

<http://researchcommons.waikato.ac.nz/>

Research Commons at the University of Waikato

Copyright Statement:

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

The thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of the thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from the thesis.



THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Spatial Hypermedia as a Programming Environment

Bryce Nemhauser

This thesis is submitted in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy at the University of Waikato.

January 2018
© 2018 Bryce Nemhauser

Abstract

This thesis investigates the possibilities opened to a programmer when their programming environment not only utilises Spatial Hypermedia functionality, but embraces it as a core component. Designed and built to explore these possibilities, SpIDER (standing for **S**patial **I**ntegrated **D**evelopment **E**nvironment **R**esearch) is an IDE featuring not only traditional functionality such as content assist and debugging support but also multimedia integration and free-form spatial code layout. Such functionality allows programmers to visually communicate aspects of the intent and structure of their code that would be tedious—and in some cases impossible—to achieve in conventional IDEs.

Drawing from literature on Spatial Memory, the design of SpIDER has been driven by the desire to improve the programming experience while also providing a flexible authoring environment for software development. The programmer's use of Spatial Memory is promoted, in particular, by: utilising fixed sized authoring canvases; providing the capacity for landmarks; exploiting a hierarchical linking system; and having well defined occlusion and spatial stability of authored code.

The key challenge in implementing SpIDER was to devise an algorithm to bridge the gap between spatially expressed source code, and the serial text forms required by compilers. This challenge was met by developing an algorithm that we have called the flow walker. We validated this algorithm through user testing to establish that participants' interpretation of the meaning of spatially laid out code matched the flow walker's implementation.

SpIDER can be obtained at: <https://sourceforge.net/projects/spatial-ide-research-spider>

Acknowledgements

I would like to thank the following members of the University of Waikato computer science department for their assistance in completing this thesis. Without their help and guidance I would not have been able to complete this task.

- **Supervisors.** David Bainbridge and Bill Rogers. Their assistance and direction throughout the development of SpIDER and subsequent thesis write-up was invaluable. I will never be able to repay them for the countless ‘longer than planned’ meetings they sat through. I hope their families forgive their repeated tardiness to dinner.
- **Previous Supervisors.** Angela Martin and Craig Taube-Schock. While only present for a limited time, Angela and Craig helped direct my research at critical moments. They were also a fountain of useful knowledge.
- **Consultation and Review.** Robert Akscyn, Sally Jo Cunningham and David Nichols. Each has been a great help by assisting with early reviews of my thesis, removing a number of spelling, grammatical and logical errors. Rob has been an ever-flowing fountain of knowledge for all things Expeditee. Sally Jo has given me the opportunity to present my research and was helpful in acquiring participants for studies. David has challenged my thinking with pointed questions, pushing me to investigate questions I would have otherwise missed.

I would also like to make a blanket thankyou to all other members of the computer science department: lecturers, office staff, teaching support staff and technical support alike. You have all contributed to making this department an enjoyable place to work.

Finally I would like to thank my friends and family—you know who you are. Without your support and patience I doubt I would have gotten this far.

Contents

Chapter 1 Introduction	1
1.1 Issues with Software Development Tools.....	3
1.2 Context.....	4
1.3 Spatial Hypermedia as a Solution	6
1.4 Thesis Outline.....	12
Chapter 2 Spatial Memory for Software Development	15
2.1 Spatial Memory.....	16
2.1.1 Short-Term Memory	16
2.1.2 Long-Term Memory	18
2.1.3 Object Location and Navigation.....	18
2.2 Single View and Viewport Interfaces	19
2.2.1 Single View Interfaces	20
2.2.2 Viewport Spatial Interfaces.....	21
2.2.3 Overviews.....	22
2.2.4 Landmarks.....	23
2.3 Spatial Stability.....	24
2.3.1 Scrolling.....	24
2.4 Programmers Navigating with Spatial Memory.....	27
2.4.1 Eclipse Package Explorer	27
2.4.2 Eclipse Outline.....	28
2.4.3 Program Code	30
2.4.4 Software Visualisation.....	32
2.5 Utilising Spatial Memory for Programming	34
2.5.1 Applying Single View and Viewport Interface Concepts to Authored Content	34
2.5.2 The Shape of Code and Other Media – Landmarks	35

2.5.3 Spatial Stability of Authored Content	36
Chapter 3 Traditional Programming	37
3.1 Traditional Integrated Development Environments	38
3.1.1 The Emergence of Integrated Development Environments	38
3.1.2 Integrated Development Environment Functionality	42
3.2 Abstractions in Programming.....	45
3.2.1 Abstractions in Planning.....	45
3.2.2 Abstractions in Code	47
3.3 The Rigidity of Traditional IDEs	48
Chapter 4 Editing Environments and Authoring	51
4.1 Defining Spatial Hypermedia.....	51
4.1.1 Absolute and Relative Space	52
4.2 Content and Meaning.....	53
4.2.1 Fundamental Elements, System Representations and First Class Citizens	54
4.2.2 System Representation as Applied to Meaning	62
4.2.3 Content and Meaning Discussion.....	64
4.3 General Purpose Spatial Hypermedia	66
4.3.1 VIKI	68
4.3.2 VKB	75
4.3.3 Expeditee.....	78
4.3.4 General Purpose Spatial Hypermedia Discussion	88
4.4 Summary	91
Chapter 5 Authoring in Spatial Hypermedia IDE Environments	93
5.1 Spatial Hypermedia in IDEs	94
5.1.1 Code Thumbnails.....	95
5.1.2 Code Canvas	98

5.1.3 Code Bubbles	103
5.1.4 Debugger Canvas	108
5.1.5 Spatial Hypermedia in IDEs Discussion	114
5.2 Discussion of Relevant Matters Arising	115
5.2.1 Spatial Memory.....	116
5.2.2 Multimedia.....	120
5.2.3 Rigidity	121
5.2.4 Occlusion.....	122
5.2.5 Collaboration.....	124
5.3 Development Direction.....	126
5.3.1 IDE into Spatial Hypermedia	126
5.3.2 Maximising the use of Spatial Memory	128
5.3.3 Summary	130
Chapter 6 Expeditee In-Depth.....	131
6.1 The Goal of Expeditee	132
6.2 User Interface Structure	135
6.2.1 Frame Title and Name.....	136
6.2.2 Message Bay	137
6.3 Creating and Manipulating the Frame and Linking System	138
6.3.1 Linking to Existing Frames.....	139
6.3.2 Creating New Frames.....	140
6.3.3 FrameSets	140
6.3.4 Frame Navigation.....	141
6.4 Text Entry in Expeditee	141
6.4.1 Adding and Modifying Text Content.....	141
6.4.2 Annotations.....	142

6.4.3 User Scripting and Actions	146
6.5 Creating Polylines and Polygons	147
6.6 Summary: Expeditee In-Depth	148
Chapter 7 Usage and Implementation of SpIDER.....	151
7.1 Conveying Meaning in SpIDER	151
7.1.1 Inferring Lines.....	152
7.1.2 Boxing.....	154
7.1.3 Out of Flow	157
7.1.4 Out of Flow Chaining.....	158
7.1.5 A Multitude of Ways	159
7.1.6 Frames and Linking.....	161
7.2 Authoring IDE Functionality	163
7.2.1 Syntax Highlighting.....	163
7.2.2 Warnings and Errors.....	164
7.2.3 Content Assist.....	167
7.2.4 Java Projects and SpIDER State	168
7.3 Running Java Programs	172
7.3.1 Executing without Breakpoints	172
7.3.2 Executing with Breakpoints.....	174
7.4 The Flow Walker Algorithm.....	179
7.4.1 Within Frame Component.....	180
7.4.2 Director Component	188
7.5 The Magnet System.....	191
7.6 Integration with the Eclipse Java Development Tools	195
Chapter 8 Evaluating SpIDER Spatial Layout.....	199
8.1 Initial Study.....	199

8.1.1 Study Results	202
8.1.2 Analysis of Initial Study Results.....	203
8.1.3 Summary of Initial Study	215
8.2 Follow up Study.....	216
8.2.1 Study Design.....	216
8.2.2 Questions and Analysis	217
8.3 Summary of Studies Evaluating SpIDER Spatial Layout	222
Chapter 9 Spatial Development Patterns	225
9.1 Maintaining Existing Functionality.....	227
9.1.1 Authoring	228
9.1.2 Hierarchical Content	230
9.1.3 Pagination	232
9.1.4 Controls	235
9.1.5 Overlaid Feedback.....	236
9.2 Expressive Patterns	238
9.2.1 In Situ Documentation	241
9.2.2 Escaping Hierarchical Structure	243
9.2.3 Emphasis	246
9.2.4 Logical Grouping	247
9.2.5 Alternative Forms of Abstraction.....	248
9.2.6 Alternative Handling of Containment	256
9.3 Process Patterns.....	259
9.3.1 Construction by means of Notes.....	259
9.3.2 Documenting History	263
9.3.3 Altering Flow	268
9.3.4 Communication Techniques	270

9.4 Summary: Applying Spatial Development Patterns	274
Chapter 10 Recommendations for the use of Spatial Layout.....	277
10.1 Ambiguous Code Layout	278
10.2 Positioning of Arrows	279
10.3 Abstraction by way of Frame	281
10.4 Summary: Recommendations for the use of Spatial Layout.....	287
Chapter 11 Conclusion	289
11.1 Summary and Discussion	290
11.1.1 Literature Summary	290
11.1.2 Future Work: Chunking	292
11.1.3 Design, Usage and Development of SpIDER.....	294
11.1.4 Evaluation.....	295
11.1.5 Future Work: Snap to Spot Feature	298
11.2 Hypotheses Revisited	299
11.3 Next Steps	301
11.3.1 Python	301
11.3.2 Inheritance	302
11.3.3 Spatial Programming Languages	303
References	305
Appendix A Evaluating SpIDER Spatial Layout: Questions.....	313
A.1 Questions from Initial Study	313
A.1.1 Part 1	313
A.1.2 Part 2.....	314
A.1.3 Part 3.....	316
A.1.4 Part 4.....	317
A.1.5 Part 5.....	319

A.1.6 Part 6.....	321
Appendix B Evaluating Spatial Layout: Confidence Level.....	323
B.1 Initial Study Confidence Level.....	323
Appendix C Ethics Application and Approval	325
C.1 Ethics Application.....	325
C.2 Ethics Approval	328

Chapter 1

Introduction

When attempting to understand an algorithm, do you fare better if it is explained diagrammatically or if you are given the straight code to read? What if you were given both? What about when you are developing an algorithm. Do you sketch it out on paper first? Even a cursory review of algorithm textbooks shows that diagrams play an important role in explanations. Despite this, we develop code primarily in tools that limit us to text authoring. This thesis documents the design, development and evaluation of an approach that seeks to address this disconnect, manifest through a software tool we have called SpIDER. Standing for Spatial Integrated Development Environment Research, SpIDER provides programmers with the capability to express executable code in a diagrammatic manner, including multimedia elements such as pictures to further promote code understanding. This is achieved while retaining functionality that programmers have come to expect in Integrated Development Environments (IDEs), such as continuous compilation, debugging and content assist.

To give an indication of this new way of writing code, Figure 1.1 shows a screenshot taken from SpIDER, featuring functioning code, which centres on modelling the lanes vehicles can use in a traffic simulation program. Examples such as Figure 1.1 are used throughout the thesis to illustrate the diversity of code structures supported by SpIDER. We return to this particular example in Chapter 9 to discuss it fully.

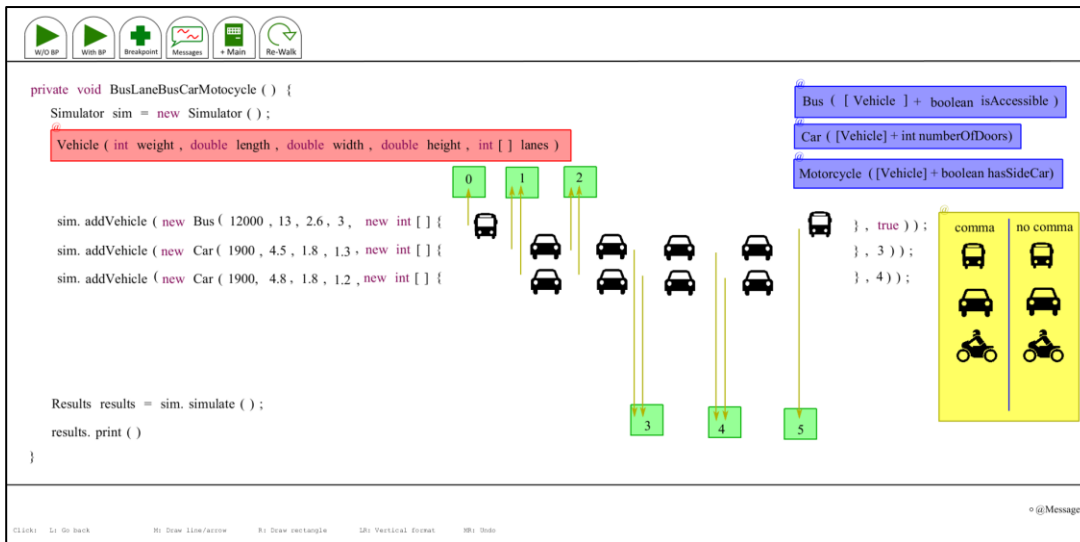


Figure 1.1: Example of functioning code produced in SpIDER, later discussed in Section 9.3.

In truth, all but the most trivial programs are a maze of logically interconnected elements that makes software difficult to understand despite our best intentions. Programming language block structures create nested hierarchies. Method calls and data access often work between the branches of these hierarchies, forming cross-cutting linkages; or even between projects, as in the case of units and their unit test code. Beyond the confines of the programming language syntax, software projects also contain heterogeneous associations, such as links from code to planning documents. Sometimes, these connections can even be between digital and physical artefacts, from the computer to the office whiteboard for example.

Many techniques exist for handling the complexity of software development. Programming paradigms provide sets of concepts designed to make certain types of programs easier to express, for example: Object-oriented code encourages programmers to encapsulate information into objects and has the programmer design their program around the interactions between these objects; Functional languages attempt to limit side effects and use immutable data which helps keep each piece of code self-contained; and Event-driven languages determine the flow of a program by reacting to received messages like user input.

Programming methodologies help by managing the development process of a piece of software and encourage (or stipulate) certain practices. Programmers following Extreme

Programming might for example: construct prototypes, program in pairs to improve the quality of code, and refactor completed code to make it more manageable.

