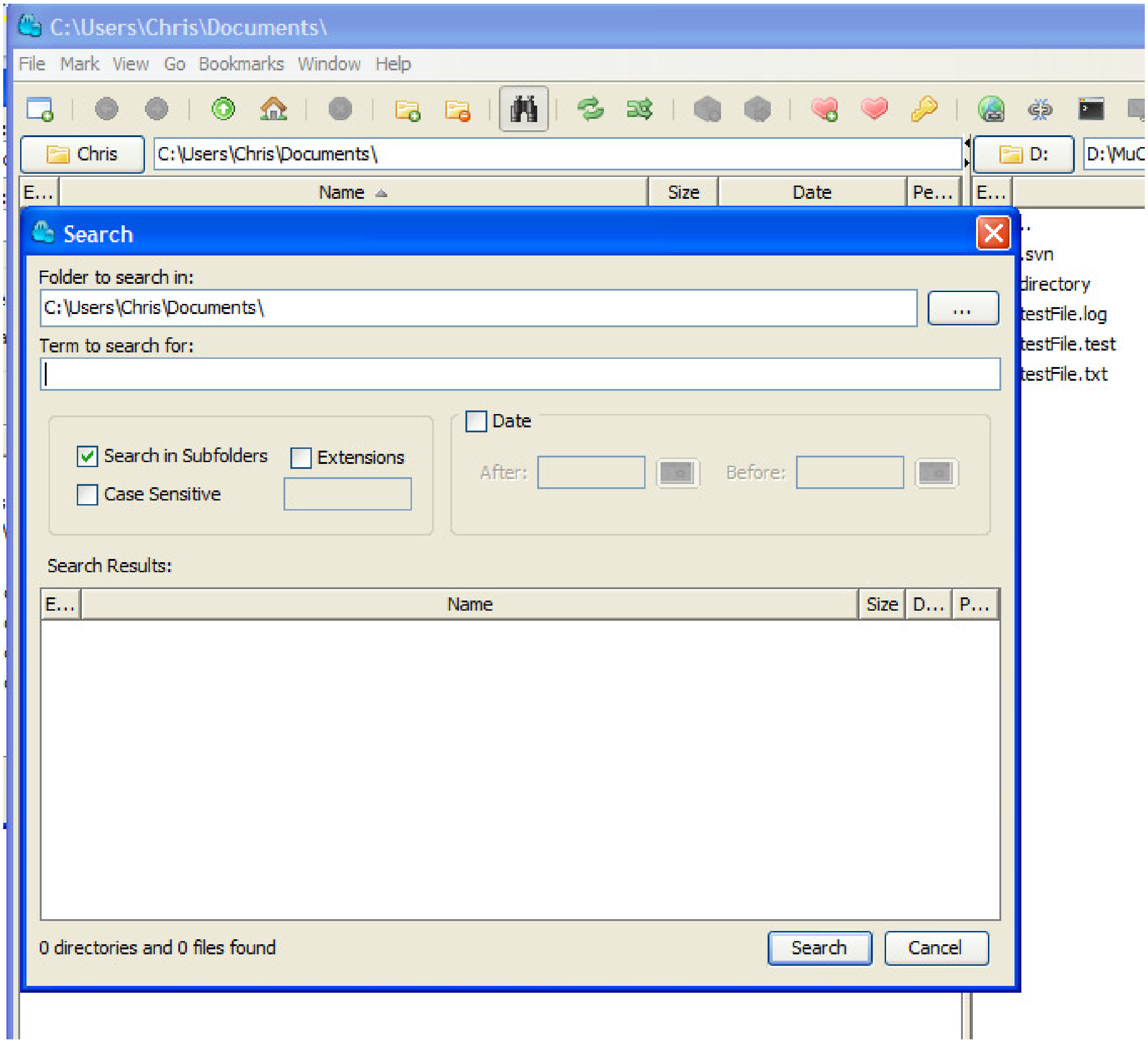


Figure A.47 Search window after Extension Search Change

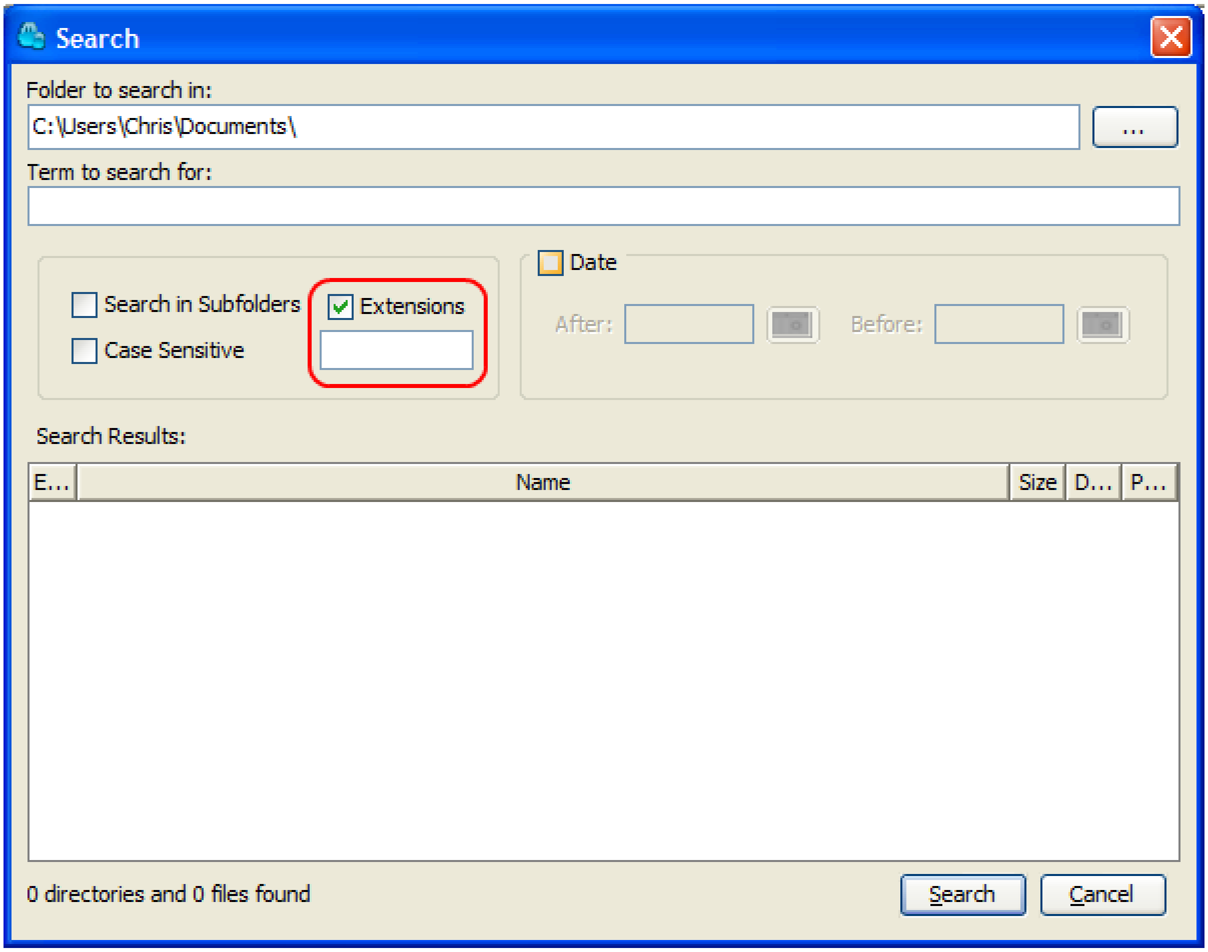


Figure A.48 Search window Extension Search Feature circled

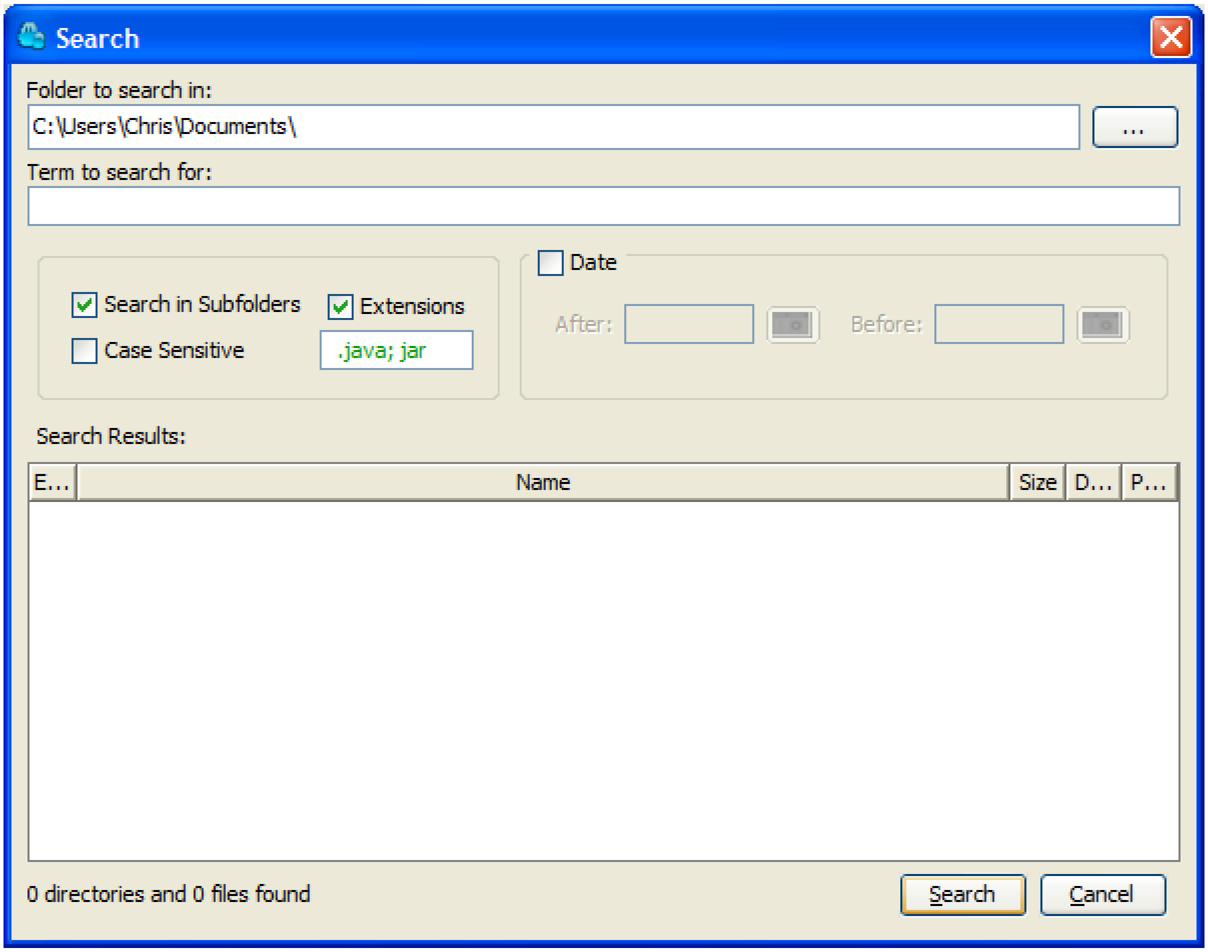


Figure A.49 Search window valid text in extension field

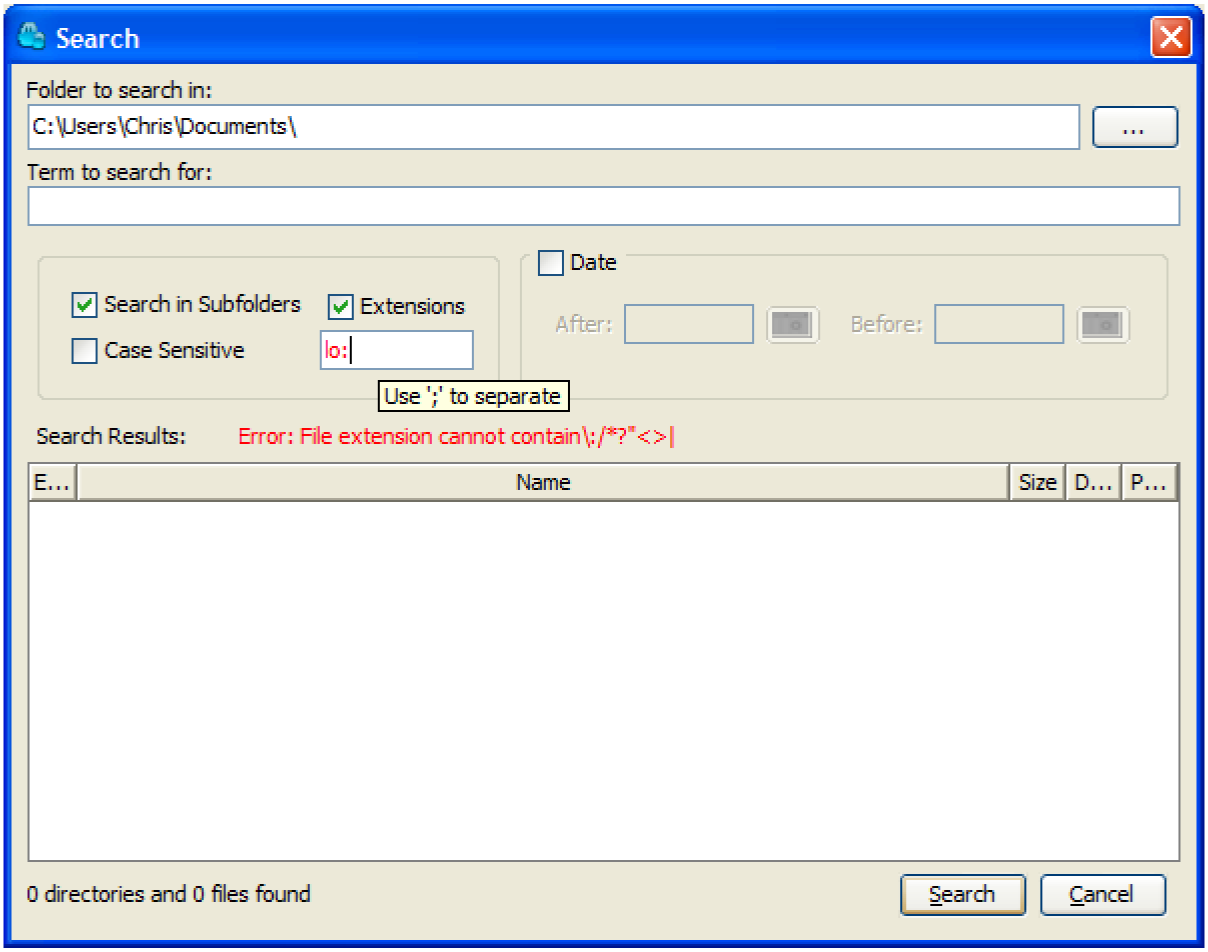


Figure A.50 Search window invalid text in extension field

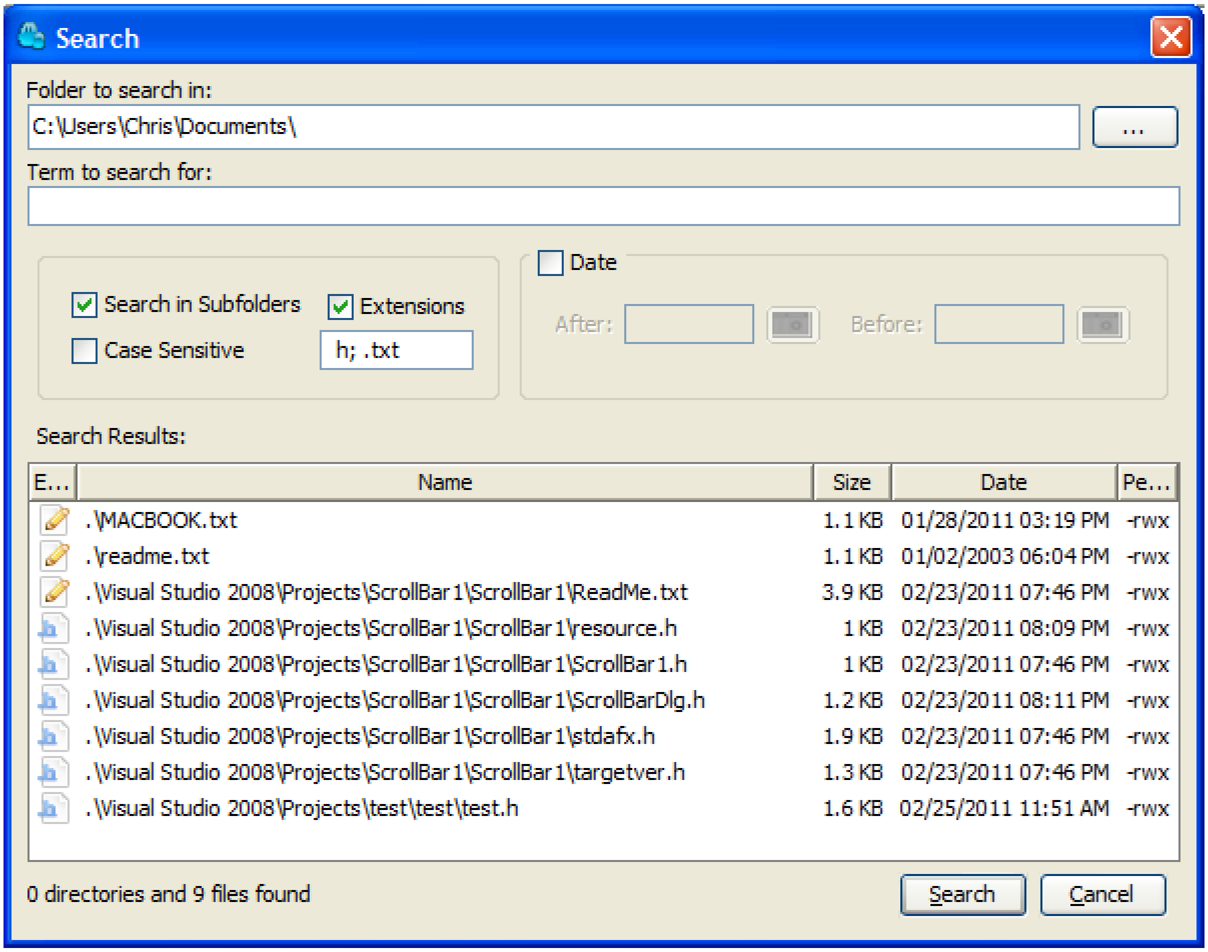


Figure A.51 Search window Extension Search Change

SIP – Change 7 Properties Search

A.7.1 Initialization

Add options to search for files based on their properties. The program, muCommander, is an application which enhances an operating system's file explorer. During the first 6 change requests, search capabilities were added; which include:

- searching for a file whose name contains a certain term, both case sensitive and insensitive,
- searching in any file system directory
- recursively searching in subfolders

- displaying results in a GUI table with the look and feel of the muCommander application
- searching for files with a certain extension
- searching for files modified within a specified date range

This change request will add the capability to search for files with specific properties. Four check boxes will be added to the GUI display that will allow the user to select which properties to search for. The properties to add are: archive file, directory, hidden file and read-only file. When one of the check boxes is selected a search will only return results of that type. If 2 or more boxes are selected, the file must meet all of the criteria; for example, if hidden file and read-only file are both selected, the results of the search will only include files that are both hidden and read-only. Since a file cannot be both an archive and a directory, if one of these properties is selected the other will be disabled.

A.7.2 Concept Location

No concept location was needed for this change. Based on experience obtained during previous changes the programmer knew the concept was located in the `BasicOptionsPanels` class which was created during change 5.

A.7.3 Impact Analysis

The programmer started impact analysis by marking the code file containing the concept location, `BasicOptionsPanels`, Impacted in JRipples; this marked 17 code files as Next. One of the Next set, `InputPanel` was visited and marked as Impacted. It has the object of `BasicOptionsPanels` and one of its methods, `createOptionsPanel()` will need to be changed. JRipples added 10 code files to the

Next set. The programmer then visited `AbstractFile`. The change requires that it has methods to check all of the properties being added. It did not have a method to check if an object of it is read-only, therefore it was marked Impacted. JRipples added 307 code files to the Next set for a total of 332.

The programmer then visited harness files `BasicOptionsPanelsTest`, `InputPanelTest`, `AbstractFileTest` and `TestConstants` marked them all Next. JRipples added their neighbors to the Next set, which now contained 329 code files.

This programmer decided not to visit the remaining set of Next classes. Most of the program is dependent on `AbstractFile`. The method the programmer planned to add to this class is a non-abstract `boolean` getter this should not affect any implementing or dependent class. Table A.76 show the total of each type of code file during impact analysis. Table A.77 is a summary of the code files visited during impact analysis. Figure A.52 is a UML diagram of impact analysis.

Table A.76 Change 7 Impact Analysis Summary

Title	**Code Files**					**Comments**
	Visited	**Impacted**	**Propagating**	**Unchanged**	**Not Visited**	
Properties Search	7	7	0	0	329	

Table A.77 Change 7 Impact Analysis Code Files Visited

#	Code File	Tool used	Impacted?	Comments
1	`BasicOptionsPanels`	JRipples → Impacted	Impacted	Concept Location
2	`InputPanel`	JRipples → Impacted	Impacted	Will need to change to accommodate new features
3	`AbstractFile`	JRipples → Impacted	Impacted	Needs new `boolean` getter method
4	`BasicOptionsPanelsTest`	JRipples → Impacted	Impacted	
5	`InputPanelTest`	JRipples → Impacted	Impacted	
6	`AbstractFileTest`	JRipples → Impacted	Impacted	
7	`TestConstants`	JRipples → Impacted	Impacted	Will need new test AbstractFile objects

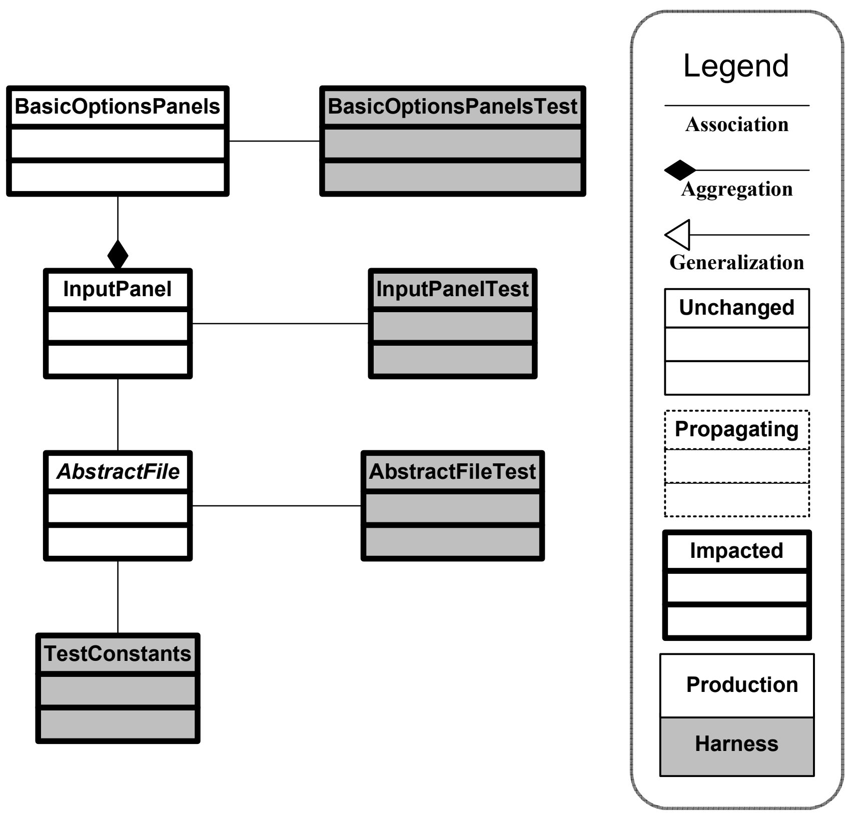


Figure A.52 Change 7 Impact Analysis UML

A.7.4 Prefactoring

No prefactoring was done during this change. The programmer did not see any prefactoring that would make the change easier. That is not to say that prefactoring could not have been done; but rather that for this change the programmer decided to do the actualization and then perform all refactoring during the postfactoring stage.

A.7.5 Actualization

During actualization, the programmer created a new class that extends `JPanel` and holds the 4 `JCheckBox` objects for properties searches. This new class was added to muCommander through incorporation. This class, `PropertiesPanel`, has a method to enable and disable the `JCheckBox` objects. It implements the `ActionListener` interface and listens to the archive and directory `JCheckBox` objects. If one of these boxes is checked the other is disabled, because it is impossible for a file to be both. It also creates objects of 4 new classes that implement the `SearchOption` interface. Additionally, a test class, `PropertiesPanelTest`, was added for this class.

The programmer added 4 new classes that implement the `SearchOption` interface, `ArchiveOption`, `DirectoryOption`, `HiddenOption` and `ReadOnlyOption`, through incorporation. They add themselves to the `SearchManager` object when their corresponding `JCheckBox` is selected. They each have a `meetsCriteria()` method from the `SearchOption` interface that returns true, if an `AbstractFile` sent to it is an archive, directory, hidden file or read-only file. The programmer added `ArchiveOptionTest`, `DirectoryOptionTest`, `HiddenOptionTest` and `ReadOnlyTest`, test classes for these classes.

The AbstractFile class had methods isArchive(), isDirectory() and isHidden() but it did not have an isReadOnly() method. The programmer added one and added a test for it to AbstractFileTest. This part of the change impacted a class not found during impact analysis, ProxyFile. ProxyFile must override all of AbstractFile's methods, so when the method isReadOnly() was added to AbstractFile, a test in ProxyFileTest failed (section A.7.7). The programmer added an overridden method isReadOnly() to ProxyFile.

The programmer then added an object of type PropertiesPanel to the BasicOptionsPanels. To accommodate the new panel in the GUI, InputPanel was changed to modify the GUI layout.

Finally, 3 new files for use in unit and functional tests were added to the project, an archive file, a hidden file and a read-only file. The programmer then added fields corresponding to them to the TestConstants class.

The total of each class by type of visit is listed in Table A.78. Table A.79 is a summary of the changes made to each class during actualization and the LOC added and deleted. Figure A.53 is a UML of actualization.

Table A.78 Change 7 Actualization Summary

Title	Code Files					
	Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
Property Search	7	7	11	0	0	1

Table A.79 Change 7 Actualization Code Files

#	Code File	Task	Lines of Code		
			Added	Deleted	Total
1	PropertiesPanel	Added class	89	0	89
2	ArchiveOption	Added class	23	0	23
3	DirectoryOption	Added class	23	0	23
4	HiddenOption	Added class	23	0	23
5	ReadOnlyOption	Added class	27	0	27
6	AbstractFile	Added method	3	0	3
7	ProxyFile	Added method	4	0	4
8	BasicOptionsPanels	Added field, changed methods	39	3	42
9	InputPanel	Changed method	2	2	4
10	PropertiesPanelTest	Added test class	76	0	76
11	ArchiveOptionTest	Added test class	43	0	43
12	DirectoryOptionTest	Added test class	43	0	43
13	HiddenOptionTest	Added test class	39	0	39
14	ReadOnlyOptionTest	Added test class	39	0	39
15	AbstractFileTest	Added test	5	0	5
16	BasicOptionsPanelTest	Changed tests	7	2	9
17	PropertySearchFuncTest	Added test class	205	0	205
18	TestConstants	Added fields	8	0	8