Integrated development environments (IDEs) bring many core tools that a programmer finds useful into one piece of software. Some tools help with producing code. Syntax highlighting, error reporting and content assist (also known as code completion) all help ease cognitive load while editing programs. They allow the programmer to concentrate more on the intellectual task at hand, and worry less about the specifics of the language they are using. Other tools within IDEs simplify the compile, run, test, debug cycle by allowing the programmer to perform all these functions in a single workflow. This avoids, for example, the necessity of having to manually relate tool output, such as error message line numbers, to source code.

1.1 Issues with Software Development Tools

In addition to editing and program execution support, IDEs provide a variety of tools for particular prescribed tasks. For example, Eclipse provides the user with a window called the Outline for navigating through the high-level hierarchy of the currently active class. The Eclipse Outline provides buttons for:

- Sorting class members in alphabetical order or in the order they occur in the file.
- Hiding or showing the fields.
- Hiding or showing static members.
- Hiding or showing non-public members.
- Hiding or showing local types.

These are all potentially useful options, however they are all options that have been manually built into the tool by Eclipse developers. If a programmer wanted to sort the Outline in any other way then they must either: convince an Eclipse developer that it would be worthwhile implementing the desired functionality, and then wait for it to happen; make the change themselves in a private copy; or do without the change. If the desired sort was something simple but specific (for example sorting alphabetically but with functions whose name starts with 'Execute' appearing at the head of the list) then the last is by far the most likely outcome.

A close study of IDEs reveals that this prescribed approach pervades the interface components provided: from the package explorer, through the creation of projects, to the exploration of a running program in the debugger. It is not even possible to include diagrams alongside code or mix font sizes. You are only able to alter the environment when the developers have thought to allow a specific modification.

So, need IDEs be so inflexible? A programmer cannot manually rearrange the Eclipse Outline because there is not a large enough demand for the functionality, however, if the widget allowed for modifications in the same fashion as the text area then programmers could arrange content as it suited on a case-by-case basis. In comparison to IDEs, other editing software is more flexible. Microsoft Word, for example, allows you to include images alongside your text and control the font size for individual tokens. These are features that a programmer may find useful but are not present in the popular IDEs of today. People acknowledge that programming is a problem-solving task and that IDEs help with this. People also acknowledge that programming is a creative task. Would a more expressive environment better support programmers? We contend that it would. This thesis explores what can be achieved by applying the expressive and flexible features of Spatial Hypermedia editing to the programming domain.

1.2 Context

This thesis builds upon three principal topics drawn from the literature: Spatial Memory, IDEs and Spatial Hypermedia. We discuss what Spatial Memory is and how it can be leveraged; the history that has led to today's IDEs and how they can be further improved; what Spatial Hypermedia is, how it differs from Hypermedia in general and how it has previously been utilised, both generally and for programming.

Spatial Memory. Spatial Memory is a field in cognitive psychology concerned with the way in which people navigate environments and remember the locations of objects. Chapter 2 discusses how Spatial Memory is automatic and benefits from practice. It is for this reason that we seek to allow programmers to exploit their spatial memory to improve the software development process. Concepts such as landmarks, overviews and the fixed or variable size of application windows are discussed as ways of affecting an individual's utilisation of Spatial Memory. We also document some research that shows that programmers are both willing and able to use their Spatial Memory to assist in programming.

IDEs. An historical approach is taken to examining the development of IDEs, culminating in popular IDEs such as Visual Studio and Eclipse. As part of this, the early adoption of tools that accelerate the programming task are documented. We establish a core set of functionality that programmers have come to expect in an IDE. The role of abstraction in programming is also discussed. This allows us to refine the question previously stated in Section 1.1: need IDEs be so inflexible? Can IDEs be more flexible?

Spatial Hypermedia. Spatial Hypermedia is a form of authoring application with a novel user interface and interaction set. This user interface and interaction set is more capable than a standard user interface at utilising Spatial Memory. Chapter 4 properly defines Spatial Hypermedia, comparing it to other forms of authoring software systems. Several Spatial Hypermedia systems are documented and examined, some laying in the programming domain.

Towards the end of Chapter 5 we expand on the Spatial Hypermedia literature by discussing the benefits and issues with specific approaches to developing Spatial Hypermedia. For example, are we better off retrofitting Spatial Hypermedia functionality into an existing IDE or adding IDE functionality to an existing Spatial Hypermedia application?

1.3 Spatial Hypermedia as a Solution

The review of these three areas of literature—Spatial Memory, IDEs and Spatial Hypermedia—assisted in the design and development of a Spatial Hypermedia based IDE that we call SpIDER. Four prominent features of SpIDER are:

- **Spatial Code Layout:** Providing a system for laying code out on fixed sized (see Section 2.5) canvases so that programmers may use layout to spatially communicate information. If a programmer desires, they are able to use this functionality to blur the lines between planning diagrams and functioning code.
- **Linking:** The ability to link canvases together in an ad hoc fashion to form a web of relationships between distinct pieces of content.
- **Informal Abstractions:** The utilisation of linking or specific multimedia objects to create abstractions distinct from the programming language.
- **Multimedia Elements:** Allowing for embedding hypermedia elements such as pictures and diagrams so that planning and development can be more tightly coupled.

SpIDER provides programmers with an authoring environment reminiscent of a canvas, lacking a scrollbar. This encourages careful consideration of how the code is written and organised, by breaking it down into logical blocks (chunks) laid out spatially and nested in a manner that can extend the syntax that the programming language allows. This new way of writing code is serialised into the form the compiler is expecting through an algorithm called the flow walker. The SpIDER canvas is referred to as a Frame. Elements on a SpIDER Frame are referred to as Items.

Spatially Laying out Code. Code can be spatially laid out to convey meaning. A major boon of laying code out spatially in a fixed sized application window is the increased opportunity to provide for the use of Spatial Memory. Figure 1.2 shows a screenshot taken from Eclipse showing the code for a small class containing a single static function. The code produces an array of a specified size and fills it with random numbers, capped to a specified maximum value. The same code can be seen spatially laid out in SpIDER in Figure 1.3. Spatial layout has been used to separate and emphasise the code that does the random generation from the code that constructs the array. Furthermore, the parameter to the constructor of

Random has been placed in its own box in order to further highlight that a seed is supplied and make it easy to change at a later date if desired.

```
package Sorts;

import java.util.Random;

public class ListGenerator {
    public static int[] Generate(int size, int max) {
        int[] arr = new int[size];
        Random rand = new Random(1000);
        for(int i = 0; i < arr.length; i++) {
            arr[i] = rand.nextInt(max);
        }
        return arr;
    }
}
```

Figure 1.2: Code to produce a list of specified size with random elements in Eclipse.

```
package Sorts ;
import java. util. Random ;
public class ListGenerator {

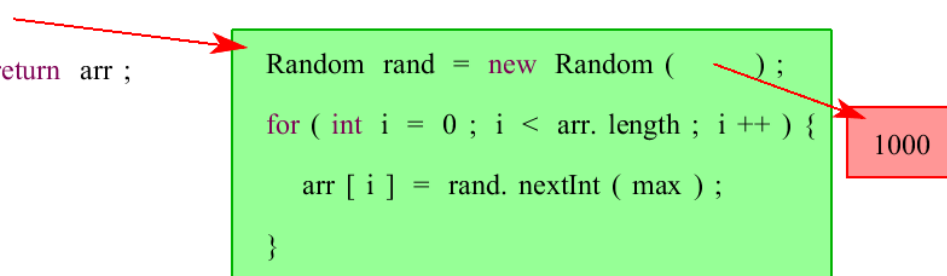
    public static int [ ] Generate ( int size , int max ) {

        int [ ] arr = new int [ size ] ;

        return arr ;

    }

}
```



```
Random rand = new Random (  ) ;
for ( int i = 0 ; i < arr. length ; i ++ ) {
    arr [ i ] = rand. nextInt ( max ) ;
}
```

1000

Figure 1.3: Code to produce a list of specified size with random elements in SpIDER.

Linking. Any Item in SpIDER can contain a hyperlink to another Frame. Items with links are denoted by a small hollow circle displaced to the left of the Item. The flow walker algorithm follows links when serialising the spatially laid out code.

Figure 1.4 shows another screenshot taken from Eclipse. This screenshot shows the entry point to the *QuickSort* class. As development is still occurring, the entry point currently contains code to test the *sort* function. It begins by gathering together information passed as arguments to the program along with user input to decide on the length of the list to generate and what the max element will be. The second step is it to generate an array using the *ListGenerator* class from Figure 1.3. The array is printed before and after sorting. The remainder of the *QuickSort* class has been excluded from the screenshot.

```
1 package Sorts;
2 import java.util.Scanner;
3 public class QuickSort {
4     public static void main(String[] args) {
5         //Get size of and max number in list
6         int size = Integer.parseInt(args[0]);
7         System.out.println("Producing a list of size: " + size);
8         System.out.println("What should the max number be?");
9         Scanner in = new Scanner(System.in);
10        int max = Integer.parseInt(in.next());
11        in.close();
12
13        int[] arr = ListGenerator.Generate(size, max);
14        System.out.println("Pre Sorted List");
15        for(int i: arr){
16            System.out.print(i + " ");
17            System.out.println();
18        }
19        int[] result = QuickSort.sort(arr);
20        System.out.println("Post Sorted List");
21        for(int i: result){
22            System.out.print(i + " ");
23            System.out.println();
24        }
25    }
```

Figure 1.4: Main function inside class *QuickSort* (Eclipse).

While this is a simple program, it can be broken down further using SpIDER. Figure 1.5 shows the same content produced in SpIDER. Links have been used to move content off-page where the developer has thought it useful to do so.

- The import statement has been moved off Frame so that it does not take up space on the current Frame. It is still accessible by clicking on the linked Item.
- The code establishing the size and max value in the randomly generated list to sort has also been moved to its own Frame.

As an aside, this is an example of the flexibility provided by SpIDER. While the first bullet point details an example that is similar to the functionality present in today's popular IDEs—the ability to collapse the import statements so that only one is visible—the second bullet point details an example where the programmer has achieved similar behaviour in an instance where the IDE is not aware of any meaningful grouping.

```
package Sorts ;  
◦imports  
public class QuickSort {  
    public static void main ( final String [ ] args ) {  
        ◦//Get size of and max number in list  
        int [ ] arr = ListGenerator. Generate ( size , max ) ;  
        System. out. println ( "Pre Sorted List: " ) ;  
        for ( int i : arr ) { }  
        int [ ] result = QuickSort. sort ( arr ) ;  
        System. out. println ( "Post Sorted List:" ) ;  
        for ( int i : arr ) { }  
    }  
    ◦SortMethod  
}
```

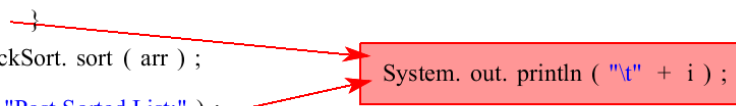


Figure 1.5: Main function inside class QuickSort (SpIDER).

Figure 1.6 is a diagram showing how the featured *QuickSort* program is broken up over multiple frames. Six Frames are present, each has been cropped to fit the dimensions of its content. Dashed arrows have been overlaid across each Frame arrived at by following the linked Items.

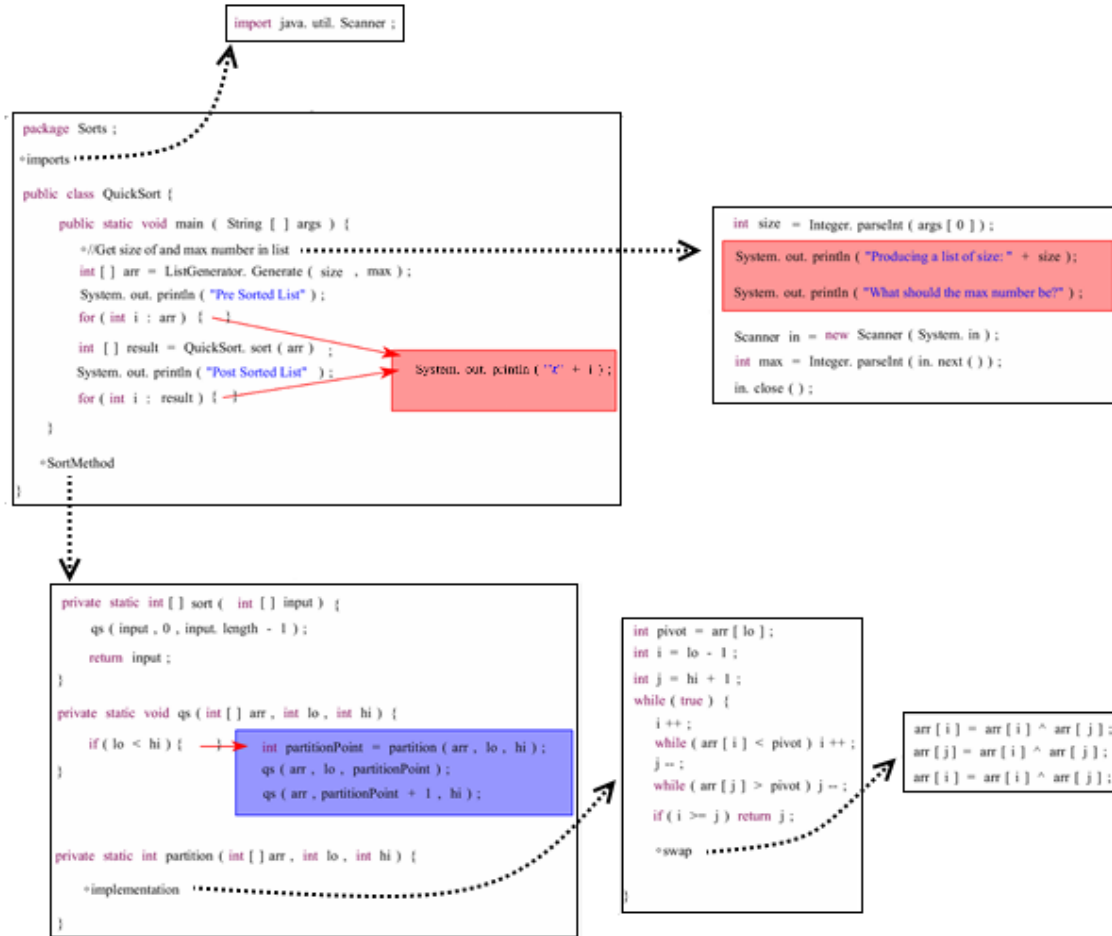


Figure 1.6: Example of linking relations between Frames.

Abstraction. Traditionally in programming languages, the smallest abstraction is considered a function, however in reality—even if a function is doing only one thing—they can still be broken down into steps. Throughout the previous section we described the how links work in SpIDER and demonstrated how a fragment of code can be moved to another Frame. This is one form of a programming language independent abstraction that SpIDER supports.