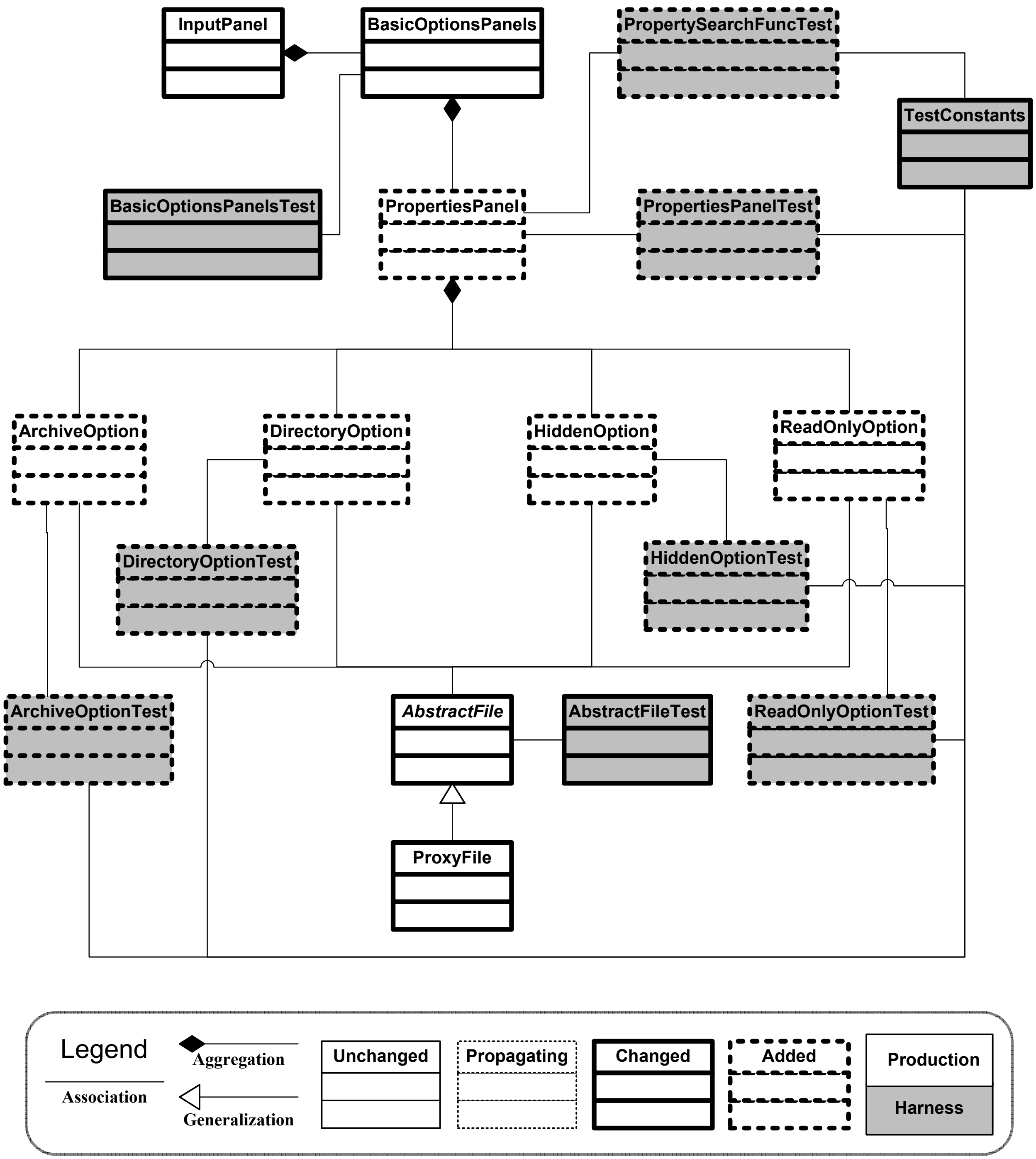


Figure A.53 Change 7 Actualization UML

A.7.5.1 PropertiesPanel class

The programmer added this class; it extends `JPanel` and contains 4 `JCheckBox` fields. These fields correspond to archive, directory, hidden and read-only searches. They each have a class implementing the `SearchOption` and `ActionListener` interfaces added as a listener.

The `setEnabled()` method was overridden to also enable the 4 `JCheckBox` objects when the class is enabled. The class also implements the `ActionListener` interface; it listens to the archive and directory `JCheckBox` objects. When one is checked the other is disabled in the `actionPerformed()` method. The methods `archiveBoxSetEnabled()` and `directoryBoxSetEnabled()` are called by `setEnabled()` and only enable the `JCheckBox` if the other is not.

A.7.5.2 ArchiveOption class

This class implements the `ActionListener` and `SearchOption` interfaces. It listens to the archive `JCheckBox` object in `PropertiesPanel` and adds itself to the `SearchManager`, if the box is checked. The `meetsCriteria()` method calls `AbstractFile`'s `isArchive()` method and returns the `boolean` value returned by that method.

A.7.5.3 DirectoryOption class

This class implements the `ActionListener` and `SearchOption` interfaces. It listens to the directory `JCheckBox` object in the `PropertiesPanel` and adds itself to the `SearchManager`, if the box is checked. The `meetsCriteria()` method calls `AbstractFile`'s `isDirectory()` method and returns the `boolean` value returned by that method.

A.7.5.4 HiddenOption class

This class implements the `ActionListener` and `SearchOption` interfaces. It listens to the hidden `JCheckBox` object in the `PropertiesPanel` and adds itself to the SearchManager, if the box is checked. The meetsCriteria() method calls AbstractFile's isHidden() method and returns the boolean value returned by that method.

A.7.5.5 ReadOnlyOption class

This class implements the ActionListener and SearchOption interfaces. It listens to the read-only JCheckBox object in the PropertiesPanel and adds itself to the SearchManager, if the box is checked. The meetsCriteria() method calls AbstractFile's isReadOnly() method and returns the boolean value returned by that method.

A.7.5.6 AbstractFile abstract class

The programmer added a method isReadOnly() to this class. The method checks the AbstractFile's permissions to see if writing is permitted; if it is it returns true, else it returns false.

A.7.5.7 ProxyFile class

The programmer missed this class during impact analysis. According to JRipples this class has 322 neighbors the programmer did not visit these classes during impact analysis. However, this class is a proxy implementation of AbstractFile and it requires that all non-final methods be overridden. To enforce this, testAllMethodsOverriden() fails if a method in AbstractFile's is not overridden by ProxyFile.

The programmer added a method isReadOnly() to this class. The method overrides isReadOnly() from AbstractFile. It just calls isReadOnly() in AbstractFile and returns the same value. The test, `testAllMethodsOverriden()` did not need to be changed because it dynamically searches for methods in `AbstractFile` and fails if `ProxyFile` does not override them.

A.7.5.8 BasicOptionsPanels class

The programmer added a field of type `PropertyPanel` to this class. The method `getBasicOptionsPanel()` was then changed to call the method `add()` with this field as a parameter. The programmer organized the `JPanel` returned from the method `getBasicOptionsPanel()` by adding 2 `JSeparator` objects and the layout of the panel was changed to a `GridBagLayout`. The `setEnabled()` method now also calls the `setEnabled()` method in `PropertiesPanel`.

A.7.5.9 InputPanel

The programmer changed the `createOptionsPanel()` method to put the `DatePanel` object below the `BasicOptonsPanel` because the 2 did not fit next to each other without expanding the width of the search window.

A.7.5.10 PropertiesPanelTest class

This class was added, it is the unit test suite for the `PropertiesPanel` class; it has 6 tests.

A.7.5.11 ArchiveOptionTest class

This class was added, it is the unit test suite for the `ArchiveOption` class; it has 3 tests.

A.7.5.12 DirectoryOptionTest class

This class was added, it is the unit test suite for the `DirectoryOption` class; it has 3 tests.

A.7.5.13 HiddenOptionTest class

This class was added, it is the unit test suite for the `HiddenOption` class; it has 3 tests.

A.7.5.14 ReadOnlyOptionTest class

This class was added, it is the unit test suite for the `ReadOnlyOption` class; it has 3 tests.

A.7.5.15 AbstractFileTest class

This class is the unit test suite for the `AbstractFile` class. It had 1 test added.

A.7.5.16 BasicOptionsPanelsTest class

This class is the unit test suite for the `BasicOptionsPanels` class. It had 3 tests changed

A.7.5.17 PropertySearchFuncTest class

This class is a functional test suite for property searches. It extends `SearchFuncTestSetUp` and has 11 tests.

A.7.5.18 TestConstants class

This class holds public static final fields used by the unit and functional tests. It added 4 fields of type `AbstractFile` corresponding to 4 files to be used for testing. One of these files is an archive, one a directory, one a hidden file and one a read-only file.

A.7.6 Postfactoring

During actualization code smells developed in `PropertiesPanel`. The responsibility to disable the archive `JCheckBox` when the directory `JCheckBox` is selected and vice-versa is misplaced. The programmer extracted a new class from

PropertiesPanel, called SearchOptionBox. It adds the responsibility of an antonym SearchOptionBox. When a SearchOptionBox is selected, it disables a registered antonym box.

The programmer placed the responsibility to add and remove the 4 classes, ArchiveOption, DirectoryOption, HiddenOption and ReadOnlyOption that implement SearchOption in these classes in actualization. This was also misplaced, so the programmer extracted this responsibility to SearchOptionBox. This class is now solely responsible for the actions of selecting the JCheckBox. This left the 4 classes that implement SearchOption with 1 method, meetsCriteria(). These classes could have been made into anonymous classes, but the programmer chose to keep them in their own files, because it makes the code clearer.

The classes InputPanel and BasicOptionsPanels shared the responsibility of laying out the GUI parts dealing with search options such as recursive searches, extension searches, property searches and date searches. After actualization it stood out that this was not clearly organized. The programmer created a new class, OptionsPanel to layout all of GUI classes that contain search options. One of these classes, BasicSearchOptionsPanels, had the JTextField that contains the search term. The programmer does not consider the search term a search option, so it was extracted to a new class SearchTermPanel.

This left InputPanel responsible for the layout of 4 panels. Three of these are separate production code classes, DirectoryPanel, SearchTermPanel and OptionsPanel. The forth panel holds a JLabel that displays a static String, a second JLabel that displays search option errors and an icon that is animated when a search is running. This panel is not significant enough for its own class; therefore it is created in a method, `createLabelPanel()` in `InputPanel`.

This refactoring resulted in broken contracts to clients of `InputPanel` and `BasicOptionsPanels`; this resulted in the programmer adding 9 code files to the changed set. The only 1 of the 9 added to the changed set that is production code is `SearchDialog` it has a method call that is responsible for requesting a `Component` to be the default when the dialog is created (section A.7.6). It is an anti-pattern that the programmer would like to remove, but it is a small concept that does not warrant its own class and the programmer is not aware of a listener that can accomplish this.

The other code files added to the change set were all part of the harness. These code files are: `BasicSearchFuncTest`, `ExtensionSearchFuncTest`, `SearchFuncTestSetUp`, `SearchTermOptionTest`, `ButtonPanelTest`, `ExtensionPanelTest`, `SearchDialogTest` and `SearchThreadTest`. The programmer did not plan to do to extract the `SearchTermPanel` and `OptionsPanel` classes at the start of the change. However, after the change code smells were present in `BasicOptionsPanels` and `InputPanel` that needed to be dealt with. The programmer decided not to visit the production code files that these harness code files test during impact analysis because he was familiar with them from his experiences in past changes. However, the programmer made the mistake of thinking the harness code files had similar dependencies as the production code files they test, which is not the case.

The harness code files have more dependencies than the production code files they test because the tests must not only create the dependencies of the class being tested, but also the dependencies of the dependencies. A test class may need objects of a few levels of dependencies. Additionally, the test's assertions may require an object of a dependency of the class being tested, especially in the case of methods with void return types. These circumstances make it likely that the changed set of the harness will be greater than the estimated impact set if refactoring not anticipated during impact analysis is done.

The total of each class by type of visit is listed in Table A.80. Table A.81 is a summary of the refactoring type and LOC added and deleted during postfactoring. Figure A.54 is a UML of postfactoring.

Table A.80 Change 7 Postfactoring Summary

Title	Code Files					
	Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
Property Search	27	27	6	0	0	9

Table A.81 Change 7 Postfactoring Code Files

#	Code File	Task	Lines of Code		
			Added	Deleted	Total
1	`InputPanel`	Extracted class from	24	35	59
2	`OptionsPanel`	Extracted class	84	0	84
3	`BasicOptionsPanel`	Renamed class, extracted class from	6	89	95
4	`SearchTermPanel`	Extracted class	27	0	27
5	`PropertiesPanel`	Extracted class from	31	62	93
6	`ArchiveOption`	Extracted class from	1	16	17
7	`DirectoryOption`	Extracted class from	1	16	17
8	`HiddenOption`	Extracted class from	1	16	17
9	`ReadOnlyOption`	Extracted class from	1	20	21
10	`SearchOptionBox`	Extracted class	55	0	55

11	SearchDialog	Changed method	1	1	2
12	AbstractFile	Javadoc	0	0	0
13	InputPanelTest	Changed tests	7	5	12
14	OptionsPanelTest	Added test class	65	0	65
15	BasicOptionsPanelTest	Renamed class, changed method, changed, extracted tests	24	65	89
16	SearchTermPanelTest	Changed tests	52	0	52
17	PropertiesPanelTest	Added method, changed, extracted tests	13	37	50
18	ArchiveOptionTest	Changed, extracted tests	4	22	26
19	DirectoryOptionTest	Changed, extracted tests	4	22	26
20	HiddenOptionTest	Changed, extracted tests	4	18	22
21	ReadOnlyOptionTest	Changed, extracted tests	4	18	22
22	SearchOptionBoxTest	Added test class	113	0	113
23	AbstractFileTest	Javadoc	0	0	0
24	PropertySearchFuncTest	Changed method, test	9	5	14
25	BasicSearchFuncTest	Changed tests	2	2	41
26	ExtensionSearchFuncTest	Changed test	1	1	2
27	SearchFuncTestSetUp	Changed method	2	2	4
28	SearchTermOptionTest	Changed method, tests	16	16	32
29	ButtonPanelTest	Changed test	1	1	2
30	ExtensionPanelTest	Changed method	2	5	7
31	SearchDialogTest	Changed test	2	2	4
32	SearchThreadTest	Changed tests	7	7	14
33	TestConstants	Added code blocks	23	0	23

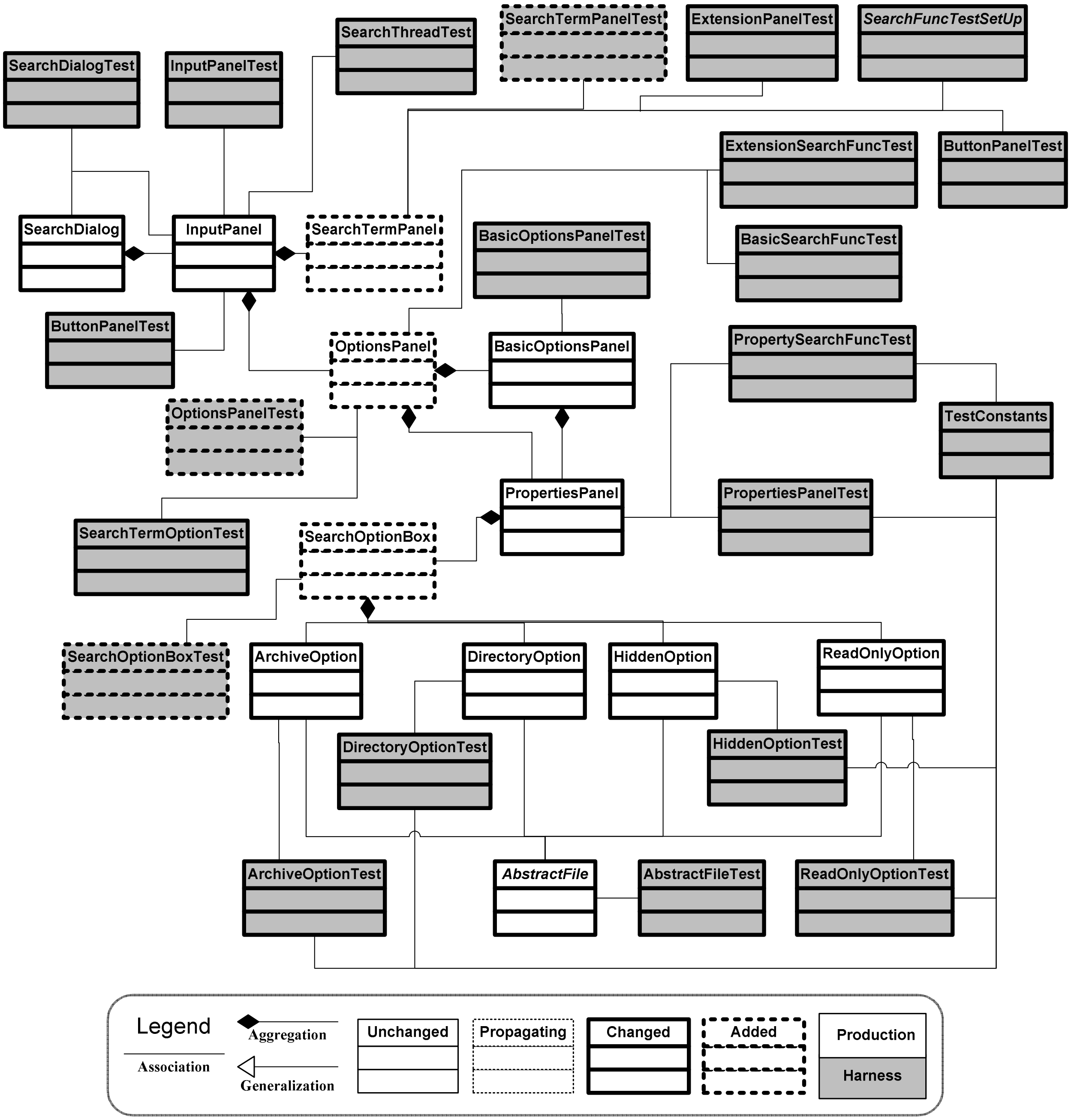


Figure A.54 Change 7 Postfactoring UML

A.7.6.1 InputPanel class

The programmer extracted the fields `DatePanel` and `BasicOptionsPanel` to `OptionsPanel` along with the method, `createOptionsPanel()`. The calls to their `setEnabled()` method were removed from the `switchToSearchState()` method.

The programmer then added new fields of type `SearchTermPanel` and `OptionsPanel`. Calls to these fields `setEnabled()` were added to `switchToSearchState()`.

A.7.6.2 OptionsPanel class

The programmer extracted this class from `InputPanel`, it extends `JPanel`. It has fields of type `BasicOptionsPanel`, `ExtensionPanel`, `PropertiesPanel`, `DatePanel` and `JPanel`. The method, `createPanel()` is called from the constructor; it adds the return value of the method `createTopPanel()` to the class along with the `DatePanel` object. The method, `createTopPanel()` lays out the field objects `BasicOptionsPanel`, `ExtensionPanel` and `PropertiesPanel` in the field object of type `JPanel` by calling `addComponent()`. The method `addComponent()` is a convenience method, that adds a `Component` to the `JPanel` field, in a designated grid cell. Finally, there is an overridden `setEnabled()` method that calls `setEnabled()` in all the inner panels.

A.7.6.3 BasicOptionsPanel class

The programmer extracted the fields, of type `JTextField`, and `SearchTermOption` along with the methods, `initInputFieldPanel()` and `getInputFieldPanel()` to a new class `SearchTermPanel`. Next the fields `ExtensionPanel` and `PropertiesPanel` along with the method `getBasicOptionsPanel` were extracted to `OptionsPanel`. The calls to these fields `setEnabled()` methods were extracted to the appropriate class from the `setEnabled()` method.

This left this class with 2 fields of type `JCheckBox` that handle the responsibility for recursive and case sensitive searches. The programmer changed the class to extend `YBoxPanel` and renamed it from `BasicOptionsPanels` to `BasicOptionsPanel` since it now only handles the responsibility for 1 panel.

A.7.6.4 SearchTermPanel class

The programmer extracted this class from `BasicOptionsPanel`. It contains a single field of type `JTextField`. It lays out that field and a static `JLabel`. There is also an overridden method `setEnabled()` to enable the field and request the focus when called.

A.7.6.5 PropertiesPanel class

The programmer extracted a new class, `SearchOptionBox` from this class. The responsibility contained in the methods `archiveBoxSetEnabled()` and `directoryBoxSetEnabled()` was extracted to this new class. The `ActionListener` and its method `actionPerformed()` was also extracted to `SearchOptionBox`. Next the 4 fields of type `JCheckBox` were changed to type `SearchOptionBox`.

The constructor was long and difficult to follow; it repeated similar code 4 times to initialize the 4 `JCheckBox` fields. A new method `addAtCell()` was extracted from it.

A.7.6.6 ArchiveOption, DirectoryOption, HiddenOption and ReadOnlyOption class

These classes were all created during actualization; they all had the same code in their constructors and `actionPerformed()` methods. The programmer extracted the field of type `SearchManager` and the `ActionListener` interface with its methods actionPerformed() to SearchOptionBox from all of these classes. This left the constructor empty, so it was deleted.

A.7.6.7 SearchOptionBox class

The programmer extracted this class from PropertiesPanel, ArchiveOption, DirectoryOption, HiddenOption and ReadOnlyOption. The class extends JCheckBox. It is responsible for adding and removing a SearchOption class from the SearchManager object passed to its constructor, when the JCheckBox is selected. It is also responsible for disabling a registered antonym SearchOptionBox when it is selected.

This class has 3 fields of type SearchOption, SearchManager and SearchOptionBox. The SearchOptionBox field is an antonym box that is disabled when this object of SearchOptionBox is selected.

The class implements the ActionListener interface. The actionPerformed() method calls enableOption() and if the antonym field is not null, it will call its setEnabled() method. The c method calls the method addOption() on the field object of type SearchManager passing the field object of type SearchOption if this object is selected, otherwise it calls removeOption() with the same field.

The method setEnabled() is also overridden; it only enables this object if it does not have a selected antonym.

A.7.6.8 SearchDialog class

The programmer did not visit or include this class in the estimated impact set. The class was impacted because its constructor calls an inherited method, `setInitialFocusComponent()`, to put the cursor in the field that accepts search terms. This field was extracted from `BasicOptionsPanels` to `SearchTermPanel` it did not make sense to create a man-in-the-middle by leaving the getter for the field in `BasicOptionsPanels`, so `SearchDialog` was impacted.

The method call in the constructor `getBasicOptionsPanels().getInputBox()` on the field object of type `InputPanel` had to be changed to `getSearchTermPanel().getInputBox()`. This method call's return value is the parameter passed to `setInitialFocusComponent()`.

A.7.6.8 AbstractFile class

The programmer added Javadoc to the method added during actualization.

A.7.6.9 InputPanelTest class

This class is the unit test class for the `InputPanel` class. It had 3 tests changed.

A.7.6.10 OptionsPanelTest class

This class was added, it is the unit test suite for the `OptionsPanel` class; it has 5 tests.

A.7.6.11 BasicOptionsPanelTest class

This class is the unit test class for the `BasicOptionsPanel` class. It had 2 tests changed, 2 added and 5 deleted. Its `setUp()` method was changed and it was renamed, dropping the ‘s’ after Panel just as the class it tests did.

A.7.6.12 SearchTermPanelTest class

This class was added, it is the unit test suite for the `SearchTermPanel` class; it has 4 tests.

A.7.6.13 PropertiesPanelTest class

This class is the unit test class for the `PropertiesPanel` class. It had 1 test changed and 3 deleted. A method `setUpBeforeClass()` was added to call the static method `loadDictionaryFile()` in the `Translator` class.

A.7.6.14 ArchiveOptionTest, DirectoryOptionTest, HiddenOptionTest and ReadOnlyOptionTest classes

These are the unit test classes for `ArchiveOption`, `DirectoryOption`, `HiddenOption` and `ReadOnlyTest` classes. They all had 1 test changed and 1 deleted.

A.7.6.15 SearchOptionBoxTest class

This class was added, it is the unit test suite for the `SearchOptionBox` class; it has 10 tests.

A.7.6.16 AbstractFileTest class

This class is the unit test class for the `AbstractFile` class. It had Javadoc added to a test added during actualization.

A.7.6.17 PropertySearchFuncTest class

This class is a functional test suite for property searches. Its `setUp()` method and 2 tests were changed.

A.7.6.18 BasicSearchFuncTest class

This class is a functional test suite for basic searches. Two tests were changed.

A.7.6.19 ExtensionSearchFuncTest class

This class is a functional test suite for extension searches. One test was changed.

A.7.6.20 SearchFuncTestSetUp abstract class

This is a class that is extended by test classes that need a `SearchDialog` object for testing. It changed its `setUp()` method.

A.7.6.21 SearchTermOptionTest class

This class is the unit test class for the `SearchTermOption` class. Its `setUp()` method and 2 tests were changed.

A.7.6.22 ButtonPanelTest class

This class is the unit test class for the `ButtonPanel` class. It had 1 test changed.

A.7.6.23 ExtensionPanelTest class

This class is the unit test class for the `ExtensionPanel` class. Its `setUp()` method was changed.

A.7.6.24 SearchDialogTest class

This class is the unit test class for the `SearchDialog` class. It had 1 test changed.

A.7.6.25 SearchThreadTest class

This class is the unit test class for the `SearchThread` class. It had 5 tests changed.

A.7.6.26 TestConstants class

This class holds public static final fields used by the unit and functional tests. The programmer added 2 static code blocks to set the properties on 2 of the fields added during actualization, so that it does not need to be done manually by programmers after checking out the project from the repository.

A.7.7 Verification

During actualization and postfactoring all regression tests passed. The programmer found 3 bugs during the change; 2 during actualization and 1 during postfactoring. The first bug found during actualization, the test, `testSetEnabled()` in the `PropertiesPanelTest` harness code file failed. The programmer added a call to the super method in the overridden method `setEnabled()` in `PropertiesPanel` then the test passed.

The programmer discovered a bug from a previous change request during actualization. When the programmer investigated the failed test, `testSetEnabel()`, he ran a manual intervention test. During this he discovered that, if a directory to search in is chosen with the file chooser, the search directory is not updated. A bug was added to the backlog.

The third bug the programmer discovered was during postfactoring. The tests `testArchiveBoxSetEnabled()` and `testDirectoryBoxSetEnabled()` both failed after the class `SearchOptionBox` was extracted from `PropertiesPanel`. During the class extraction the programmer neglected to add the lines `archiveBox.addAntonym(directoryBox);` and `directoryBox.addAntonym(archiveBox);` to the `PropertiesPanel` constructor. The programmer added the lines and continued with postfactoring. Table A.82 shows the statement level verification coverage of each production code file changed.

Table A.82 Change 7 Statement Verification

#	Code File	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	SearchOptionBox	23	23	100.0	0	0
2	BasicOptionsPanel	13	13	100.0	0	0
3	OptionsPanel	43	43	100.0	0	0
4	PropertiesPanel	24	24	100.0	2	2
5	SearchTermPanel	11	11	100.0	0	0
6	ArchiveOption	1	1	100.0	0	0
7	InputPanel	27	27	100.0	0	0
8	DirectoryOption	1	1	100.0	0	0
9	SearchDialog	44	43	97.7	0	0
10	HiddenOption	1	1	100.0	0	0
11	ReadOnlyOption	1	1	100.0	0	0
12	AbstractFile	233	170	73.0	0	0
13	ProxyFile	64	54	84.4	0	0

A.7.8 Timing Data

Table A.83 contains the timing data for the change.

Table A.83 Change 7 Timing Totals

Phase	Time (hh:mm)
Concept Location	0:00
Impact Analysis	0:38
Prefactoring	0:00
Prefactoring Testing	0:00
Actualization	2:57
Actualization Testing	2:32
Postfactoring	3:54
Postfactoring Testing	4:22

A.7.9 Conclusions

The programmer mistakenly thought that this change would be simpler than it was to actualize. The timing data shows that the change's actualization and prefactoring phase took longer than change 6, which the programmer considered more difficult. The total time of the change was 94% of the change 6 total time. The impact analysis should have been more rigorous. This led to extra time being spent on testing during postfactoring.

The changed set of 7 code files was equal to the estimated impact set. However, an extra production code file was impacted and one of the harness code files was not. During actualization, a regression test failed because the class `ProxyFile`, an implementation of `AbstractFile`, did not implement a method the programmer added. The programmer mistakenly assumed that an added `boolean` getter would not have an impact. However, `ProxyFileTest` requires `ProxyFile` to override all

AbstractFile's methods. The harness code file InputPanelTest did not need to be changed.

During postfactoring 9 code files that were not part of the estimated impact set or the changed set were impacted. At the start of postfactoring it became clear to the programmer that the responsibility held in InputPanel and BasicOptionsPanels could be better organized. The programmer extracted OptionsPanel and moved responsibility between these code files. This opportune reorganization impacted the 9 additional code files.

After completing this change request, the search feature of muCommander has grown quite capable. It still has room to grow, but it provides a user a large combination of methods to search for files in the file system. Table A.84 lists the totals for each set of code files for each change request of this iteration to date. Table A.85 is the current product backlog. Figure A.55 to Figure A.59 are screen shots of muCommander showing the change request functionality.