Figure 1.5 also shows another form of programming language independent abstraction that SpIDER supports. A print statement has been placed in a box with two arrows from separate parts of the code pointing to it. We call this Anonymous Indirection. It utilises out of flow functionality provided by the flow walker.

Multimedia Elements. As a Spatial Hypermedia system, SpIDER allows for the embedding of a variety of non-textual information. This allows programmers to integrate more of their work into one platform. Rather than having to switch between planning documents and

code, programmers can instead include their planning documents side-by-side with their code. Figure 1.7 shows an example of multimedia being used to enrich the implementation of a graphical user interface (GUI) for a simple up/down counter application. The code is boxed and positioned around the image to communicate how the sketch relates to the code. Details concerning how spatially laid out code such as this is serialised is provided in Section 7.1.

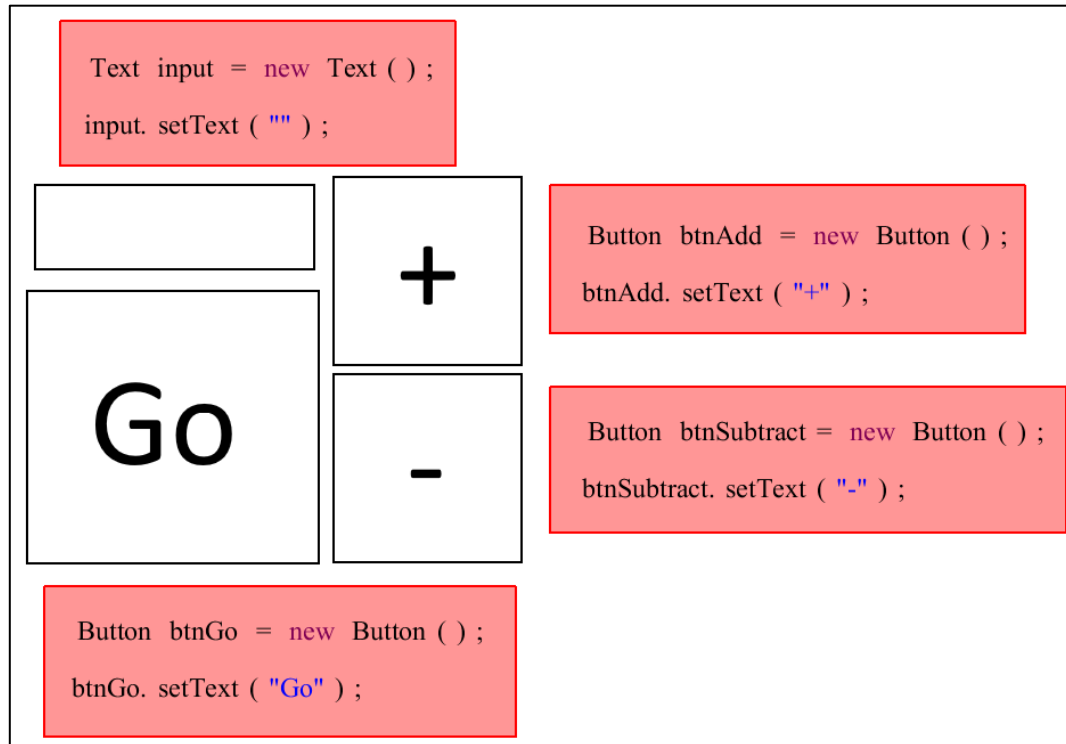


Figure 1.7: Integrating sketch with code.

The sketch itself links to the page seen in Figure 1.8. The programmer has added the controls to the GUI but has yet to customise them: a screenshot of the running program has been taken and positioned in a similar fashion as to the sketch from Figure 1.7. By keeping the screenshot up-to-date the programmer can see what is left to be done. Moreover, in SpIDER, it is easy to keep the old screenshots archived on a separate Frame, allowing the programmer to access a history of development if they so desired.

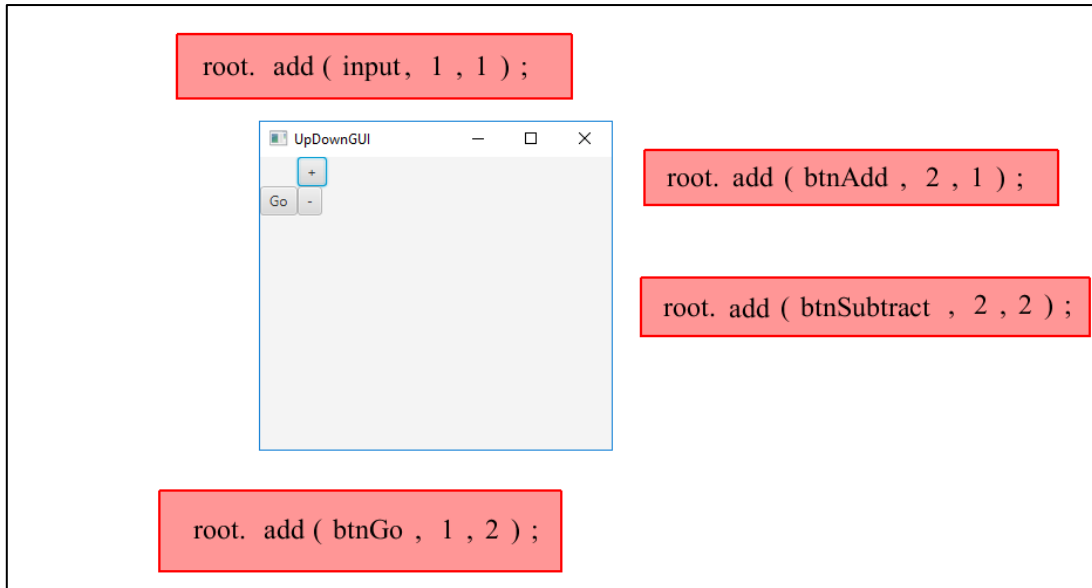


Figure 1.8: Integrating screenshot with code.

1.4 Thesis Outline

This thesis has been organised as follows. Chapter 2 begins by describing Spatial Memory and how programmers already utilise it through space, shape and relative position of code. It explains the motivations for creating a Spatial Hypermedia-based IDE. We then move to an analysis of existing IDEs such as Eclipse in Chapter 3, and establish what functionality is core. We also examine elements of spatial functionality present in these IDEs.

General purpose Spatial Hypermedia environments are reviewed in Chapter 4. In support of this examination, Chapter 4 introduces a formal descriptive model of the authoring process. In Chapter 5 we turn our attention to existing IDEs with significant use of spatial features, again using the formal model to structure the analysis. Building on these examples, the chapter continues with a broader discussion showing that there are more ways in which Spatial Hypermedia capabilities can be applied to the needs of programmers than can be seen in existing solutions. This leads to the decision to extend an existing general purpose Spatial Hypermedia system and integrate IDE functionality into it by using the Eclipse Java Development Tools. The resulting system was called SpIDER.

Expeditee, the Spatial Hypermedia system that SpIDER is built on, is further documented in Chapter 6. Its unusual user interface, with a lack of widgets and heavy use of the mouse is discussed. Chapter 7 documents the design and implementation of SpIDER. The flow

walker is evaluated in Chapter 8 through an initial and follow up study. The results show that: with restrictive examples, on their first impression, people are able to understand code laid out in SpIDER's spatial style. These results influence Chapter 9 where the idea of 'Spatial Development Patterns' are introduced and discussed. A series of examples is used to show the benefits of SpIDER whilst discussing the motivations behind each example. While not exhaustive, these examples demonstrate what has been gained through a combination of spatial layout and uniform treatment of elements. Chapter 10 explores the potential pitfalls associated with the spatial freedom provided by SpIDER. Chapter 11 concludes the thesis by summarising what has been gained from the research and a brief discussion of future work.

Two hypotheses encapsulate the goals of this thesis:

- A process can be established that allows for spatially arranged code to be unambiguously understood by programmers and compilers. This process should allow code layout practice to range from serial (as seen in conventional IDEs such as Visual Studio) to diagrammatic.
- A Spatial Hypermedia-based IDE can be used to integrate many stages of the software development process. The uniform treatment of elements within a Spatial Hypermedia system will allow programmers to intertwine forms of documentation that are traditionally kept separate from the code (test results, variable dumps etc.) and allow programmers to better customise their environment for each project.

The key contributions of this research are:

- A review of literature connecting Spatial Memory, Spatial Hypermedia and programming in Integrated Development Environments.
- A review of the Spatial Hypermedia system Expeditee.
- The design and implementation of the Spatial Hypermedia-based IDE SpIDER.
- The analysis and evaluation of the algorithm designed to allow for the spatial layout of code in SpIDER.
- An exploration of how a Spatial Hypermedia system can be used for programming, the benefits and risks involved, especially concerning the quality of produced code.

Chapter 2

Spatial Memory for Software Development

Spatial Memory is a field in cognitive psychology concerned with the way in which people navigate environments and remember the locations of objects [1]. People utilize their Spatial Memory on a daily basis to navigate around environments and locate objects they need. Making your way to your desk when you arrive at work and fetching a particular size of paper from the office stationery cupboard to refill the printer are both examples where Spatial Memory is used. There are parallels in these activities with those that software developers undertake when writing software, especially when they are required to navigate through extensive bodies of source code (text) to access information they have visited before.

A large fraction of time programming is spent on complex comprehension tasks such as debugging and testing [2]. Spatial Memory can be leveraged to reduce cognitive load [3, 4], reducing the time spent attempting to understand previously authored code and consequently increasing the time they have available to focus more on the complex comprehension tasks necessary to complete their work.

In this chapter, we define what Spatial Memory is and how it can be leveraged to help with programming, particularly when complex comprehension tasks are performed. This discussion is a precursor to presenting the design and implementation of SpIDER in Chapter 7.

This chapter also covers:

- i. The terminology used for describing graphical interfaces that utilise Spatial Memory.
- ii. Examples of the way in which programmers currently use Spatial Memory.
- iii. Issues for designing an IDE that amplifies the use of Spatial Memory, using observations made about (i) and (ii).

Section 2.1 elaborates on Spatial Memory by discussing the short and long-term components and how it can be utilized for navigation and object location. The terminology we will be using to describe interfaces that utilize Spatial Memory (spatial interfaces) will be covered in Sections 2.2 and 2.3 and is adapted from Scarr et al. [5]. These sections review Spatial Memory literature (with a focus on object location over navigation) and produce a set of guidelines designed to help interface designers create interfaces that can utilize Spatial Memory. Section 2.4 gives examples and an analysis of how programmers currently make use of their Spatial Memory. Section 2.5 extends the work done by Scarr et al. by applying the concepts discussed in previous sections to authored content—with a focus on program code.

2.1 Spatial Memory

Cognitive psychology uses a model of Spatial Memory consisting of two components: short and long-term memory. Below we discuss the functioning of these components and give examples of how they are utilized by people in day-to-day life. Also discussed are the differences between utilizing Spatial Memory for object location and navigation.

2.1.1 Short-Term Memory

Short-term Spatial Memory is a space limited system that is useful for performing complex cognitive tasks [6]. People are able to keep pertinent information in mind while working towards a specific goal, such as remembering which streets they walked down on the way to the park so that they can retrace their steps to return home. Baddeley and Hitch’s multi-

component model [7] is a popular theory explaining how three parts of working memory are able to work together to achieve these results. Baddely and Hitch's model has been subject to refinement since its initial publication [8]; Figure 2.1 shows recent thinking on how these three parts interact.

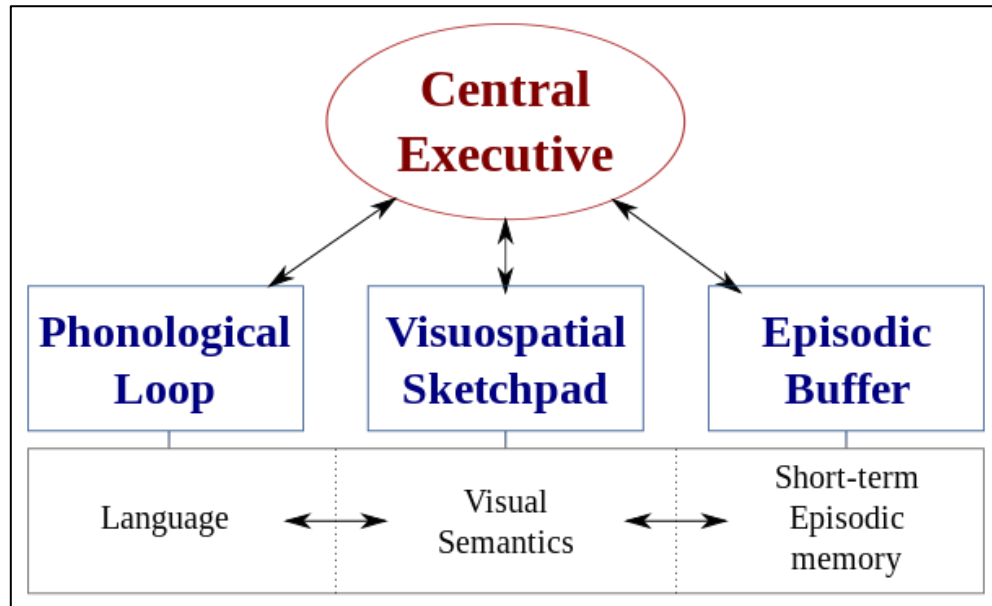


Figure 2.1: The functioning of three separate but interacting systems in working memory.¹

The diagram shows three separate systems that interact indirectly [8]:

1. The Phonological Loop which processes and stores information relating to auditory and linguistic information.
2. The Visuospatial Sketchpad which processes and stores visual and spatial information.
3. The Episodic Buffer that orders events.

¹ <https://commons.wikimedia.org/w/index.php?curid=12843390>. Last Accessed on during October 2017

These three systems are directed by the Central Executive to achieve a specific task.

Extending our example from earlier: when travelling from home to the shops to buy milk a person utilises the three systems of their short-term memory by:

- Remembering the names of the streets they walk down, a language task handled by the Phonological Loop.
- Recalling the order of the streets they walk down, utilising the Episodic Buffer.
- Forming a mental map of their trip in their mind, utilising the Visuospatial Sketchpad.

Spatial Memory is primarily concerned with the Visuospatial Sketchpad, which processes both spatial and visual information.

2.1.2 Long-Term Memory

Long-Term Spatial Memory utilizes a hierarchical structure [9]. In our example, when a person forms a mental map of their trip to the shops in their mind, this map is represented hierarchically once moved into long-term Spatial Memory. When repeating the trip at a later time the person starts at their 'home' node and decides on which street to walk down, eventually arriving at the next node and making another decision; at each node the person is able to recall the next step to take.