Table A.84 Change 7 Code File Summary

#	Change	Number in Code files						
		Visited Concept Location	Estimated Impact Set	Changed Set	Added during			Total Project
					Pre	Act	Post	
0	Original Baseline	N/A	N/A	N/A	N/A	N/A	N/A	1,070
1	Basic Search	5	3	4	0	4	0	1,074
2	Recursive search	0	3	4	4	0	5	1,083
3	Advanced Output	6	21	11	2	4	10	1,099
4	Date Search	0	13	12	2	16	3	1,120
5	Case Sensitive	0	16	15	8	2	3	1,133
6	Extension Search	0	11	6	2	7	(5)	1,137
7	Properties Search	0	7	7	0	11	6	1,154

Table A.85 Change 7 Current Product Backlog

#	Title	Complete	User Story
1	Basic Search	x	Add a basic search function that allows a user to search in the current directory for all or part of the title of a folder or file, and return a list of the matching files and directories.
2	Recursive Search	x	Add the ability to search inside all directories.
3	Advanced Output	x	Change the output to a table similar to the main muCommander window.
4	Date Search	x	Allow the user search by a date of file's modification.
5	Case Sensitive Search	x	Add capability to search by case sensitive search terms.
6	Extension Search	x	Add the ability to search for files with specific extensions.
7	Properties Search	x	Add options to search for files based on their properties.
8	Directory Chooser Bug		Choosing a directory with the file chooser doesn't update the search directory.
9	Date Bug		DateOption is not removed when disabled.
10	Size Search		Add the ability to search for a file by its size.
11	Regular Expression Search		Add capability to search by a regular expression.
12	Lucene Search		Incorporate the Apache Lucene search.
13	JDayChooserTest Bug		The test testSetMonth() fails on last day of month, if next month has fewer days

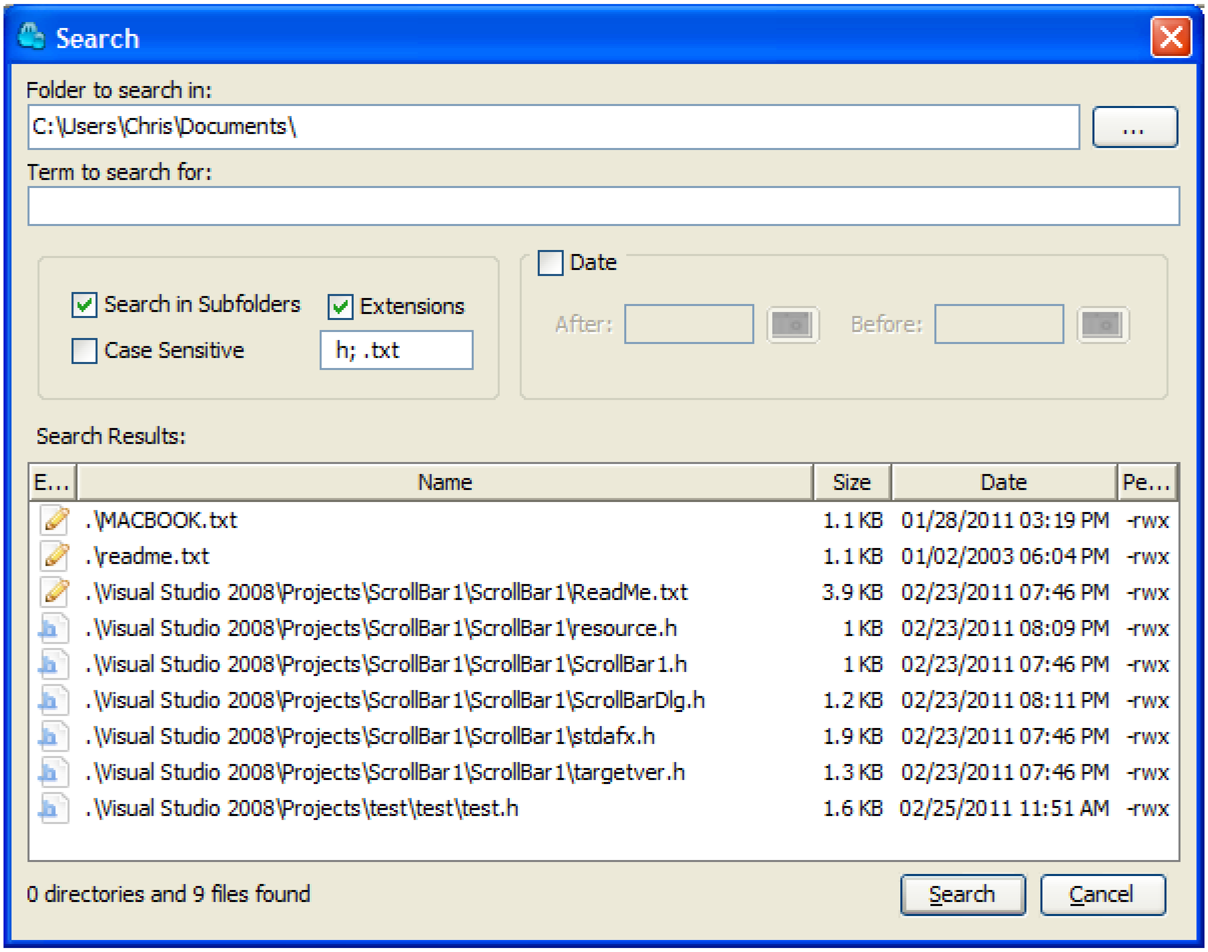


Figure A.55 Search window before Properties Search Change

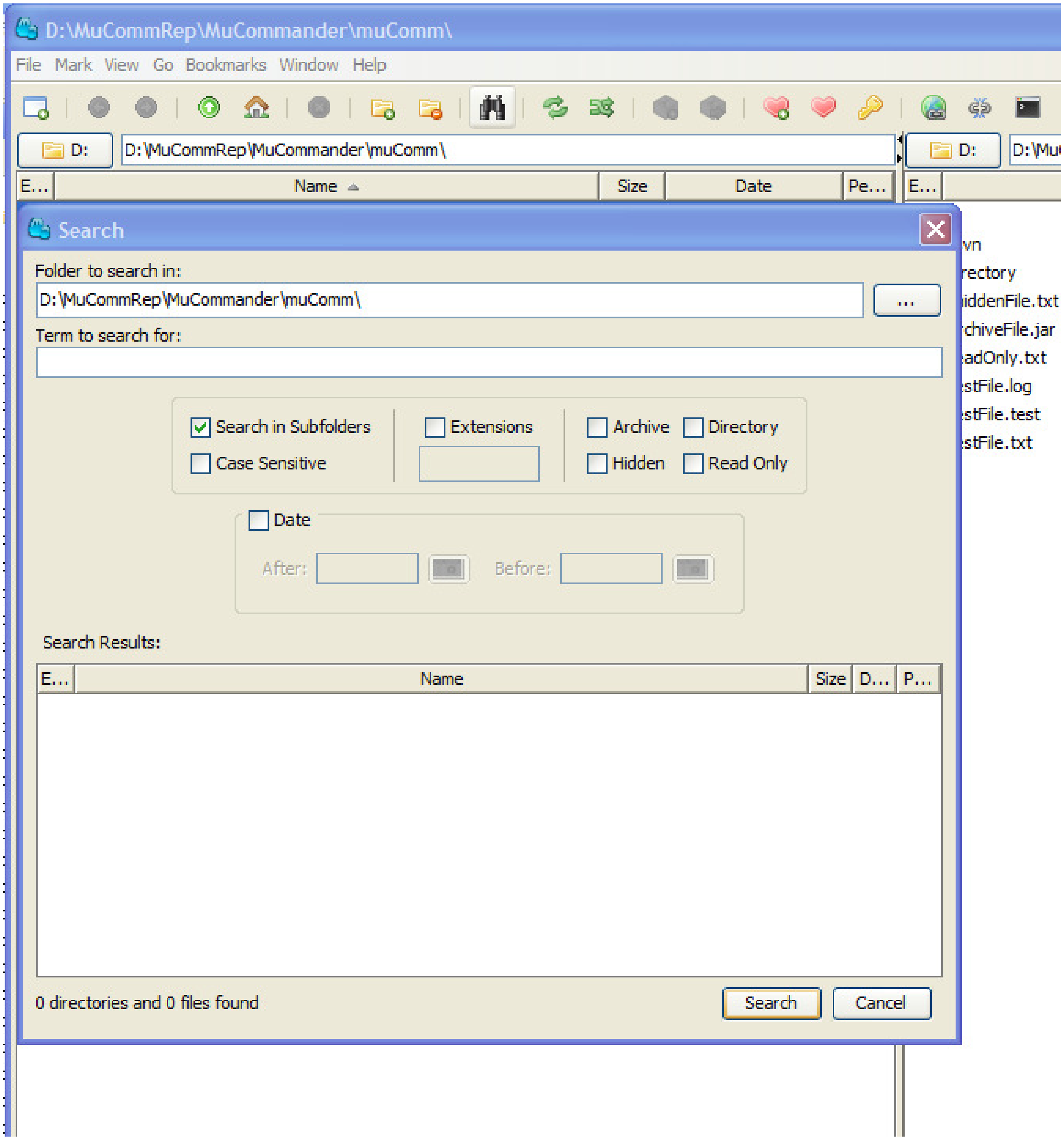


Figure A.56 Search window Properties Search Change

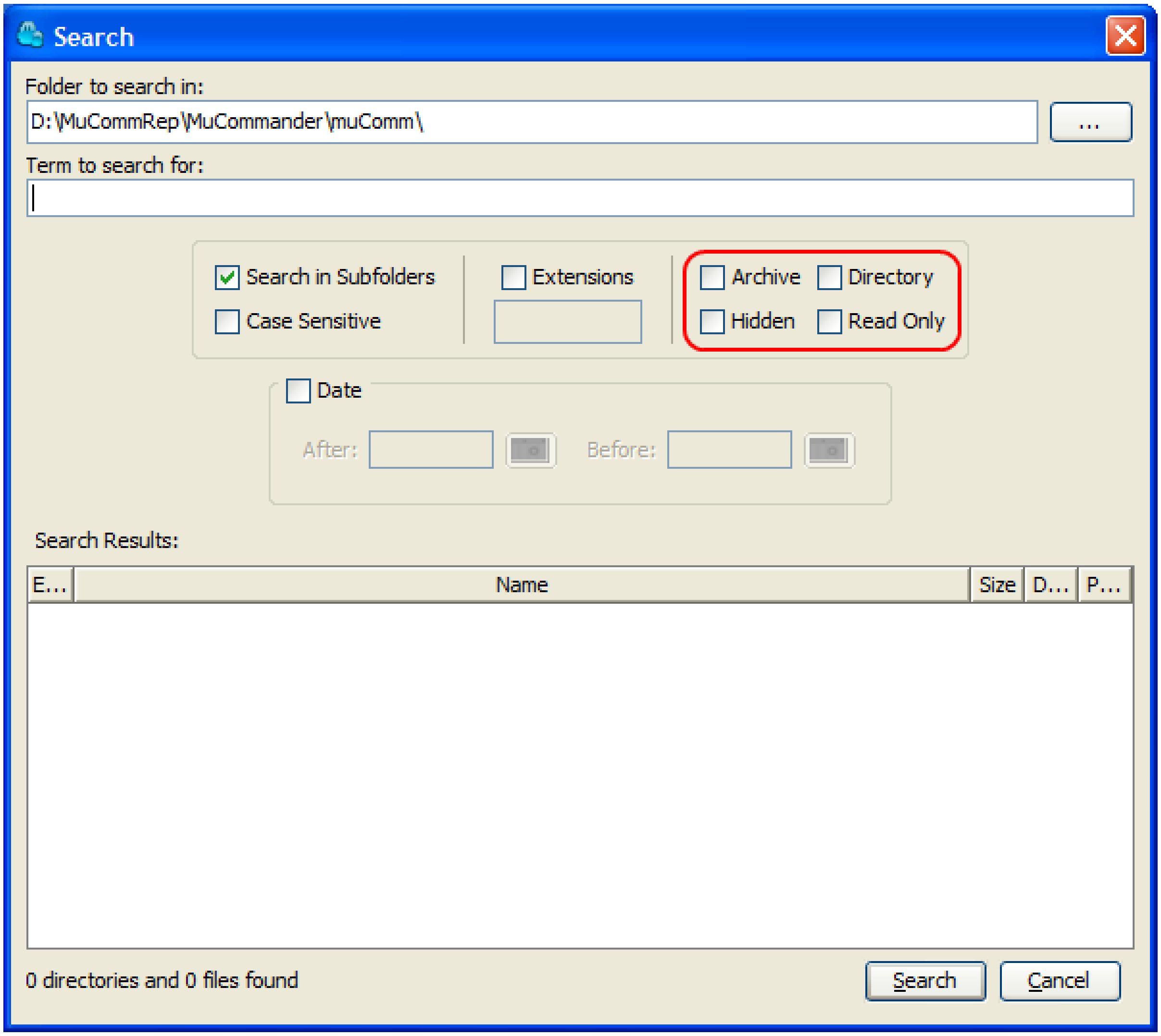


Figure A.57 Search window Properties Search circled

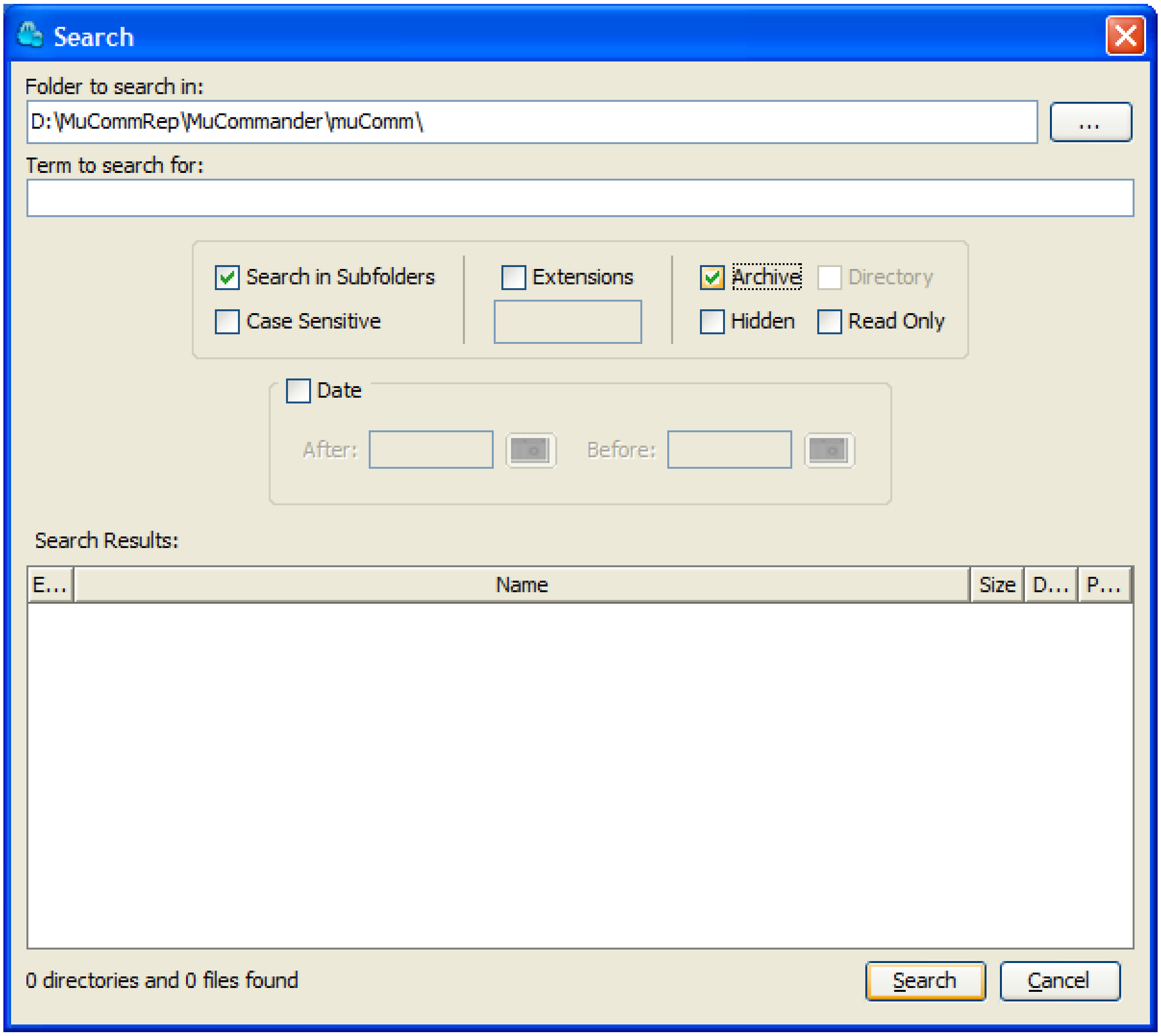


Figure A.58 Search window Archive checked, Directory disabled

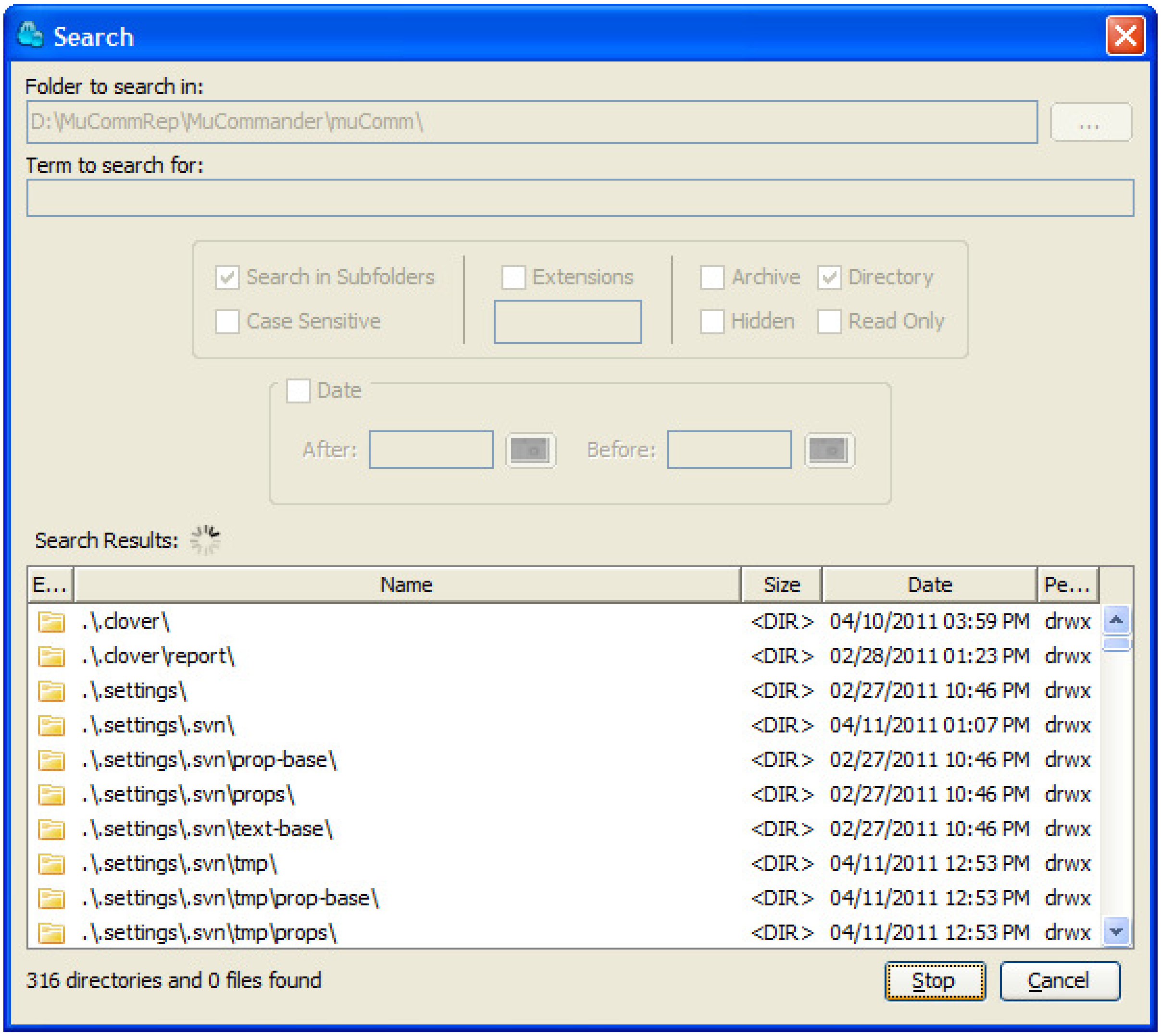


Figure A.59 Search window search running, returning Directories

A.8 SIP – Change 8 File Chooser Bug

A.8.1 Initialization

Choosing a directory with the file chooser does not update the search directory. The programmer discovered a bug in muCommander during change request 7 through code inspection. He determined that it was caused during the prefactoring phase of change request 5. The issue is that a user can type a directory directly into the text box to search it, but if the user chooses a directory from the GUI file chooser, the search directory is not updated. The programmer added this bug to the product backlog as a priority 3 bug because there is a workaround.

A.8.2 Concept Location

No concept location was needed for this change. The programmer found this bug during a code inspection; the concept extension is located in the `DirectoryPanel` code file.

A.8.3 Impact Analysis

No impact analysis was necessary. The programmer identified the file with the concept extension, `DirectoryPanel` as the only production code file in the estimated impact set. He added the harness code files `DirectoryPanelTest` and `BasicSearchFuncTest` so he could add tests to prevent the bug from reoccurring. Table A.86 lists the code files in the estimated impact set. Figure A.60 shows a UML diagram of the estimated impact set.

Table A.86 Change 8 Impact Analysis Code Files Visited

#	Code File	Tool used	Impacted?	Comments
1	`DirectoryPanel`	Code inspection	Impacted	Contains concept extension.
2	`DirectoryPanelTest`	Previous Knowledge	Impacted	Not Visited
3	`BasicSearchFuncTest`	Previous Knowledge	Impacted	Not Visited

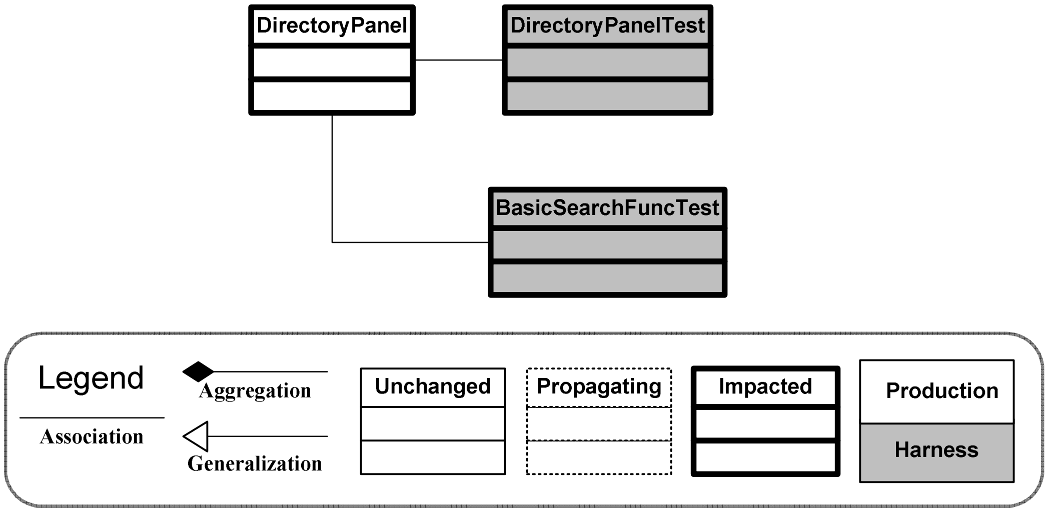


Figure A.60 Change 8 Impact Analysis UML

A.8.4 Prefactoring

The programmer extracted a method called `directoryFieldUpdate()` from the existing `keyReleased()` method in `DirectoryPanel`. All of the body of `keyReleased()` was extracted to the new method. He did this because the `KeyListener` interface and its `keyReleased()` method will be replaced during actualization to fix the bug. The programmer also added a test for the new method, to `DirectoryPanelTest`.

Table A.87 is the total code files the change propagated to. Table A.88 is a summary of the LOC for each code file and Figure A.61 is a UML of prefactoring.