2.1.3 Object Location and Navigation

Spatial Memory can be used for both navigation and object location [5]. These two uses of Spatial Memory differ significantly according to the viewpoint of the individual when attempting recall. During object location, a person has an overview of the environment they are engaged with, for example: finding an item in a filing cabinet or a particular icon on their desktop computer. Conversely when using Spatial Memory to navigate a person is inside the environment and is able to build a map in their mind to get to their destination, for example: taking the quickest route to the office with the filing cabinet. The map created in the individual's mind provides an overview of the route. In contrast to the overview present when performing object location, this overview is imagined.

A popular test for measuring a candidate's ability to leverage their Spatial Memory is the Corsi Block-Tapping Task [13]. In this test participants are exposed to a set of spatially and haphazardly arranged blocks. The scientist running the experiment taps on several of the

blocks and has the participant reproduce the order. This continues in rounds with the number of blocks being ‘activated’ increasing on each round. Other studies, both for navigation [10, 11] and object location [12, 4, 13], have shown the importance of Landmarks, which are discussed in more detail in Section 2.2 below.

2.2 Single View and Viewport Interfaces

Scarr et al. discuss how choices made in designing a program interface can affect a user’s ability to utilize their Spatial Memory [5]. Here we begin to summarise their work with particular emphasis on their concepts of ‘Single View’ and ‘Viewport’ interfaces.

Interfaces—or large interface components—that show all their widgets at once are classified as a ‘Single View’ interface. Figure 2.2 shows a screenshot of the Windows 10 Scientific Calculator, which is an example of a ‘Single View’ interface. Conversely, interfaces that conceal some fraction of their widgets at any given time and provide the user with some level of control over what is visible—for example, through scrollable panels—are considered ‘Viewport’ interfaces.

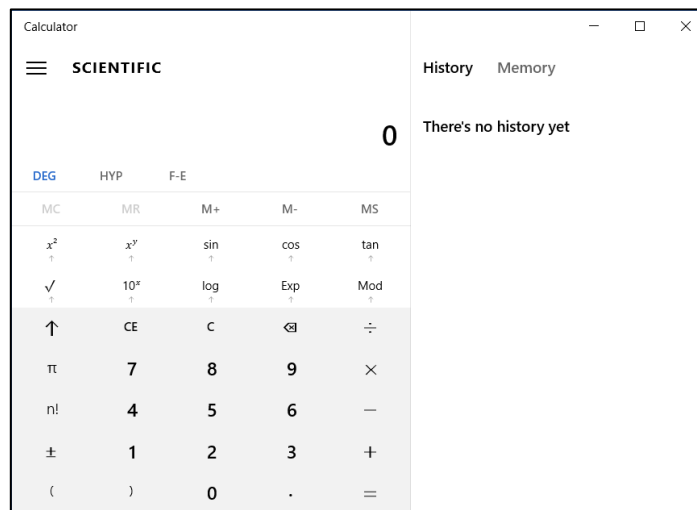


Figure 2.2: The Windows 10 Scientific Calculator: an example of a ‘Single View’ interface.

2.2.1 Single View Interfaces

When designing interfaces to utilize a user's Spatial Memory for object location, a 'Single View' interface tends to produce better results than a 'Viewport' interface. This is due to two characteristics of Spatial Memory:

- Spatial Memory is built through the repeated use of the spatial interface [14].
Having a single view means each use of the interface does not split progress developing Spatial Memory between multiple views. Furthermore, Doeller and Burgess discuss the benefits of the strict boundaries present in 'Single View' interfaces and how they help build Spatial Memory [4].
- When a user is provided with an interface they are familiar with, the efficiency of using Spatial Memory to locate, navigate and activate controls causes actions that change the visibility or position of controls to be a comparative bottleneck [15].

Scarr et al. also discuss their experiments with widgets they call CommandMaps [15]. Figure 2.3 shows a CommandMap widget being used to replace the existing Ribbon system in Microsoft Word. They find that experienced users are able to use the CommandMap system to execute commands faster than the Ribbon system. Inexperienced users perform about the same regardless of which system is being used.

Placing all controls onto one panel, rather than requiring users to switch between which set of controls is active, has allowed for better utilisation of Spatial Memory by those with experience. As inexperienced users perform the same, regardless of their use of CommandMap or traditional layout, this suggests that there is no downside to using a CommandMap.

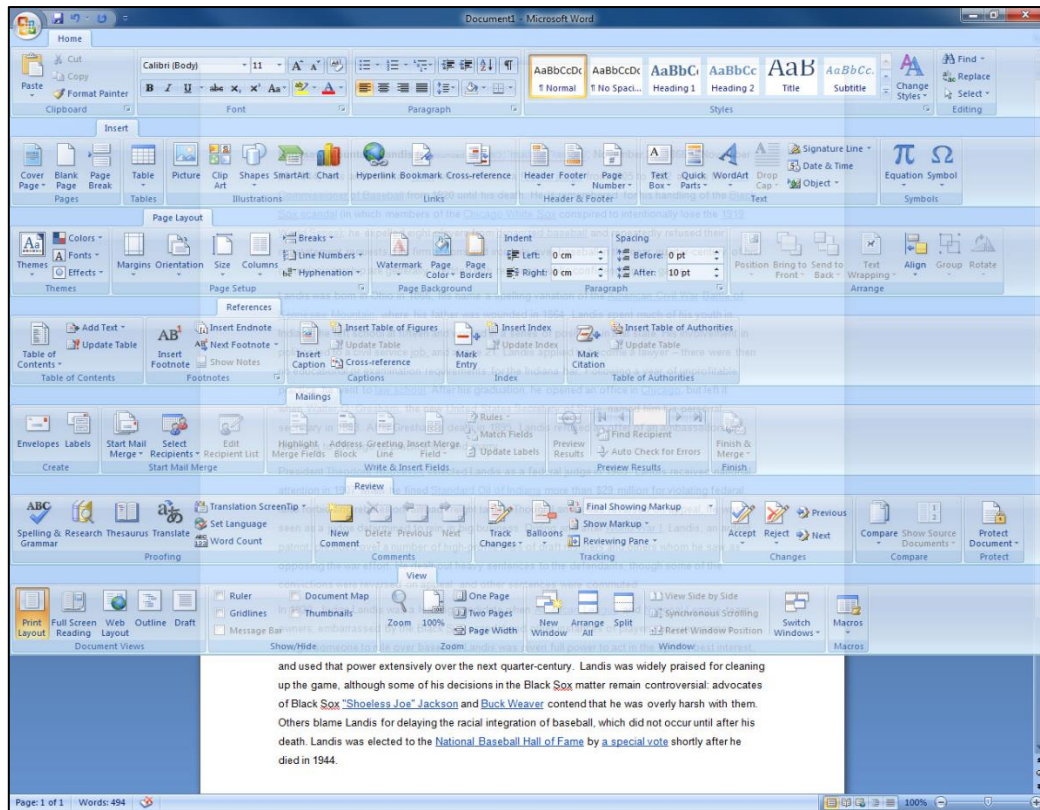


Figure 2.3: An example of a CommandMap widget being used in place of the Microsoft Word Ribbon [15].

2.2.2 Viewport Spatial Interfaces

Due to restrictions such as screen size or design goals, it is often implausible to present all the controls to the user at once. Therefore ‘Viewport’ interfaces work by providing the user with a limited view. A common method to achieve viewport behaviour is to provide the user with a pan and zoom interface.

Pan and zoom interfaces give the user fine-grain control over what region of the information space is visible. They are able to centre details that they are currently engaged with and choose a level of zoom that provides them with the information they want to see while minimizing unneeded information. Google Maps is a prime example of an interface that needs to use a viewport spatial interface, driven by the fact that it is providing access to an information space (the map) that is vast in size.

Spatial Memory is not as easily leveraged in a ‘Viewport’, compared to a ‘Single View’ interface, because the boundaries of the information are no longer anchored to the edges

of the application window [5, 4]. There are two methods that can be used to mitigate this issue: overviews and landmarks.

2.2.3 Overviews

Overviews are low-detailed, scaled down representations of the entire information space. They feature heavily in strategy computer games such as the Sid Meier's Civilisation series. An in-game screenshot of Sid Meier's Civilisation V (Civ5) can be seen in Figure 2.4, where the screenshot has been modified to include dotted line circles for the purpose of discussion.

Inside the red dotted circle is the mini-map. The mini-map contains:

- A trapezoid-shaped box showing where the current viewport is currently aimed at.
- Fog-of-war representation: a concept from strategy games of map area that the player has yet to explore.
- Colour coded areas representing territory controlled by other in-game factions.

The information communicated by the mini-map can help the player build and retain a spatial map in their mind by anchoring the player's position to an absolute position on the map. Broadly speaking an overview provides a undetailed and miniature stand-in 'Single View' spatial interface inside the larger and more detailed 'Viewport' spatial interface that the user can use to obtain an absolute position of the active region they are examining through the viewport.



Figure 2.4: An example of a Viewport spatial interface featuring the use of an overview and several landmarks.

2.2.4 Landmarks

Another potential way for users to build and retain Spatial Memory in a ‘Viewport’ spatial interface is through the use of landmarks [11]. Landmarks are noticeable and significant aspects of the interface that can be used as anchor points. A spatial map can be built with the landmark as a central point and with other items in the interface being remembered in relation to that landmark.

Unfortunately, like overviews, and as Doeller and Burgess show, landmarks are not a complete stand-in for a ‘Single View’ interface as building spatial relationships to landmarks is not as automatic as building those relationships to the edge of an interface [4]. Doeller and Burgess examine the automatic nature of Spatial Memory. They find evidence suggesting that people build a spatial map in their mind automatically when dealing with ‘Single View’ spatial interfaces but require a more conscious effort when utilising landmarks in ‘Viewport’ spatial interfaces.

When using Spatial Memory to navigate in the real world, landmarks might be notable shop signs or large roundabouts. Consider, for example, how many times you have received directions to a location where one of the instructions was along the lines of: “keep going until you see the big green sign”. In software, landmarks might be particularly important

icons. Using Figure 2.3 as an example, the “Spelling & Grammar” icon could be considered a landmark due to its size, prominent positioning and possibly frequent use.

Figure 2.4 uses blue dotted line circles to give examples of landmarks present in Civ5. Four of the five highlighted areas are towns—a major aspect of the game—three of which are from one faction and the fourth from another faction. Furthermore, these landmarks can be seen as coloured dots on the mini-map, connecting the overview with common landmarks. The fifth landmark is not a city and is not represented on the mini-map, it is a mountain range. Mountain ranges are reasonably rare impassable terrain which makes them good candidates from which to anchor other aspects of the game off.

2.3 Spatial Stability

Scarr et al. also discuss the concept of spatial stability [5]. An interface whose content tends to remain in a fixed position is referred to as a spatially stable interface. A spatially stable interface lends itself to utilising Spatial Memory for object location more than a malleable interface, allowing users to navigate directly to remembered location of the content they want rather than having to search for it. Figures 2.2 and 2.3 are both examples of spatially stable interfaces.

- The calculator in Figure 2.2 is spatially stable but is also very simple.
- The CommandMap example from Figure 2.3 is also highly spatially stable but significantly more complicated. Complex components, such as the font selection feature, utilise a scrollbar and are therefore not spatially stable. A spatially stable alternative to font selection is discussed in Section 2.3.1.

2.3.1 Scrolling

Scrollable widgets are commonly used and unfortunately cause problems with utilising Spatial Memory. Three studies are detailed below that compare scrollable content with spatially stable content, finding that participants both prefer spatially stable content and that, when interacting with spatially stable content, they complete complex comprehension tasks to a higher standard. It should be noted that while scrollable widgets destabilise the position of content, and therefore interfere with the development of Spatial Memory, they are still a useful component for accessing an unbounded quantity of information; especially if the information lends itself to a specific ordering, such as alphabetic.

O'Hara and Sellen discuss an experiment comparing scrollable on-line documents with paper versions of those documents [16]. They report that participants moved through pages in the paper version with speed and accuracy, and were able to use "the fixity of information with respect to the physical page" to find what they were looking for. In sharp contrast, on-line document navigation was found to be slower and participants found themselves splitting the cognitive load between the task they were attempting to achieve and navigating around the document.

Cockburn et al. developed and reported on an interface called Space-Filling Thumbnails (SFT) [17] that was designed for navigation using Spatial Memory object location. Instead of being able to scroll through pages a user is able to click the middle mouse button to be taken to a navigation page as seen in Figure 2.5. Clicking on a thumbnail in this view navigates the user to that page. Results from comparing SFT to other types of interfaces showed that SFT performed well, especially when compared to scrolling interfaces.



Figure 2.5: Space-Filling Thumbnails [17].

Gutwin and Cockburn developed another interface called ListMaps [18], designed to utilise Spatial Memory for object location. ListMaps take content normally presented in a scrollable list (such as fonts) and displays that content in a 2D array-like structure, a screenshot of which can be seen in Figure 2.6. Through experimentation, they found that task times for inexperienced participants increased but task times for practised participants were significantly faster.



Figure 2.6: ListMap for font selection [5].

2.4 Programmers Navigating with Spatial Memory

With Sections 2.4 and 2.5 we now turn our attention to looking at the task of applying Spatial Memory to programming. This goes beyond the work done by Scarr et al. [5, 17, 18, 15] who focused on traditional computer interfaces that contained non-editable components such as buttons and labels. Prior to discussing our solution for applying their work to program code we will first analyse three examples from popular IDEs and discuss how programmers can utilise their Spatial Memory with the example interface.

2.4.1 Eclipse Package Explorer

The Package Explorer in Eclipse (as seen in Figure 2.7) is an interface designed to allow a programmer to navigate around their project. As explained in Section 2.1, long-term Spatial Memory is hierarchical in nature, and so too is the structure of a software project. This

match should help a programmer perform faster object location by using packages as landmarks. While a small project with all of the components expanded could be represented with a 'Single View' interface, more reasonably sized projects are likely to require scrollable interfaces--and so the representation will interfere with spatial memory. Furthermore, spatial stability is hurt further by the collapsible components of this interface, for example: expanding the *org.apollo.gui* package (Figure 2.7) will place the revealed classes at different positions, based on what is expanded above it.