Table A.87 Change 8 Prefactoring Summary

Title	Code Files					
	Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
Directory Chooser Bug	2	2	0	0	0	0

Table A.88 Change 8 Prefactoring Code Files

#	Code File	Task	Lines of Code		
			Added	Deleted	Total
1	`DirectoryPanel`	Extracted method	3	0	3
2	`DirectoryPanelTest`	Added test	6	0	6

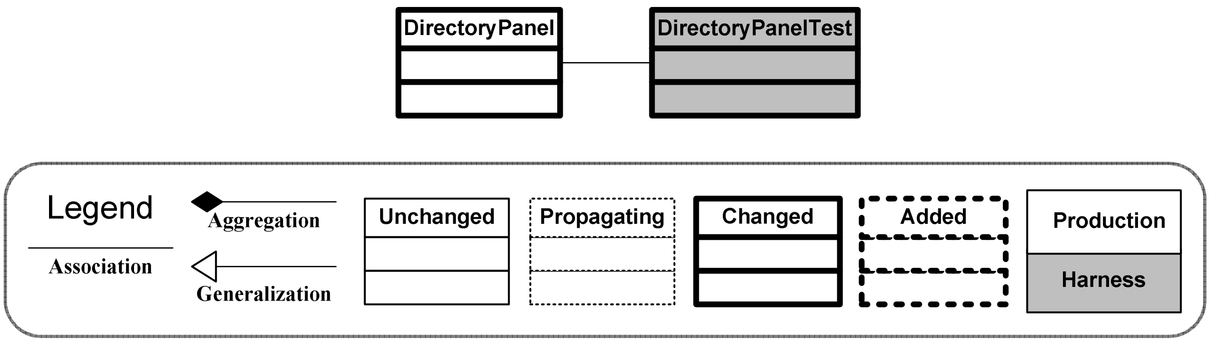


Figure A.61 Change 8 Prefactoring UML

A.8.4.1 DirectoryPanel class

The programmer extracted the method `directoryFieldUpdate()` method from the method `keyReleased()`. The extracted method contains the entire body of `keyReleased()`, which now just calls the extracted method. The programmer did this to make it easier to replace the `keyReleased()` method during actualization.

A.8.4.2 DirectoryPanelTest class

This class is the unit test suite for the `DirectoryPanel` class. It had 1 test added.

A.8.5 Actualization

To actualize the change request, the programmer replaced the `KeyListener` interface with a `DocumentListener` interface. This interface initiates an event if the text in a `JTextField` is changed regardless of the source; the `KeyListener` interface only initiated events if the user typed a key. So when the directory chooser updated the text field, there was no event.

The programmer then added tests to `DirectoryPanelTest` for the `DocumentListener` interface's methods and deleted the test for the `keyListener()` method. He then added a test to `BasicSearchFuncTest` that uses the GUI file chooser to select a directory to search and asserts that the selected directory is the current search directory.

The total of each class by type of visit is listed in Table A.89. Table A.90 is a summary of the changes made to each class during actualization and the LOC added and deleted. Figure A.62 is a UML of actualization.

Table A.89 Change 8 Actualization Summary

Title	Code Files					
	Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
Directory Chooser Bug	3	3	0	0	0	0

Table A.90 Change 8 Actualization Code Files

#	Code File	Task	Lines of Code		
			Added	Deleted	Total
1	`DirectoryPanel`	Added, deleted, changed methods	10	9	19
2	`DirectoryPanelTest`	Added, deleted tests	9	3	12
3	`BasicSearchFuncTest`	Added test	23	0	23

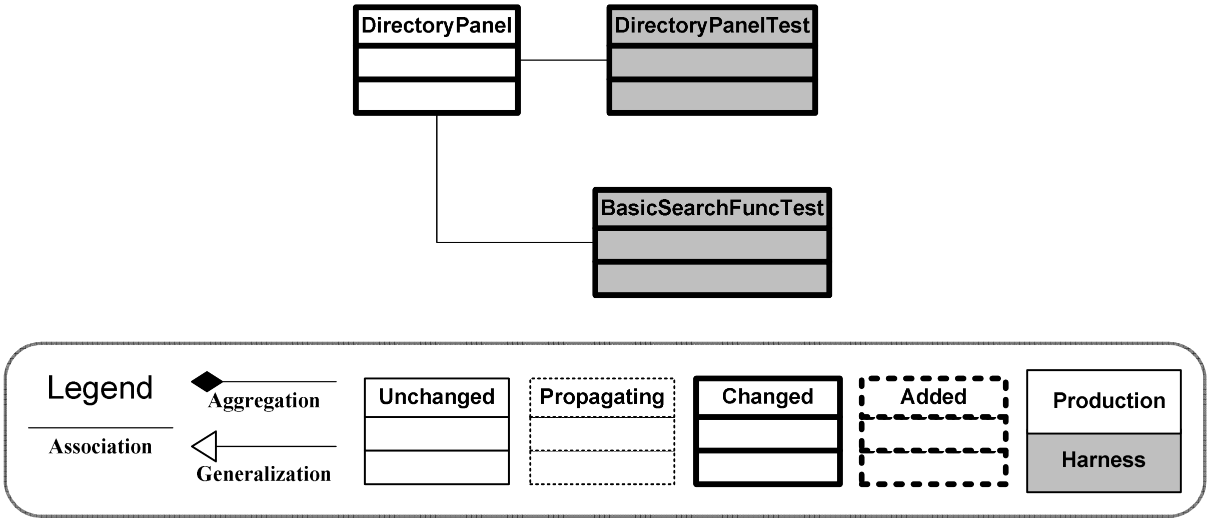


Figure A.62 Change 8 Actualization UML

A.8.5.1 DirectoryPanel class

The programmer removed the `KeyListener` interface from this class and its 3 methods. Only the `keyReleased()` method from the interface was used; it called `directoryFieldUpdate()` on a key released event. The programmer added a `DocumentListener` interface, with its 3 methods. The `insertUpdate()` and `removeUpdate()` methods both call `directoryFieldUpdate()`. The third interface method is `changedUpdate()` is unused.