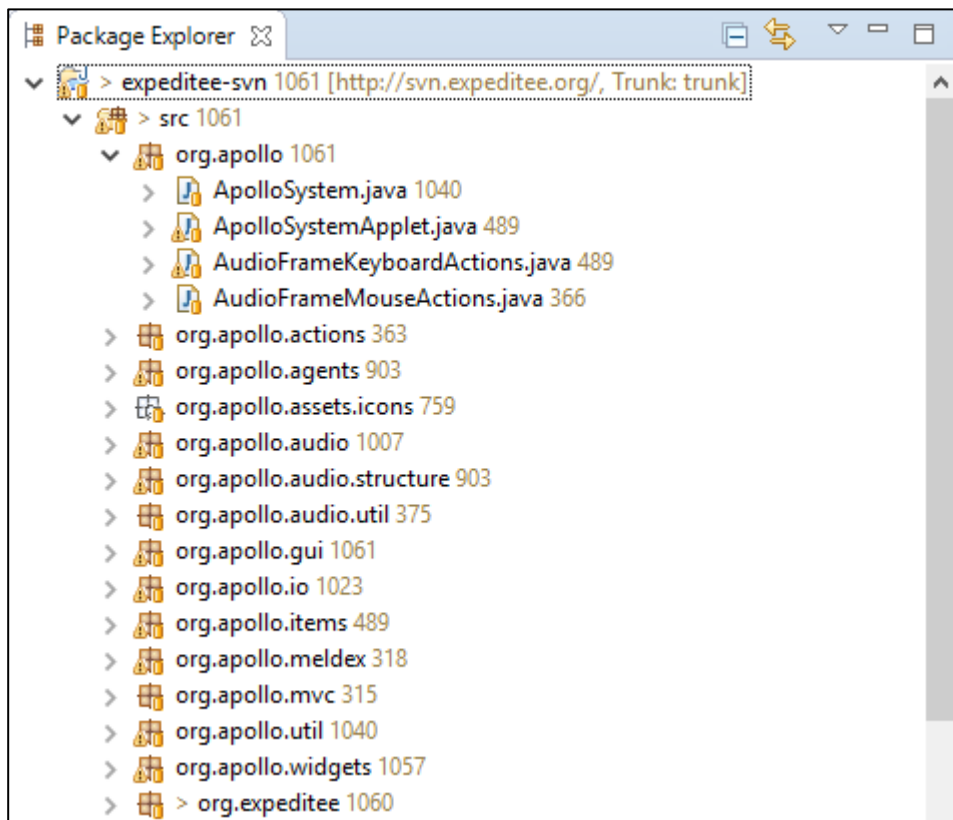


Figure 2.7: Hierarchical structure of program as shown in Eclipse.

2.4.2 Eclipse Outline

The Eclipse Outline is another interface that has trouble utilising Spatial Memory. It summarises the content of the selected Java source file, an example of which can be seen in Figure 2.8. Like the Package Explorer, the Outline also features collapsible components with the potential for scrolling; as a result of this, they share the same problems utilising Spatial Memory. However, it also adds additional functionality--sorting and filtering. The programmer is able to move the position of the content around by sorting it alphabetically or filtering certain items out, for example: hiding private fields will move the constructor

ExpReader(String) further up the list. It should be noted that while these features of the interface hurt spatial stability (and therefore Spatial Memory), they do not make it a bad interface. Instead, it shows that it is more useful for searching or browsing than it is for utilising Spatial Memory for object location.

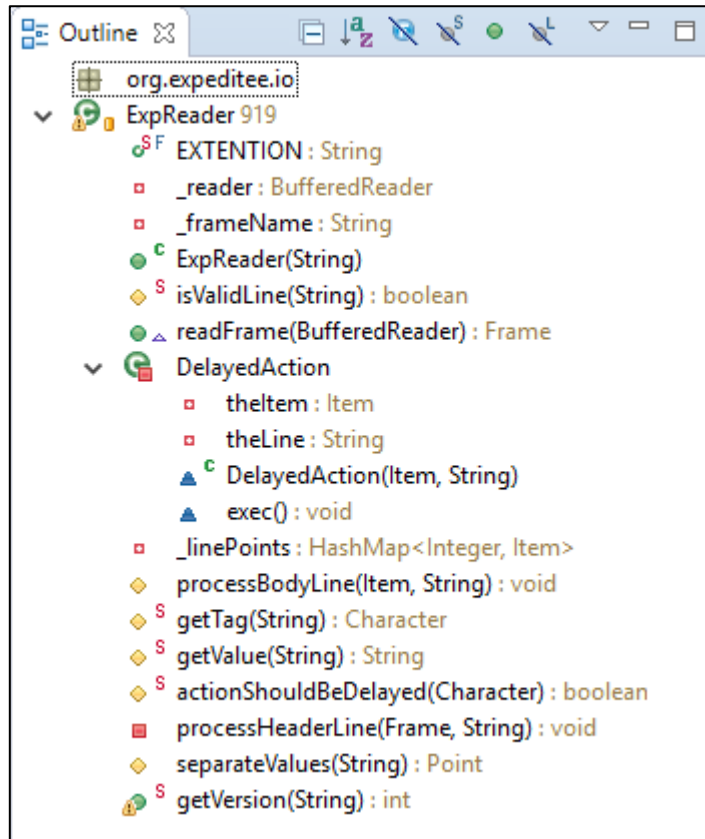


Figure 2.8: The Eclipse outline interface: useful for searching but not spatial object location.

2.4.3 Program Code

While not specifically an interface, code lends itself to being utilised by Spatial Memory. In this way, it is more akin to the Google Maps interface that provides access to an information space. The hierarchical structure of classes (classes containing functions, functions containing statements, etc.), heavy use of indentation and other white space all contribute to forming Spatial Memory. DeLine et al. augment Visual Studio [12] to leverage the unique shape of code for Spatial Memory. Two separate augmentations are developed and tested:

1. The Code Thumbnail Scrollbar (CTS)—as seen in Figure 2.9—is an interactive augmented scrollbar that acts as an overview to the complete editor. As discussed in Section 2.2 the editor can be considered a ‘Viewport’ spatial interface (when there is enough content to require scrolling) because the scrollbar allows the user access to currently unseen content. Also, as discussed, adding an overview to a ‘Viewport’ spatial interface should help the user leverage their Spatial Memory.
2. The Code Thumbnail Desktop (CTD)—as seen in Figure 2.10—is a newly designed ‘Single View’ spatial interface similar to Space-Filling Thumbnails [17] designed by Cockburn et al.

Their evaluation used 11 participants, with an average age of 34 and an average of 15 years programming experience. Once participants had been given time to familiarise themselves with the provided code they were asked to make a series of alterations. Following this, they were asked to perform a series of targeted search tasks, such as being asked to navigate to a particular function. The version of Visual Studio the participants used not only included Code Thumbnails but also recorded their actions.

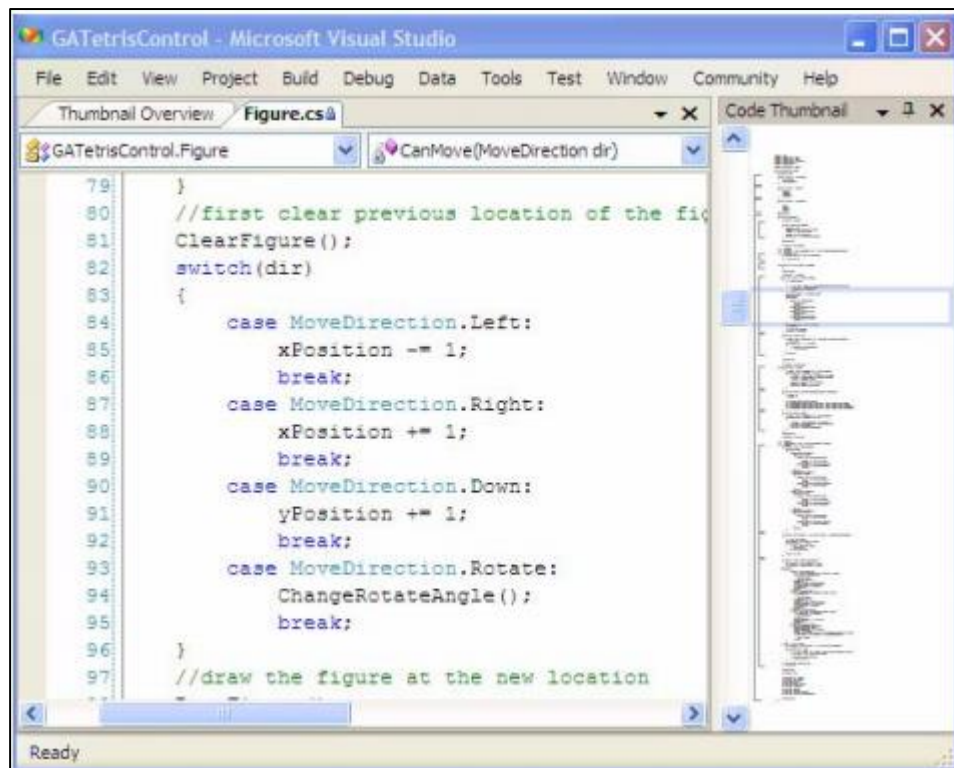


Figure 2.9: Code Thumbnail Scrollbar [12].

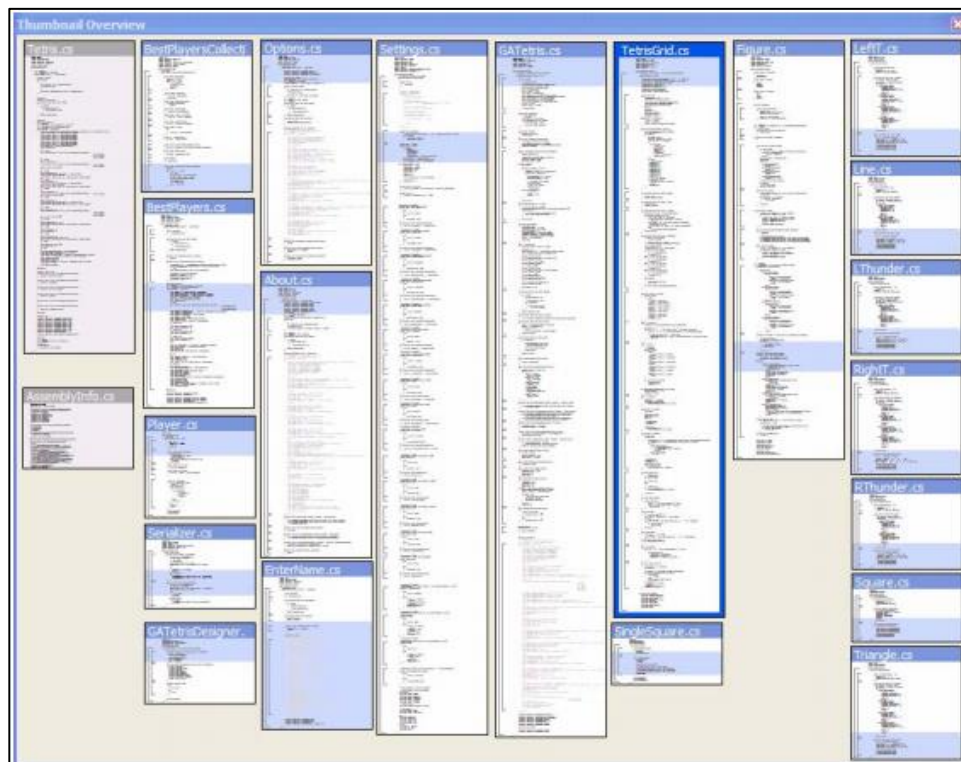


Figure 2.10: Code Thumbnails Desktop [12].

During the series of programming tasks, DeLine et al. found that participants opted to use some aspect of Code Thumbnails for between 40% and 91% of their navigations [12]. CTD was used (on average) slightly more frequently than CTS. Other common forms of navigation were: using Visual Studio's build-in search, 'Solution Explorer' (equivalent to Eclipse's 'Package Explorer') and 'Go to Definition' functionality.

When performing targeted search tasks rather than alterations to the code, the programmer's behaviour changed. The CTD was used more frequently, being used on average in 64% of searches. However, the CTS was only used during 11% of searches, falling behind text search at 16%. This suggests that having had practice using both new systems, participants favoured the 'Fixed View' spatial interface option.

2.4.4 Software Visualisation

Software Visualisations, such as Codemap which Kuhn et al present [19], use Spatial Hypermedia elements such as size and position to communicate information about the code base of a software project. An example of a Codemap can be seen in Figure 2.11. In this example, directed arrows are used to communicate the flow of potential execution from an origin: a function within the *MenuAction* class. The proximity and size of heat map bubbles in relation to labels communicates the extent of test coverage on that area of code. Therefore, in this particular example, there might be a reproducible bug present in the application that can be triggered by interacting with the menu item associated with the *MenuAction* class—this visualisation can assist in identifying likely locations where the offending code is.

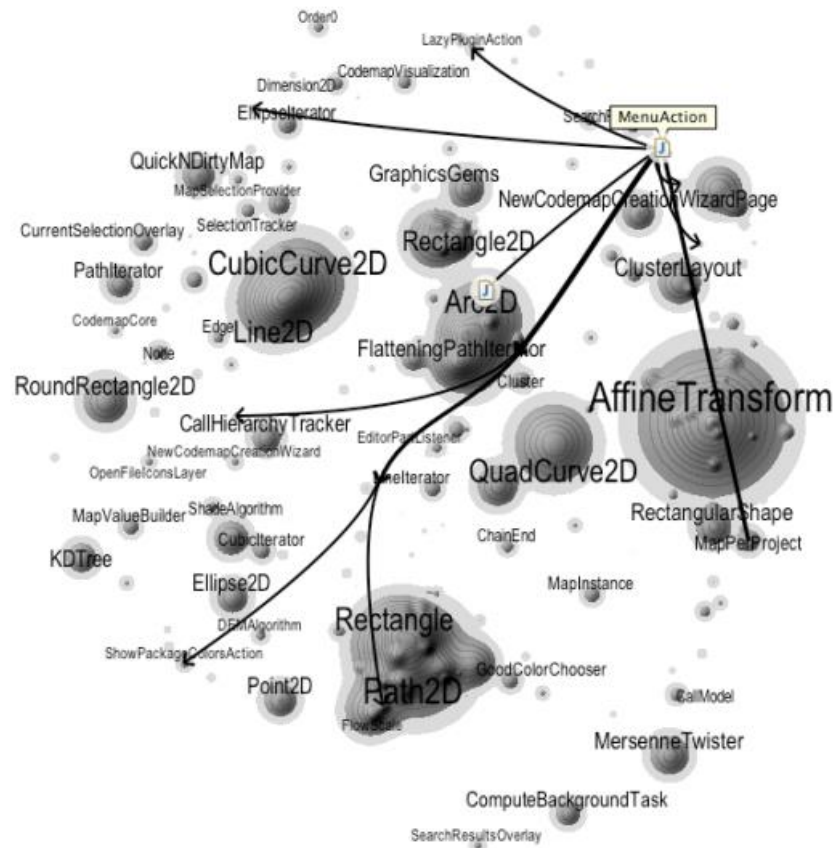


Figure 2.11: Example output of the Codemap Software Visualisation Tool.

Evaluation of the Codemap software visualisation found evidence that:

- Programmers reason about a code base in spatial terms—thinking about some programming elements as being higher or lower than others—and that ideally programmers should be given the freedom to spatially arrange visualisations themselves rather than be limited to a procedurally generated visualisation.
- Spatial layout can promote the usefulness of functionality, such as search results, that are frequently displayed in a mundane fashion.
- Programmers strongly relate program package structure with code connectivity and act surprised when a visualisation spatially positions functionality based on some other ruleset; even when there is good reason for this to be the case.

Another stated use for the Codemap software visualisation is the ability to colour code a heat map representation of the code to signify which source files individual programmers on a team currently have open.

2.5 Utilising Spatial Memory for Programming

This chapter has listed and discussed the considerations that need to be made when designing an interface to utilise Spatial Memory. However, the literature discussed deals with non-authored content—widgets such as buttons and labels. The experiments carried out by DeLine et al. show that experienced programmers are both willing and able to use Spatial Memory to navigate around C# projects [12]. However, this thesis is concerned with creating an IDE that utilises Spatial Memory throughout the entire development process; not only for navigation but also for planning, authoring (editing code) and debugging.

The question remains: What would an IDE designed to utilise Spatial Memory from the ground up look like?

2.5.1 Applying Single View and Viewport Interface Concepts to Authored Content

Throughout Section 2.2 we established that:

- ‘Single View’—compared to ‘Viewport’—spatial interfaces produce better results when attempting to utilise Spatial Memory.
- Overviews and landmarks in ‘Viewport’ spatial interfaces help close this gap.
- Landmarks are generally helpful, regardless of which type of spatial interface is present.

While it might be better—for Spatial Memory—to use a ‘Single View’ spatial interface, it is not always possible or desirable due to the amount of content. This problem is especially prominent in an authoring application where the quantity of content is not fixed—and realistically unbounded. Consider an application such as Microsoft Word, where the quantity of information that needs to be displayed will be different based on what is written. At one extreme of the spectrum, a short poem may be written, such as a limerick, at the other, a Computer Science PhD thesis could be written! For this reason, it is less useful to talk about ‘Fixed View’ and ‘Viewport’ spatial interfaces when talking about authoring applications. However, it is still beneficial to consider concepts such as landmarks, overviews and utilisation of the edges of the screen when discussing authoring applications.

As an extension to the work done by Scarr et al. [5, 15] we use the terms Fixed Size Spatial Interface and Variable Size Spatial Interface to classify authoring applications based on how

they occlude content once there is too much to display without growing the size of the application window. Strictly speaking, both classifications are a form of ‘Viewport’ interface, though Fixed Sized Spatial Interfaces share many of the properties that ‘Single View’ interfaces do.

A Fixed Size Spatial Interface aims to allow the edges of the screen to be used for growing a user’s Spatial Memory. In order to achieve this, methods for occluding information by stretching or distorting the size of the information space—commonly scrolling and zooming—cannot be used, as these techniques cause the position of items relative to the edge of the screen to change. Instead, functionality that provides a surrogate is used. For example, on a webpage, a hyperlink with the text ‘Employees’ that directs a user to another webpage listing a company’s employees is a surrogate for the content on that page. Another commonly seen example is the ability to collapse a named section of a Microsoft Word document.

Conversely, a Variable Sized Spatial Interface does not limit the types of functionality that can be used to occlude information. Typically, a Variable Sized Spatial Interface makes use of scrolling (or panning). Such functionality is able to dynamically adjust the size of the authorable space.

It should be noted that these definitions rely on a fair appraisal of how the application was intended to be used. To clarify, avoiding the use of scrollbars in authoring applications such as Microsoft Word is possible by limiting the amount of content or making heavy use of external-facing links. However, this is not the typical use of this software and as such we consider Microsoft Word to be a Variable Sized Spatial Interface.

2.5.2 The Shape of Code and Other Media – Landmarks

Compare the shape of code with the shape of content in Microsoft Word. An argument can be made that code lends itself to utilising Spatial Memory more due to the heavy use of indentation—creating more unique shapes and therefore giving more opportunities for landmarks. Conversely, the opposite argument can be made as code tends not to have other forms of media such as figures—lessening the opportunity for landmarks.

2.5.3 Spatial Stability of Authored Content

As program code is authored and frequently rewritten, spatial stability is a unique challenge. Traditional flat file text editors, as seen in currently popular IDEs, can have chain reactions occur throughout significant portions of the file simply by adding a character to a string. For example, a traditional flat file editor may decide to move an entire word to the following line as it grows in length, potentially having run-on effects throughout the file. Furthermore, traditional text editors use scrolling, which—beyond the discussion in Section 2.3—has the effect of ordering functions in a similar fashion to how paragraphs are ordered when writing a prose. When writing prose, the order paragraphs appear in contributes to the meaning of the entire text. However, the ordering of functions in a file is comparatively pointless.

Chapter 3

Traditional Programming

Anecdotally, programmers often talk about the time they spend authoring code. Regardless of experience—whether they have been programming for years or decades—they note how it always seems to take them longer to ‘code up something’ than they expect. It is also often said that a programmer spends more time thinking about their existing code than writing new code. Noticing the lack of empirical evidence to support such claims, Minelli et al. conducted an investigation of how programmers spend their time [20]. They achieved this by collecting fine-grained usage data with a custom built interaction profiler. The results of their study showed that very little time spent programming is actually spent editing code. Most notably, 70% of a programmer’s time is spent performing code understanding, higher than the 50% figure they cite as being anecdotally reported in the literature. Their study also found that 14% of a programmer’s time is spent performing ‘non-significant’ user interface interactions. These are classified as actions, such as moving or resizing windows, that do not directly contribute to the completion of the programmer’s task. The remaining portions of a programmer’s time are spent (in order of magnitude) outside the IDE, editing code and navigating. The IDE Minelli et al. used in their experiment makes it difficult to generalise their results due to differences in user interface design as compared with popular IDEs such as Eclipse [21].

These first two statistics, the time spent performing code understanding and performing off-task user interface interactions, are the biggest concerns. In Chapter 2 we discussed Spatial Memory, the benefits it provides and how to best leverage it. With up to 70% of a programmer’s time being ‘consumed’ by code understanding, we seek to reduce this figure through the introduction of Spatial Hypermedia to promote the use of Spatial Memory. The introduction of Spatial Hypermedia to programming may also help reduce time spent on

off-task user interface interactions and minimise the need to task switch to other applications—a cognitively expensive process [22]—through its inclusion of multimedia.

Throughout this thesis, we need to refer to the type of IDE that is prevalent today, such as Eclipse and Visual Studio. It is tempting to refer to today’s programming environments as modern IDEs. However, for our purpose, this term is ambiguous. To distinguish research-led experimental IDEs, such as those we review in Chapter 4, from main-stream commercial IDEs, we refer to the latter as traditional IDEs.

In this chapter, we examine the development of programming environments in order to identify some underlying issues that contribute to longer than necessary software development cycles, such as those identified above by Minelli et al. Identifying these issues will help direct the focus of this thesis. In Section 3.1 we document traditional IDEs, both the history behind them and the resulting form they take today. We take note of their core components, using these notes to help establish the scope of development for our Spatial IDE (SpIDER). Functionality that is considered core to the design of a traditional IDE must have an analogue in SpIDER. Section 3.2 looks at the concept of abstractions in programming. We document both prescribed and artificial abstractions and discuss the support they receive in traditional IDEs. Using our analysis from Section 3.2 we then address rigidity in IDEs in Section 3.3, identifying where there is room for improvement.

3.1 Traditional Integrated Development Environments

Today, the base feature set of an IDE is well established. However, this has not always been the case; functionality such as syntax highlighting was once considered a novel idea [23]. Section 3.1.1 uses an historical lens to examine the emergence of IDEs as tools designed to assist with the task of programming. Notable milestones, driven by academic and economic forces are documented. Having arrived at the modern-day IDE by the end of Section 3.1.1, Section 3.1.2 then discusses the collection of functionality that we consider ‘core’ to the IDE experience. This allows us to establish the set of functionality that our spatial IDE will support.

3.1.1 The Emergence of Integrated Development Environments

Historically. In the 1950s high-level programming languages such as FORTRAN, developed at IBM, first appeared as alternatives to assembly language. They were characterized by a

batch driven process of compile, link and go. Each of these stages was segregated from the other. The 1960s and 1970s saw further development in programming languages. Two notable languages from this period were Dartmouth BASIC and FORTRAN 77.

Thanks to several advances, including the ability to store entire programs in memory, these next-generation languages were able to more tightly couple the execution of code with a command line environment—the beginnings of integrated development environments (IDEs). Dartmouth BASIC allowed programmers to issue commands via a command line interface. For example, 'LIST' could be used to display the currently loaded program to the screen and 'OLD' allowed a previously saved program to be moved from long-term storage to memory. Programming in FORTRAN 77 took a significant step forward with the development of the utility program WATFOR at the University of Waterloo. WATFOR allowed students at the university to submit their code for execution without having to concern themselves with the compile, link and go process, freeing them to focus on producing better quality code. With the compile, link and go process being handled in a single pass process, should a student's code contain errors, they were able to get immediate feedback. WATFOR was to be succeeded by multiple versions of WATFIV.

The next significant milestone in the historical development of IDEs came in the form of Maestro I and Turbo Pascal. Many consider Maestro I—released in the mid to late 1970s—to be the first purpose-built commercial IDE. Differing from previous systems that used card readers and punch cards to load a programmer's source code into memory, the Maestro I included a purpose-built keyboard that allowed programmers to directly type their code into the system. This furthers the established pattern of integrating more of a programmer's activities into a single environment, thereby providing a more responsive production cycle. As with previous innovations, and as indicated by its name, Turbo Pascal—released in the 1980s—placed an emphasis on the speed that code could be compiled and ran. As a piece of software, Turbo Pascal was both commercially and functionally successful. Unlike Maestro I which was designed to be rentable, Turbo Pascal was priced for installation on personal computers. Like Maestro I, programming in Turbo Pascal was achieved by entering program code via a keyboard. Turbo Pascal also featured debugging functionality and the ability to include inline assembly language.

Traditional IDEs. With the advent of graphical window desktop environments, IDEs underwent another transformation, transitioning from the text-based interfaces like those seen in Turbo Pascal (Figure 3.1) to the IDEs we use today. Microsoft Visual Studio and Eclipse are the two most popular IDEs [24], with Eclipse being preferred by over half of Java developers [21]. By taking a historical look at the development of IDEs we have been able to see the breadth of innovation they have progressed through. There is little need for the average modern-day programmer to think about the process of compiling and linking their code. Integrated debugging, as seen in Turbo Pascal, is now commonplace. Furthermore, functionality such as content assist and hyperlink marked up code, and graphical user interfaces (GUIs) with mouse interaction have further accelerated the development process.

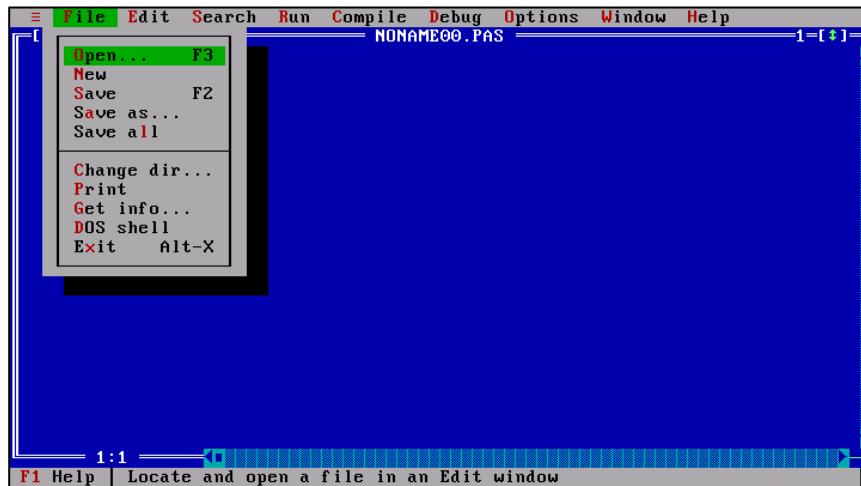


Figure 3.1: Screenshot of Turbo Pascal.

An examination of the last couple decades shows that innovation in traditional IDEs has slowed. Eclipse 1.0, developed by IBM, was released open source in November of 2001,² shortly followed by Eclipse 2.0 in June of the following year. As noted by Nackman, the vice president of product development at IBM, it was at this point that it became clear that Eclipse would be successful [21].

² Archived versions of the Eclipse IDE: <http://archive.eclipse.org/eclipse/downloads/index.php>

A technical overview, which was initially produced for Eclipse 1.0 and then updated for Eclipse 2.1, documents the features that Eclipse provided to programmers in its formative years [25]. It lists support for:

- The creation and management of Java Projects.
- The browsing of Java Projects that are arranged hierarchically using Java elements such as packages, types, functions and fields.
- Editing Java code. This includes the code formatter, code completion and error messages.
- Refactoring Java code such as function extraction.
- Searching Java Code with results hyperlinked to appropriate locations.
- Comparison of two Java files, presumably for the purposes of version control.
- Compile, run and debug functionality.

Innovation since then is difficult to point to. All of the functionality in the above list has been iterated on and as a result improved, however additional functionality or significant changes to the environment is left to the realm of plugins. While the plugin system of a traditional IDE can be impressive [26, 27], it relies on programmers being aware they want certain functionality and the ability to locate and install the appropriate plugin. In place of a historical view on Visual Studio, it is sufficient to notice that the feature set in all traditional IDEs is similar. Task tracking, GUI creation and integrated testing are examples of innovation that builds on the above list. However, they tend to be separate, specially created widgets with minimal impact on other parts of the environment.

The remaining content of this section is divided into two parts. In order to establish what the core functionality of a traditional IDE is, and to point out the similarities between them, the first part documents and describes important aspects of IDEs. In the second part, we then perform a brief evaluation of the user interface (UI) associated with the functionality that part one listed. We note the lack of flexibility and difficulty in making minor changes to this UI.