A.8.5.2 DirectoryPanelTest class

This class is the unit test suite for the `DirectoryPanel` class. It had 2 tests added and 1 deleted.

A.8.5.3 BasicSearchFuncTest class

This class is a functional test for basic search functionality. It had 1 test added.

A.8.6 Postfactoring

No Postfactoring was necessary for this change request.

A.8.7 Verification

All regression tests passed after the change request. No new bugs were found. Table A.91 shows the test coverage of `DirectoryPanel`, the only production code file changed.

Table A.91 Change 8 Statement Verification

#	Code File	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	`DirectoryPanel`	55	54	98.2	0	0

A.8.8 Timing Data

Table A.92 contains the timing data for the change.

Table A.92 Change 8 Timing Totals

Phase	Time (hh:mm)
Concept Location	0:00
Impact Analysis	0:00
Prefactoring	0:07
Prefactoring Testing	0:09
Actualization	0:16
Actualization Testing	0:37
Postfactoring	0:00
Postfactoring Testing	0:00

A.8.9 Conclusions

This bug fix went smoothly; extracting a method during prefactoring made actualization simple. The functional test added during actualization is important, it will assure that if this bug is added to the program again the programmer will know it quickly and can address it before it is committed to another baseline.

Table A.93 lists the totals for each set of code files for each change request of this iteration to date. Table A.94 is the current product backlog.

Table A.93 Change 8 Code File Summary

#	Change	Number in Code Files						
		Visited Concept Location	Estimated Impact Set	Changed Set	Added during			Total
					Pre	Act	Post	Project
0	Original Baseline	N/A	N/A	N/A	N/A	N/A	N/A	1,070
1	Basic Search	5	3	4	0	4	0	1,074
2	Recursive search	0	3	4	4	0	5	1,083
3	Advanced Output	6	21	11	2	4	10	1,099
4	Date Search	0	13	12	2	16	3	1,120
5	Case Sensitive	0	16	15	8	2	3	1,133
6	Extension Search	0	11	6	2	7	(5)	1,137
7	Properties Search	0	7	7	0	11	6	1,154
8	Date Chooser Bug	0	3	3	0	0	0	1,154

Table A.94 Change 8 Current Product Backlog

#	Title	Complete	User Story
1	Basic Search	x	Add a basic search function that allows a user to search in the current directory for all or part of the title of a folder or file, and return a list of the matching files and directories.
2	Recursive Search	x	Add the ability to search inside all directories.
3	Advanced Output	x	Change the output to a table similar to the main muCommander window.
4	Date Search	x	Allow the user search by a date of file's modification.
5	Case Sensitive Search	x	Add capability to search by case sensitive search terms.
6	Extension Search	x	Add the ability to search for files with specific extensions.
7	Properties Search	x	Add options to search for files based on their properties.
8	Directory Chooser Bug	x	Choosing a directory with the file chooser doesn't update the search directory.
9	Date Bug		DateOption is not removed when disabled.
10	Size Search		Add the ability to search for a file by its size.
11	Regular Expression Search		Add capability to search by a regular expression.
12	Lucene Search		Incorporate the Apache Lucene search.
13	JDayChooserTest Bug		The test testSetMonth() fails on last day of month, if next month has fewer days

SIP – Change 9 Date Search Bug

A.9.1 Initialization

The `DateOption` is not removed from the `SearchManager` when it is disabled. The programmer discovered a bug in during the impact analysis phase of change request 6. When the `JCheckBox` that turns the date search on and off is unchecked to turn the date search off, the `DateOption` objects are not removed from the `SearchManager`. This means that the date search is still enabled, resulting in incomplete search results. If the date search is never enabled or if dates are not entered in the `DateField` objects, the search will be correct; therefore, this bug has a priority of 3.

A.9.2 Concept Location

No concept location was needed for this change. The programmer found this bug during a code inspection; the concept extension is located in the `DatePanel` code file.

A.9.3 Impact Analysis

No impact analysis was necessary. Based on knowledge from previous change requests the programmer knew that the code file with the concept extension, `DatePanel` and `DateField` and `DateOption` would all be in the estimated impact set. He added the harness code files `DatePanelTest`, `DateFieldTest`, `DateOptionTest` and `DateSearchFuncTest` so he could add tests to prevent the bug from reoccurring. Table A.95 lists the code files in the estimated impact set. Figure A.63 shows a UML diagram of the estimated impact set.

Table A.95 Change 9 Impact Analysis Code Files Visited

#	Code File	Tool used	Impacted?	Comments
1	DatePanel	Code inspection	Impacted	Concept Location
2	DateField	Previous Knowledge	Impacted	Supplier to DatePanel Not Visited
3	DateOption	Previous Knowledge	Impacted	Supplier to DatePanel Not Visited
4	DatePanelTest	Previous Knowledge	Impacted	Not Visited
5	DateFieldTest	Previous Knowledge	Impacted	Not Visited
6	DateOptionTest	Previous Knowledge	Impacted	Not Visited
7	DateSearchFuncTest	Previous Knowledge	Impacted	Not Visited

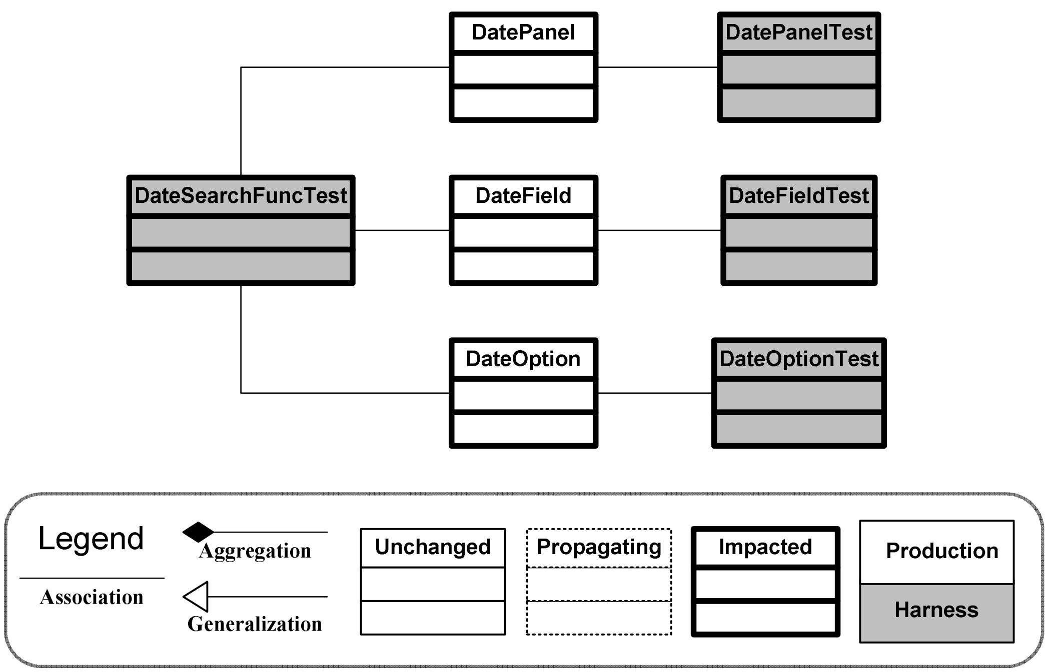


Figure A.63 Change 9 Impact Analysis UML

A.9.4 Prefactoring

No prefactoring was necessary for this change request.

A.9.5 Actualization

To actualize the change request, the programmer added the ActionListener interface to the DateOption class. He then added the DateOption objects initialized in DatePanel as listeners to the dateBox field. This will add and remove objects of this class to the set of SearchOption objects in SearchManager as appropriate. The change propagated to DateField, which had a redundant method call in its focusLost() method that was adding the DateOption object back into SearchManager.

The programmer then changed tests in DatePanelTest and DateOptionTest to test the new contracts. He then added a test to DateSearchFuncTest that enables and disable a date search and asserts that the DateOption objects are removed from SearchManager. The change request did not propagate to the DateFieldTest harness code file, its tests still passed after the redundant call was removed from DateField.

The total of each class by type of visit is listed in Table A.96. Table A.97 is a summary of the changes made to each class during actualization and the LOC added and deleted. Figure A.64 is a UML of actualization.

Table A.96 Change 9 Actualization Summary

Title	Code Files					
	Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
Date Search Bug	7	6	0	0	1	0

Table A.97 Change 9 Actualization Code Files

#	Code File	Task	Lines of Code		
			Added	Deleted	Total
1	`DatePanel`	Changed method	6	4	10
2	`DateField`	Changed method	1	2	3
3	`DateOption`	Added method	8	1	9
4	`DatePanelTest`	Changed test	10	0	10
5	`DateOptionTest`	Added test	14	0	14
6	`DateSearchFuncTest`	Added test	11	0	11

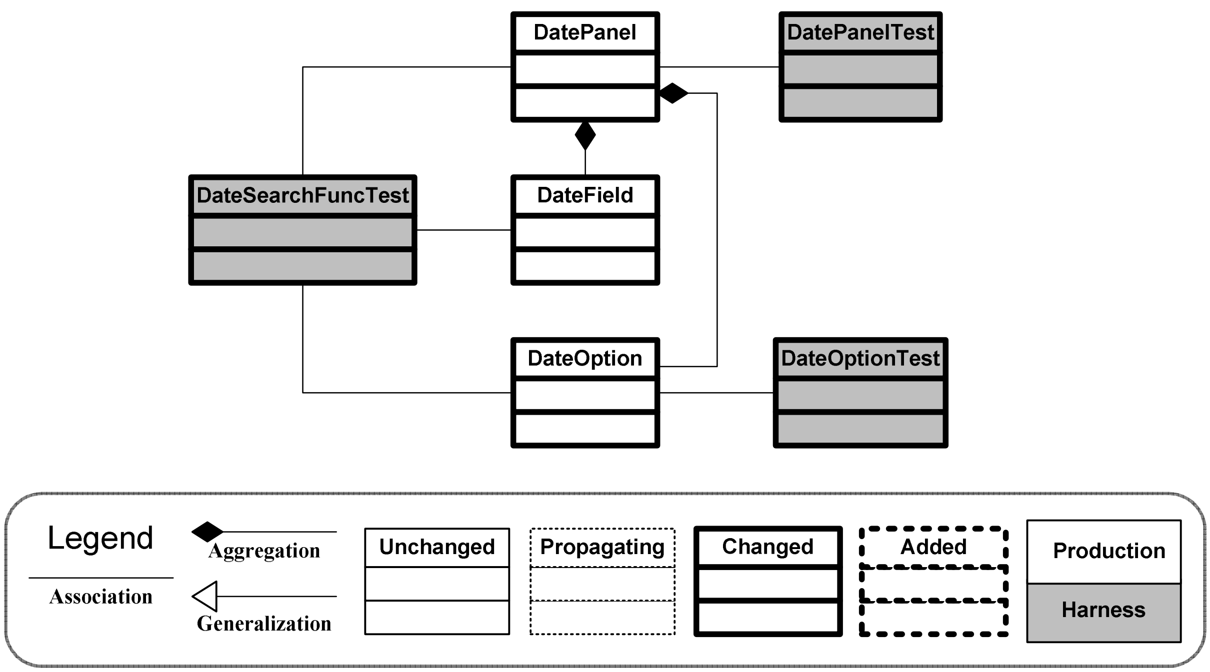


Figure A.64 Change 9 Actualization UML

A.9.5.1 DatePanel class

The programmer added the existing objects of `DateOption` to the `ActionListener` of `JCheckBox` field `dateBox` in the constructor.

A.9.5.2 DateField class

The `focusLost()` method had called the `dateTextCheckBox()` method, but it was redundant, so the programmer removed it.. He also added a condition to the if statement to only call the `setText()` method, if the `DateField` object is enabled.

Either of these conditions could cause the `DateOption` object to be added back to the `SearchManager` incorrectly. During change request 5, the programmer was probably trying to address these conditions when he introduced the bug.

A.9.5.3 DateOption class

The `ActionManager` interface and its `actionPerformed()` method was added to this class. Objects of this class are added to the `dateBox` `JCheckBox` field in `DatePanel`; when the box is selected, the `actionPerformed()` method calls the class's `setEnabled()` method with `dateBox`'s `isSelected()` method as a parameter.

A.9.5.4 DatePanelTest class

This class is the unit test suite for the `DatePanel` class. It had 1 test changed.

A.9.5.5 DateOptionTest class

This class is the unit test suite for the `DateOption` class. It had 1 test added.

A.9.5.6 DateSearchFuncTest class

This class is a functional test for date search functionality. It had 1 test added.

A.9.6 Postfactoring

No Postfactoring was necessary for this change request.

A.9.7 Verification

All regression tests passed after the change request. No new bugs were found. Table A.98 shows the test coverage of the production code files changed.

Table A.98 Change 9 Statement Verification

#	Code File	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	`DatePanel`	62	61	98.4	0	0
2	`DateField`	68	64	94.1	0	0
3	`DateOption`	21	21	100.0	0	0

A.9.8 Timing Data

Table A.99 contains the timing data for the change.

Table A.99 Change 9 Timing Totals

Phase	Time (hh:mm)
Concept Location	0:00
Impact Analysis	0:00
Prefactoring	0:00
Prefactoring Testing	0:00
Actualization	0:23
Actualization Testing	0:22
Postfactoring	0:00
Postfactoring Testing	0:00

A.9.9 Conclusions

This bug fix went smoothly. The `focusLost()` method of `DateField` had a redundant call to the `dateTextCheckBox()` method, which caused the fix to take slightly longer than planned. However, it was quickly found and fixed for a successful bug fix.

Table A.100 lists the totals for each set of code files for each change request of this iteration to date. Table A.101 is the current product backlog.