3.1.2 Integrated Development Environment Functionality

Project Management. Traditional IDEs such as Eclipse and Visual Studio reflect the hierarchical structure of the languages they support. Purpose-built panels, such as the Outline or Package Explorer in Eclipse and Solution Explorer in Visual Studio use collapsible tree widgets (as seen in Figure 2.7) to reflect the languages hierarchical structure. These panels provide a visual way to browse the content of a project. When combined with hyperlinked markup they also provide a way to navigate to the file that results from that browsing. For example, in Java, the top level component is a project. Projects contain references to directories such as 'src' and 'bin' which contain source code and binaries respectively. Source files are then arranged into packages. Each of these levels of abstraction is represented in Eclipse's Package Explorer. The completely collapsed tree widget will show the name of the project. Expanding the tree widget one level will reveal the directories containing source code and binaries. Further expanding the tree by opening the representation of the source code directory will provide the programmer with access to the packages and source files of the project.

Support for other project management tasks such as managing external libraries and maintaining the classpath is also provided. In contrast to the GUI panels that provide browsing and navigation over a project, these tangential tasks use secondary windows that are accessible through a series of menus. This lower level of accessibility is indicative of how frequently the developers of traditional IDEs believe it will be used.

Syntax Highlighting. Traditional IDEs use syntax highlighting to increase the readability of code. By assigning specific colours to specifically recognised tokens, an IDE assists the programmer by providing visual cues about the state of the code. Syntax highlighting allows a programmer to more easily comprehend large fragments of code or easily confirm they have entered a recognised keyword. For example, under default settings, both Eclipse and Visual Studio set the font colour of comments to green. This allows a programmer to mentally segregate comments from code, reducing the effort required in code understanding. This has been shown to have a positive effect by Sarkar [28].

Problem Reporting. Similar to syntax highlighting, error reporting has a visual effect on the code. When an IDE is informed (by the compiler) that there is a problem with the produced

code, it flags the offending tokens by highlighting them. The two most common categories of problem reporting are warning and error notifications.

Traditional IDEs take this process a step further. As an acknowledgement of the fact that warnings and errors can occur on tokens not currently visible on screen, a purpose-built panel—‘Problems’ in Eclipse, ‘Error List’ in Visual Studio—lists all the problems currently being reported. The items in this list are hyperlinked to the location of the problem. A programmer can activate this hyperlink by double-clicking on the list item. Doing so brings the file containing the problem into focus and causes the text area to scroll to the appropriate position in the file.

Content Assist. Content assist, also commonly referred to as code completion, provides a programmer with a list of syntactically valid options for completing a statement, reserved keyword or language construct. One way content assist can be used is to accelerate the typing of long member names. Consider a function with the name “parseJavaCodeFromString”. Instead of scrolling through a potentially long list of suggestions from the IDE, the programmer can make use of the fact that camel case (medial capitals) is being used and can instead type the first letter of each word—p-J-C etcetera. As the programmer types, the list of possible options is filtered to include this new context. By pressing enter, the IDE will complete the string that programmer had selected in the provided list.

Some completion results will contain structure themselves and the IDE may position the cursor and alter the behaviour of the tab key to assist with further completion. Extending our previous example, a complete function signature could be “parseJavaCodeFromString(String, String, int)”. When the programmer has pressed enter to select the correct suggestion, the IDE then places the cursor in the position of the first parameter so that the desired value may be entered. Instead of inserting white space, pressing the tab key will then move the cursor to the position of the next parameter.

Continual Compilation. During the introduction to this section, we mentioned the system WATFOR and how it supported the use of FORTRAN77 by simplifying the compile, link and go processes. Traditional IDEs take this a step further by periodically compiling the code in the background and maintaining rich data structures such as abstract syntax trees. These

data structures are used to provide problem reporting and content assist in a timely manner.

Run & Debugging. A programmer is able to execute the code they have written from within their IDE. Not having to perform work outside of the IDE makes it easier for programmers to make incremental additions to their code, checking their results as they go.

When a programmer's code is producing unexpected results, the integrated debugging system can be used to investigate. Programmers are able to set breakpoints and execute code in a stepwise fashion, investigating the changes in data as the program runs. A purpose built panel allows for code inspection and contains controls for performing steps. When debugging a console application standard out and standard error are redirected to a purpose-built panel, referred to as "Console" in Eclipse and "Output" in Visual Studio.

Code Formatting. Traditional IDEs provide functionality aimed at keeping produced code tidy and easy to read. Upon starting a new line, a traditional IDE will insert a number of tabs to the head of the new line. The number of tabs that are inserted depends on the 'depth' of the surrounding structure as defined by the supported language. For example, when writing a Java For Loop in Eclipse, the content between the two braces that represent the scope of the loop, will be one level of depth more than the loop statement itself. Therefore, this internal content will contain one additional tab.

Another form of Code Formatting occurs at the request of the programmer. When the programmer executes a specific key combination the IDE will alter the code to make it more readable. This is achieved by adding whitespace characters such as tabs and newlines. For example, a long line of code may be split over two lines. A programmer is able to alter the behaviour of the code formatter through a settings dialog.

Refactoring. A collection of refactoring tasks exist, aimed at making non-trivial but tedious programming tasks easier by having the IDE perform some of the work. For example, if a frequently used function name needs to be changed, then instead of manually changing the code in each place the function is called, a rename refactoring can be performed. Through the use of an ephemeral menu, the programmer can request that the function name is changed. After supplying the new name, the IDE will automatically find and replace all instances of the old name with the new one. A variety of refactoring tasks are provided,

such as the ability to transform a selection of lines into a new function or the ability to promote class members to a superclass.

3.2 Abstractions in Programming

Throughout the development of an application, a programmer will make use of abstractions. Some of these abstractions are **rigorously defined**. When developing a piece of functionality for an end user (in the case of an IDE, the end user is a programmer), a rigorously defined abstraction provides a blueprint from which to build—guaranteeing that, if the protocol of the abstraction is carefully followed, the finished product will be widely compatible and understandable. Consider the `//TODO:` tag in Eclipse. A programmer is able to use this tag to leave themselves a note, stating what is left to be done. This tag is an example of a rigorously defined abstraction. The established protocols around this tag allow task tracking extensions to automatically update.

Conversely, other abstractions are **informal**. An informal abstraction lacks a specified protocol, unfortunately meaning that they cannot be relied upon when developing a system for an end user. Informal abstractions provide a lot of flexibility to the end user. They rely on good judgement and consistency from the end user if they are to be helpful. Consider the ability to reorganise tabs—each holding the content of a different file—in traditional IDEs. When developing the IDE, the programmer may not consider the order of the tabs, as specified by an end user, to be important. There is certainly no rigorously defined rule stating that the tabs should be kept in a specific order. This gives the programmer (end user) the option of ordering their tabs in a useful fashion; perhaps placing the tab holding the content of a superclass prior to those holding the content of subclasses.

Sections 3.2.1 and 3.2.2 below each present two examples of abstractions used when developing applications; one of which is rigorously defined and one of which is not. Section 3.2.1 covers abstractions that are used during the planning phase of development and are thus likely distinct from the IDE. Section 3.2.2 specifically covers abstractions used to assist with the authoring of code.

3.2.1 Abstractions in Planning

Rigorously Defined. UML—Unified Modelling Language in full—is a set of rigorously defined abstractions used in software development. An example of a class diagram, one of

the structural diagrams in the UML standard, can be seen in Figure 3.2. The symbols denoting classes, their functions and the relationships between classes are all specified in a standardised set of rules. For instance, a two tailed arrow with a hollow arrow head signifies that the *Leaf* and *Composite* classes are both subtypes of *Component*. The protocol established by the collection of all standardised rules allows applications to provide specialised support for UML diagrams.

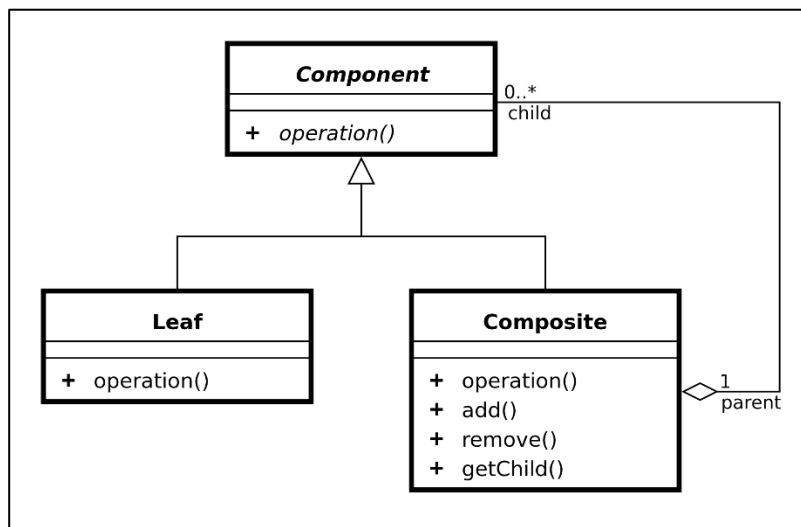


Figure 3.2: A class diagram in UML.

If a programmer wishes to document something in a UML diagram, but is unable to find a way to express it within the rigorously defined ruleset of UML, then they are forced to either forgo strict adherence to the rules or use a less formal diagramming method. If they are using an application that has been specifically designed to support UML, then the first option may not be feasible.

Informal. Instead of using a formal documentation method such as UML, a programmer may decide to use an informal diagramming technique. Such techniques rely on intuition to be widely understood. While informal diagramming may have some basic tool support, such as a canvas on which to sketch, formally defined support is not possible.

Informal diagramming can be anything from class documentation that looks vaguely like UML—but does not conform upon closer inspection—to crude sketches of what an interface might look like. As informal abstractions do not need to abide by a specification, the

environment programmers may produce informal diagrams in can vary widely, from a physical piece of paper (potentially scanned), to a photograph of a whiteboard (potentially photographed), to a professional paint application on their computer.

3.2.2 Abstractions in Code

Rigorously Defined. Consider the Eclipse Outline. One of the stipulations of the Java language is that a Java function must exist within a Java class. This stipulation means that the Outline does not need a way of expressing the idea that a Java function is contained in a package.

Informal. An example of an informal abstraction can be seen when separating two sets of statements from each other with a blank line. For example, blank lines may be used to visually distinguish the base case, processing and recursive case in a recursive function from each other. Note that, the inclusion of a blank line is completely up to the discretion of the programmer. From the perspective of the environment, there is no functional reason to insert a blank line.

In their development of an application to automatically assess the readability of code, Buse and Weimer establish that there is a strong positive correlation between average number of blank lines in a function and code readability [29]. As the average number of blank lines increases, so too does the readability. Inserting a blank line into a function is a spatial behaviour; visually dividing one group of statements from another. Given the relevance of informal abstractions to this research, we wished to assess how frequently blank lines were used in this fashion.

For our experiment we processed a corpus of over 14 thousand Java projects and found prevalent use of blank lines. The corpus of code we used was compiled by Allamanis and Sutton, sourced from GitHub repositories [30]. We wrote a program to count the number of blank lines in each function and calculate how frequently a function contains at least one line. Our analysis program was written by utilising the Java parsing functionality provided by the Eclipse Java Development Tools (JDT). Of the 14,317 Java projects, we were able to parse all but 78 (0.55%) without error. Those that we were unable to successfully parse with JDT were excluded from our calculations.

We analysed a total of 17,675,847 functions, counting a total of 26,305,608 blank lines. Blank lines that occur after the function signature but before the first statement in the body, and those that occur after the last statement in the body but before the closing bracket were not counted. We found that 35% of functions contained at least one blank line—a non-trivial but not especially high figure. Further inspection suggested that functions without a blank line were often very short. This in combination with our finding that functions which contain at least one blank line, contain an average of 4.5 blank lines, suggests that a calculation excluding outliers would significantly increase the 35% figure.

3.3 The Rigidity of Traditional IDEs

Through our analysis of traditional IDEs in Section 3.1 and the use of abstractions in software development as documented in Section 3.2, we are lead to pose the question: where is the support for informal abstractions in IDEs? Abstraction in IDEs, where supported, tends to be rigorously defined and rigid.

In Section 3.2.1 we discussed the use of blank lines when authoring code. While this is an example of informal, and thus flexible, abstraction, it is not an abstraction that is specifically supported by IDE functionality. In other words, even if all the IDE functionality was taken away, the ability to use blank lines as a form of abstraction would still be possible: it is a product of the relative authoring environment that traditional IDEs use.

There are some glimmers of support for informal abstractions that we are able to identify. For example, the ability to reorganise the order of tabs in a traditional IDE as we discussed in the introduction to this chapter. Alternatively, the ability to relocate panels so that they are docked to various sides of the IDE. It is debatable as to whether these abstractions are supported by IDE functionality or coincidentally result from unrelated IDE development decisions.

It is our opinion that IDEs lack support for informal abstractions and that this is a shortcoming. We also note that, some functionality, such as the state of the program while halted at a breakpoint, is not only rigid, but also that it is transient in nature. As the information provided by this functionality cannot be stored, it is not possible to use it for longer term improvements to code. For example, the user is unable to utilise IDE functionality to record the state of the program at a specific time. We seek to address these

shortcomings by consciously maximising the flexibility our Spatial IDE will provide to programmers. With this goal in mind, we now move forward to discuss general purpose authoring—with a focus on Spatial Hypermedia—in Chapter 4.

Chapter 4

Editing Environments and Authoring

This chapter explores and compares various distinct forms of authoring, demonstrating that a set of—often seemingly incidental—design decisions in a specific editing environment shape the type of, and form of, content that it can accept. In order to assist with this comparison a formal descriptive model of the authoring process has been developed. The model introduces the notions of Fundamental Element, First Class Citizen and the data structures used to represent them.

One particular form of authoring that is less widely understood than more traditional authoring environments is Spatial Hypermedia. As we are using Spatial Hypermedia authoring to address the goals of this thesis, we preface the primary content of this chapter in Section 4.1 by defining Spatial Hypermedia through a comparison to a more widely understood concept—Hypermedia. Section 4.2 then describes the formal descriptive model and uses it to analyse traditional text, pixel image, Hypermedia and Spatial Hypermedia authoring environments; the later bridging the definition from Section 4.1 with the model of a minimalistic Spatial Hypermedia application. Section 4.3 then moves from the theoretical to the concrete by reviewing three general purpose Spatial Hypermedia authoring environments. For each of these applications, their citizenry, exploitation of spatial memory and methods for authoring are analysed. For each environment, a discussion of their previous evaluation, as presented in the literature, is included.

4.1 Defining Spatial Hypermedia

As noted in [31], due to the infrequent and varying use of Spatial Hypermedia interfaces, it is difficult to form a precise definition of Spatial Hypermedia that has widespread agreement. In this thesis, we define Spatial Hypermedia as a form of interface that

promotes a users' Spatial Memory by subscribing to the absolute view of space rather than the relative view of space, as explained shortly in this section.