Table A.100 Change 9 Code File Summary

#	Change	Number in Code Files						
		Visited Concept Location	Estimated Impact Set	Changed Set	Added during			Total Project
					Pre	Act	Post	
0	Original Baseline	N/A	N/A	N/A	N/A	N/A	N/A	1,070
1	Basic Search	5	3	4	0	4	0	1,074
2	Recursive search	0	3	4	4	0	5	1,083
3	Advanced Output	6	21	11	2	4	10	1,099
4	Date Search	0	13	12	2	16	3	1,120
5	Case Sensitive	0	16	15	8	2	3	1,133
6	Extension Search	0	11	6	2	7	(5)	1,137
7	Properties Search	0	7	7	0	11	6	1,154
8	Directory Chooser Bug	0	3	3	0	0	0	1,154
9	Date Search Bug	0	7	6	0	0	0	1,154

Table A.101 Change 9 Current Product Backlog

#	Title	Complete	User Story
1	Basic Search	x	Add a basic search function that allows a user to search in the current directory for all or part of the title of a folder or file, and return a list of the matching files and directories.
2	Recursive Search	x	Add the ability to search inside all directories.
3	Advanced Output	x	Change the output to a table similar to the main muCommander window.
4	Date Search	x	Allow the user search by a date of file's modification.
5	Case Sensitive Search	x	Add capability to search by case sensitive search terms.
6	Extension Search	x	Add the ability to search for files with specific extensions.
7	Properties Search	x	Add options to search for files based on their properties.
8	Directory Chooser Bug	x	Choosing a directory with the file chooser doesn't update the search directory.
9	Date Bug	x	DateOption is not removed when disabled.
10	Size Search		Add the ability to search for a file by its size.
11	Regular Expression Search		Add capability to search by a regular expression.
12	Lucene Search		Incorporate the Apache Lucene search.
13	JDayChooserTest Bug		The test testSetMonth() fails on last day of month, if next month has fewer days

APPENDIX B.

Defect Log

This appendix contains Table B.1the defect log at the end of the SIP iteration.

Table B.1 Defect Log

Found Date	Time	Task	Location	Description	Origin Date	Origin Task	Fixed
2/13	8:25	Act	`DirectoryPanel`	Blank directory throws uncaught Exception	2/13	Act	2/13
2/13	9:17	Act	`SearchThread`	Inaccessible directory throws Exception	2/13	Act	
2/13	9:34	Act	`SearchDialog`	No results not showing up	2/13	Act	
2/27	7:55	Act	`SearchTable`	Results not showing up in table	2/27	Act	2/27
3/4	3:34	Post	`SearchTableModel`	Shows parent name if searching root	3/4	Post	3/4
3/14	1:25	Act	`DatePanel`	Search results outside of date range	3/14	Act	3/14
3/14	1:47	Act	`DatePanel`	Two digit years show up as 1st century	3/14	Act	3/14
3/28	10:23	IA	`DatePanel.datePanelSetEnabled()`	`DateOption` not removed when disabled	3/14	Act	6/23
3/31	4:25	Post	`JDayChooserTest.testSetMonth()`	Fails on last day of month, if next month has fewer days	3/14	Act	
3/31	4:07	Post	`SearchTermOption.insertUpdate()`	Empty string in `searchTermBox` throws Exception	3/31	Post	3/31
3/31	4:34	Post	`SearchTermOption`	Case lost on `searchTerm` when switching between case sensitive/insensitive	3/31	Post	3/31
4/8	2:32	Act	`DirectoryPanel`	Choosing a directory with the file chooser does not update the search directory	3/17	Pre	6/22

APPENDIX C.

Glossary of Terms

This appendix is a list of terms used in the thesis.

Actualization/Postfactoring Code Files Changed: Any code file modified during the phase; this may include code files that were created during an earlier phase of the change that are not included in the changed set.

Production Code File: A code file as defined in this document that is not a harness code file.

Changed Set: The set of code files that existed before the change and were modified during any phase of the change.

Code file: When used in a table or count, such as “the set of code files was 12” the term code file refers to a file that contains at least one class, enum or interface. If a code file contains multiple classes, enums or interfaces or some combination of these, it will be counted as 1 code file.

Harness: Any code that is a test or stub or simulation.

Harness Code file: Any code file that contains exclusively harness.

Lines of Code: Line of code (LOC) refers to non-comment lines of code (NCLOC) which is any single line of code, that does not start will a comment or is a blank line. The added and deleted numbers are all derived from a program DiffStats written for the project.

Testing Coverage: The verification section lists test coverage by code file. It lists the coverage of the production code files written during the iteration by the entire test suite.

The production code files that existed in muCommander at the start of the SIP iteration are not listed. At that time, it was deemed that the existing muCommander regression tests were adequate. This means that if refactoring is done to an existing muCommander and the regression tests pass, the refactoring is deemed to be of adequate quality. If evidence is found during the iteration that the test regression test suite is inadequate, a change to improve the regression test suite will be added to the product backlog for the code file as a protective change.

C.1 Class change table terms

These terms are used in the Prefactoring, Actualization and Postfactoring Code File tables in Appendix A.

Added class: This class was added to the project.

Removed class: This class was removed from the project.

Moved class: This class was moved from one package to another.

Renamed class: This class had its name changed.

Extracted class: This class was created in this phase by a class extraction.

Extracted class from: One or more classes were extracted from this class.

Merged class: This class was merged into another class.

Merged class to: A class was merged into this class.

Extracted super class: This abstract class was created in this phase by a super class extraction.

Extracted super class from: One abstract class was extracted from this class.

Added method: One or more methods were added to the class.

Changed method: One or more methods were changed in the class.

Deleted unused method: One or more methods that were never called were deleted.

Extracted method: One or more methods had partial responsibility extracted to a method from another method.

Renamed method: One or more methods in this class had their names changed.

Moved method: One or more methods were moved to this class.

Moved method from: One or more methods were moved from this class.

Removed method: One or more methods were deleted from this class.

Renamed field: One or more fields were renamed.

Extracted field: One or more fields were extracted from method variables.

Changed Field: One or move fields changed type.

Moved Field: One of more fields were moved to this class.

Moved Field from: One or more were moved from this class.

Changed variable type: One or more temporary variables' type changed.

Added cast: One or more method calls were cast.

Extracted constant: One or more constants were extracted from method variables.

Added code block: One or more static code blocks were added.

Added test: One or more tests were added to the class.

Changed test: One or more tests were changed in the class.

Extracted test: One or more tests had partial responsibility extracted to a test from another test.

Renamed test: One or more tests in this class had their names changed.

Moved test: One or more tests were moved to this class.

Moved test from: One or more tests were moved from this class.

Removed test: One or more tests were deleted from this class.

Javadoc: The Javadoc of this class was updated.

REFERENCES

[1] Rajlich, V. *Software Engineering: The Contemporary Practice*. To be published by CRC Press, 2011.

[2] Brooks, F. P. *The mythical man-month: essays on software engineering*. Addison-Wesley Reading, MA, 1995.

[3] Fowler, M. and Highsmith, J. The agile manifesto. *Software Development*, 9, 8 (August 2001), 28-35.

[4] Fowler, M. *AgileSignatory*. http://martinfowler.com/bliki/AgileSignatory.html, Date Accessed: June 29, 2011.

[5] Matthews, O. and Howell, G. A. Integrated project delivery: an example of relational contracting. *Lean construction journal*, 2, 1 (April 2005), 46-61.

[6] Lehman, M. M. and Ramil, J. F. Software evolution and software evolution processes. *Annals of Software Engineering*, 14, 1 (December 2002), 275-309.

[7] Febbraro, N., Rajlich, V. The Role of Incremental Change in Agile Software Processes. In *Proceedings of the Agile Conference* (Washington, D.C., 13 - 17 August, 2007). Agile 2007, 2007.

[8] Rajlich, V. and Gosavi, P. Incremental change in object-oriented programming. *Software, IEEE*, 21, 4 (July/August 2004), 62-69.

[9] Humphrey, W. *Introduction to the personal software process (sm)*. Addison-Wesley, Reading, MA, USA, 1996.

[10] Humphrey, W. S. Using a defined and measured personal software process. *Software, IEEE*, 13, 3 (May 1996), 77-88.

[11] Ferguson, P., Humphrey, W. S., Khajenoori, S., Macke, S. and Matvya, A. Results of applying the personal software process. *Computer*, 30, 5 (May 1997), 24-31.

[12] Johnson, P. M. and Disney, A. M. A critical analysis of PSP data quality: Results from a case study. *Empirical Software Engineering*, 4, 4 (December 1999), 317-349.

[13] Schwaber, K. Scrum development process. In *Proceedings of the OOPSLA'97 Business Object Workshop* (Atlanta, Georgia, USA, 6 October, 1997). Citeseer, 1997.

[14] Rising, L. and Janoff, N. S. The Scrum software development process for small teams. *Software, IEEE*, 17, 4 (July/August 2000), 26-32.

[15] Martin, R. C. Professionalism and test-driven development. *IEEE Software*, 24, 3 (May/June 2007), 32-36.

[16] Müller, M. M. and Tichy, W. F. Case study: extreme programming in a university environment. In *Proceedings of the 23rd International Conference on Software Engineering* (Toronto, Ontario, Canada, 2001). IEEE Computer Society, 2001.

[17] Cockburn, A. and Highsmith, J. Agile software development, the people factor. *Computer*, 34, 11 (November 2001), 131-133.

[18] Marcus, A., Rajlich, V., Buchta, J., Petrenko, M. and Sergeyev, A. Static techniques for concept location in object-oriented code. In *Proceedings of the 13th International Workshop on Program Comprehension* (St. Louis, Missouri, USA, 15-16 May, 2005). IEEE, 2005.

[19] Chen, K. and Rajlich, V. Case study of feature location using dependence graph. In *Proceedings of the 8th International Workshop on Program Comprehension* (Limerick , Ireland, 10 - 11 June, 2000). IEEE, 2000.

[20] Ren, X., Chesley, O. C. and Ryder, B. G. Identifying failure causes in java programs: An application of change impact analysis. *IEEE transactions on software engineering*, 32, 9 (September 2006), 718-732.

[21] Han, J. Supporting impact analysis and change propagation in software engineering environments. In *Proceedings of the Eighth IEEE International Workshop on incorporating Computer Aided Software Engineering]* (London , UK 14 - 18 July, 1997), 1997.

[22] Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. R. *Improving the design of existing code*. Addison–Wesley, 1999.

[23] Fowler, M. *Refactoring*. http://refactoring.com/, Date Accessed: June 6, 2011, 2011.

[24] Mens, T. and Tourwé, T. A survey of software refactoring. *Software Engineering, IEEE Transactions on*, 30, 2 (Feb 2004), 126-139.

[25] Buckner, J., Buchta, J., Petrenko, M. and Rajlich, V. JRipples: A tool for program comprehension during incremental change. In *Proceedings of the 13th International Workshop on Program Comprehension* (IWPC, 15-16 May, 2005). IEEE, 2005.

[26] Gamma, E. and Beck, K. JUnit: A cook's tour. *Java Report*, 4, 5 (August 1999), 27-38.

[27] Thornton, M., Edwards, S. H., Tan, R. P. and Pérez-Quiñones, M. A. Supporting student-written tests of gui programs. In *Proceedings of the 39th SIGCSE technical*

symposium on Computer science education (Portland, OR, USA, 12-15 March 2008). ACM, 2008.

[28] Yang, Q., Li, J. J. and Weiss, D. A survey of coverage based testing tools. In *Proceedings of the 2006 international workshop on Automation of software test* (Shanghai, China, 23 May, 2006). ACM, 2006.

[29] Petrenko, M., Poshyvattyk, D., Rajlich, V. and Buchta, J. Teaching software evolution in open source. *Computer*, 40, 11 (November 2007), 25-31.

[30] The Linux Information Project *Inode Definition*. http://www.linfo.org/inode.html, Date Accessed: June 28, 2011.

[31] Bernard, M. *MuCommander*. http://www.mucommander.com/, 2011.

[32] Beck, K. *JUnit*. http://JUnit.org, 2011.

[33] Oracle *Swing (Java Foundation Classes)*. http://download.oracle.com/javase/6/docs/technotes/guides/swing/, 2011.

[34] The Eclipse Foundation *Eclipse*. http://www.eclipse.org/, 2011.

[35] Petrenko, M. *JRipples*. http://jripples.sourceforge.net/, 2011.

[36] Atlassian Pty Ltd. *Clover Java Code Coverage & Test Optimization*. http://www.atlassian.com/software/clover/, 2011.

[37] The Eclipse Foundation *Mylyn*. http://www.eclipse.org/mylyn/, 2011.

[38] Tasktop Technologies *Tasktop*. http://tasktop.com/, 2011.

[39] Wall, T. *Abbot Java GUI Test Framework*. http://abbot.sourceforge.net/doc/overview.shtml, 2008.

[40] Apache Software Foundation *Subversion*. http://subversion.apache.org/, 2011.

[41] The TortoiseSVN Team *TortoiseSVN*. http://tortoisesvn.net/, 2011.

[42] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design patterns: elements of reusable object-oriented software*. Addison-wesley Reading, MA, 1995.

[43] Kumar, S. *ComponentTitledBorder*. http://www.jroller.com/santhosh/entry/component_titled_border, 2005.

[44] Toedter, K. *JCalendar*. http://www.toedter.com/en/jcalendar/, 2009.

[45] Van Deursen, A., Moonen, L., van den Bergh, A. and Kok, G. *Refactoring test code*. Technical Report 869201, Citeseer, Amsterdam, The Netherlands, 2001.

[46] Humphrey, W. S. Why don't they practice what we preach? *Annals of Software Engineering*, 6, 1 (March 1998), 201-222.

[47] Boehm, B. and Basili, V. R. Software defect reduction top 10 list. *Software engineering: Barry W. Boehm's lifetime contributions to software development, management, and research*, 34, 1 (January 2007), 75.

[48] Johnson, P. M., Kou, H., Agustin, J., Chan, C., Moore, C., Miglani, J., Zhen, S. and Doane, W. E. J. Beyond the Personal Software Process: Metrics collection and analysis for the differently disciplined. In *Proceedings of the 25th International Conference on Software Engineering* (Portland, Oregon, 2003). IEEE Computer Society, 2003.

ABSTRACT

AN EXPERIENCE REPORT OF THE SOLO ITERATIVE PROCESS

by

CHRISTOPHER DORMAN

August 2011

Advisor: Dr. Václav Rajlich

Major: Computer Science

Degree: Master of Science

The field of software engineering is over 50 years old; originally, mathematicians and engineers thought software development was more of an art form than a defined process. These first software engineers managed to produce a variety of complex, working software; however, many software engineers today use agile processes. This thesis is an experience report in an agile process called the *Solo Iterative Process*.

In this thesis, previous research is reviewed in previous solo processes, team processes, individual phases of software evolution and software evolution tools. Then the Solo Iterative Process is defined. To begin the experience report a subject software, a change request and the tools and technologies are identified. Then 9 change requests are performed on the subject software. The discussion looks at matters of individual phases that occur over a set of change requests.

AUTOBIOGRAPHICAL STATEMENT

Christopher Dorman is and shall forever be the author of this thesis.