Content in a Spatial Hypermedia system can be spatially positioned to communicate meaning, for example, spatially positioning content into columns to indicate similarity. Furthermore, Spatial Hypermedia gives ample opportunities to create landmarks, both through the thoughtful positioning of important content—such as headings and images—and through emphasis given to text through choices of colour, size and font style. Spatial Hypermedia achieves this treatment of content through flexible building blocks which we call First Class Citizens—defined in Section 4.2. Our assertion is that the rigidity of traditional IDE functionality such as content assist and debugging can be reduced by using First Class Citizens instead of purpose-built user interface components. The reduced rigidity of IDE elements will give programmers more opportunities to adapt their programming environment to suit the tasks at hand.

4.1.1 Absolute and Relative Space

While the term Spatial Hypermedia is not well understood, the term Hypermedia is much more commonly used. The qualifier 'Spatial' suggests that Spatial Hypermedia is a more narrowly defined form of Hypermedia. However, given the way the term Hypermedia is traditionally used, this is not the case. Instead, both Hypermedia and Spatial Hypermedia are a refinement of a broader, unnamed concept. As a solution to this confusion, when referring to Spatial Hypermedia, we are careful to specifically use the term Spatial Hypermedia. When using the term Hypermedia, we are specifically referring to Hypermedia that is not spatial.

Kolb employs an analogy to discuss a key difference between Hypermedia and Spatial Hypermedia [32] that alludes to Leibniz's and Newton's rival theories of the nature of space. Leibniz contended that space was defined by the relationships between the objects that existed in space, whereas Newton considered space to be absolute with objects having their position as a property. This analogy emphasizes the difference between Hypermedia and Spatial Hypermedia: Hypermedia subscribes to the Leibnizian view on space and Spatial Hypermedia to the Newtonian view. Consider web pages, a prevalent example of Hypermedia: the author must consider the order they create content. The position of content will be relative to the content listed before and after it in the HTML file.

In contrast to Hypermedia, Spatial Hypermedia utilises the absolute positioning of content. Information entered into a Spatial Hypermedia system becomes an 'item' that is positioned on a canvas at specific coordinates and therefore the order in which items are created is of no importance—the location of the item is what matters. Editing a slide in Microsoft PowerPoint is an example of this and as such PowerPoint can be considered a form of Spatial Hypermedia.

When applying this analogy to help distinguish between Hypermedia and Spatial Hypermedia the dominant aspect of the program must be considered. Microsoft Word provides users with the ability to insert shapes, text boxes and images at an absolute position (having existing text flow around text boxes and images) and therefore it might be tempting to classify Microsoft Word as a Spatial Hypermedia system. However, text input is the primary feature of Microsoft Word, and that behaves relatively, with the text being reshuffled when new text is added. For this reason, we consider Microsoft Word to primarily subscribe to the Leibnizian view on space and therefore a Hypermedia system, with supporting uses of Spatial Hypermedia. On the other side of the coin a user might look at a collection of slides in Microsoft PowerPoint and decide that, as the slides come in a specific order and are positioned relative to each other, it is a Hypermedia environment rather than—as we have determined in the previous paragraph—a Spatial Hypermedia environment. However, just as we consider absolute image positioning in Microsoft Word to be a secondary feature, we consider the ordering of slides in Microsoft PowerPoint to be of less importance than the ability to edit the content of slides. We recognise that, as slide ordering is necessary in all but the most remedial Microsoft PowerPoint files, this second example is more subjective, however, it demonstrates the difficulty of classifying applications as Spatial Hypermedia or not.

4.2 Content and Meaning

Because of the absolute position of content that is characteristic of Spatial Hypermedia, those wanting to design and build Spatial Hypermedia systems must consider how content is going to be created, interacted with and stored. This section works to provide an explanation of the requirements of a Spatial Hypermedia system as well as address some of the decisions that can be made to maximise the benefits that Spatial Hypermedia provides to authors. To achieve this, we first introduce the idea of a 'Fundamental Element' in an

authoring application—a specific and important component in the application. We then discuss how this can be used to measure the mutability of other kinds of components present in the same application. We examine a progression of authoring systems, starting with traditional text editors and image editors, progressing to include non-textual media with multimedia editors and moving on to Hypermedia to discuss tree-like sequential ordering of elements. For each type of application, we identify its Fundamental Element and use it to evaluate other components of the system. Finally, we discuss the abstract notion of a Spatial Hypermedia system and the implications this has on the requirements for a Spatial Hypermedia's Fundamental Element. During the review of existing Spatial Hypermedia in Sections 4.3 and 5.1 we show how each specific system fulfils and expands on these requirements.

4.2.1 Fundamental Elements, System Representations and First Class Citizens

The Fundamental Element of an authoring system is defined to be:

The primary building block for creating content. For example, a character.

The System Representation of an authoring system is defined to be:

A conceptual data structure specifying properties and operations that can be applied to the Fundamental Element and other authored components; and how the Fundamental Elements are stored.

Traditional Text Editors. In traditional text editors, such as Microsoft Notepad and GNU Emacs the only, and therefore primary, building block is the character. Even whitespace, such as paragraph breaks and indentation, are implemented using specific characters. By the definition provided above, the character is the Fundamental Element of a traditional text editor. Characters are stored in sequence, forming a string. Operations concerning characters are those that manipulate this sequence by inserting, removing or replacing characters. Figure 4.1 diagrammatically shows the Fundamental Element (left) and System Representation (right) of a traditional text editor, taking into account the above description.

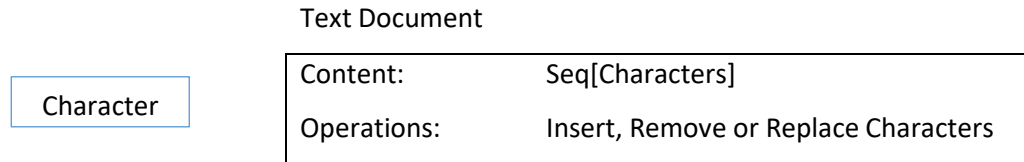


Figure 4.1: The Fundamental Element of a traditional text editor and its System Representation

In general, to discuss the functional capabilities of applications, we make—what we believe to be fair—judgements on what operations to include when talking about System Representations. These judgements are made based on how important they are to a user’s experience. For example, a traditional text editor needs to print characters to the screen using a font. However, it is reasonable to omit the ability to change fonts, or even specify the presence of a font, in the System Representation—at least in a traditional text editor. In applications, such as Microsoft Notepad, a user rarely considers the font that is being used. Furthermore, lacking the ability to change the font of individual characters, if the user wished to change the font then they would have to change the font used for the entire string. Conversely, we include the operation to replace characters in Figure 4.1 (right). Technically speaking, the replace functionality could be implemented as a pair of remove and insert operations. However, from the user’s perspective, the ability to replace a stream of characters is a common editing task and one that a user perceives as close to atomic in operation. Therefore, we include the replace operation in the System Representation.

When discussing authoring applications, we use the term ‘citizen’ to refer to components in the system used for authoring. We use the term ‘information space’ to refer to the collection of citizens that convey information to the authors/readers. The operations that can be applied to manipulate the citizens or their ordering alter the information space. As alluded to earlier, in traditional text editors the only citizen is the character. As we discuss other systems we will have a more varied collection of citizens and therefore a more complicated information space. When this occurs, we will distinguish between those citizens which are first class and those which are not.

We define a First Class Citizen in an authoring system to be:

A citizen of the system that can be manipulated in same ways as the Fundamental Element.

We adopt this term—First Class Citizen—from the programming literature [33] and adapt it to suit our needs. A function is considered a First Class Citizen in its programming language if it can be used in a similar fashion to other elements in the language. Given a specific programming language, if we consider variables to be First Class Citizens, then for a function to be considered the same it must be usable in the same ways variables are. For example, a programming language that allows functions to be passed as parameters (unevaluated), in the same way as variables, treats its functions as first class citizens.

Pixel Image Editors. Pixel image editors—such as Microsoft Paint—give users the ability to modify individual pixels in an image. Like traditional text editors, it is an example of a homogeneous editing environment, this time for modifying images. The Fundamental Element of an image editor, pixels—unlike their counterpart characters—can only be replaced. Functionality such as the ability to resize an image may be considered operations that add or remove pixels from an image, however, these are not atomic operations and result in new images. This observation allows us to specify the data structure for holding pixels to be an array (as opposed to a sequence), specifically a 2-D array. Figure 4.2 shows the Fundamental Element (left) and System Representation (right) of a pixel image editor.

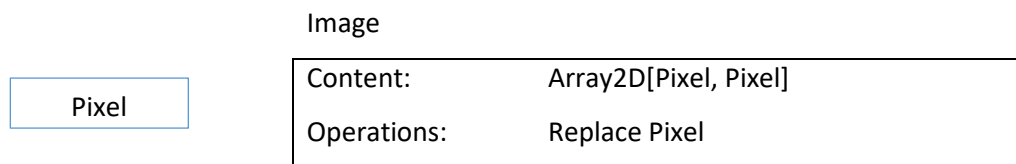


Figure 4.2: The Fundamental Element of an image editor and its System Representation.

Multimedia Editors. A multimedia editor—such as Microsoft WordPad—extends a traditional text editor by adding non-textual media such as images; transitioning the editor from supporting homogeneous to heterogeneous citizens. To help with the progression of editors being described we assume minimal functionality in this multimedia editor. Text is plain, without formatting, hyperlinking or nesting. Images are embedded into the sequence of citizens, which is predominately text in practice.

Characters remain the Fundamental Element of the system. This means that a multimedia editor must retain the ability to add, remove or replace characters in a sequence of citizens that make up the information space. Figure 4.3 shows a diagrammatic representation of the citizens (left) and System Representation (right) for our minimal multimedia editor. The box

representing characters in the diagram is outlined in blue to emphasise that it is the Fundamental Element.

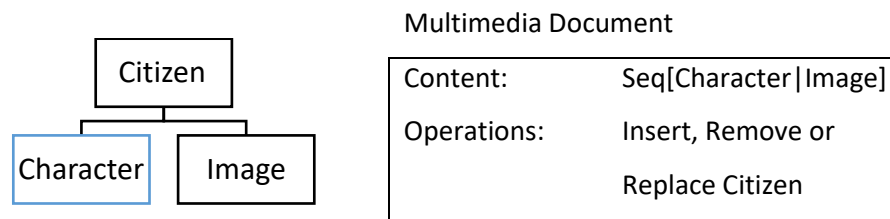


Figure 4.3: The citizens and System Representation from a minimal multimedia editor.

The inclusion of additional citizens (as compared to traditional text editors) has resulted in two changes. The first is that each additional citizen has its own set of operations. The new operations that come with additional citizens can be used to decide if they are First Class Citizens. In our example from Figure 4.3, the operations of the two citizens (characters and images) are identical. Because of this, images from that example are considered First Class Citizens. Should a system not support certain operations on an image then this would no longer be the case. For example, a multimedia editor application that did not allow images to be copied, but did allow characters to be, would not have images as First Class Citizens.

The other change is that the data structure used to store citizens has had to be altered to cope with non-textual citizens. This has been achieved by changing the data structure from a sequence of characters to a sequence of characters and images. It is important to note that the data structure is still a sequence; indicating that the content is relatively positioned. For example, an image is not positioned at absolute coordinates, rather it is positioned between two other citizens, such as characters. Both changes are represented in the example System Representation in Figure 4.3.

Hypermedia. When examining hypermedia systems, we see that their structure is more complicated than those discussed earlier. Let us use a web page built using HTML as an example. We want to consider applications designed specifically for authoring web pages. While many of us have become accustomed to editing HTML syntax directly using a text

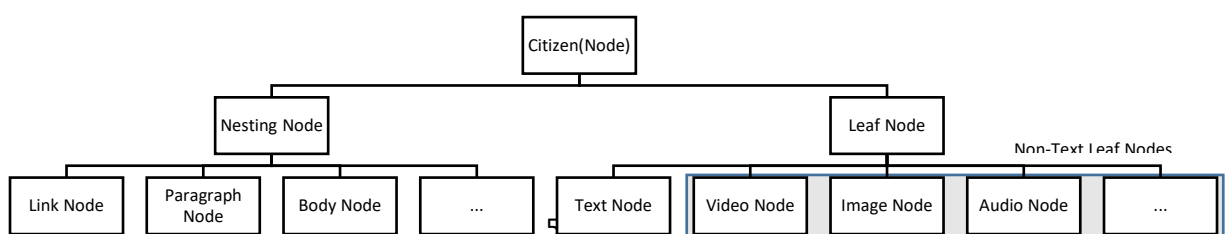


Figure 4.4: The citizens of a web page built using HTML.

editor, this is not the best way to conceive of HTML for the purposes of our discussion. Instead, consider applications such as Adobe Dreamweaver that include specifically designed functionality for authoring web pages.

A web page can be viewed as a collection of nodes arranged in a tree [34]. An editor for web pages needs to be able to understand—and present for modification—this tree structure. The application need not explicitly show the tree structure to the author, but it must show some representation of it. Figure 4.4 shows a selection of possible Nodes and hierarchically arranges them per their use. We classify some nodes to be nesting nodes and others to be leaf nodes. The defining feature of a nesting node is that it contains a sequence of other nodes. This allows for the tree structure to be built. Nesting nodes affect the structure of the content rather than the content itself.

Consider Figure 4.5, which shows an example of a paragraph node. A paragraph node is a nesting node because it may contain several other nodes. This example contains three child nodes: a text node,

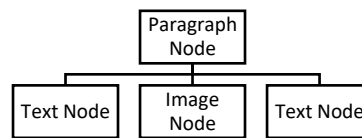


Figure 4.5: A paragraph node structuring an image and some text.

followed by an image node followed by another text node. The ordering of the nodes is significant. When the web page is displayed (assuming no style sheets or additionally executed code such as Java Script), the image produced by the image node will occur after the text produced by the first text node and before the text from the second text node.

Figure 4.6 shows the System Representation for the abstract HTML editing environment we have discussed. It showcases the nesting ability of nodes, the operations each node can execute and properties that nodes can contain. The complete document is represented by a single root node that contains—as its content—a sequence of other nodes. Some of these nodes are nesting nodes, which is denoted with the subscript ‘NNode’. Each node has the ability to insert new or remove existing nodes from its own content; allowing nodes to form a hierarchical structure. Each node also has a type, specifying its function. The highest-level parent node will have a type marking it as such. Finally, each node has a set of adjustable properties.

Hypermedia Document

Content:	Seq[Link _{NNode} Paragraph _{NNode} Body _{NNode} ... Text]
Content.Operations:	Insert or Remove Nodes
Type:	Link _{NNode} Paragraph _{NNode} Body _{NNode} ... Text
Properties:	Set[Align Class Font ]
Properties.Operations:	Replace Property Value

Figure 4.6: A System Representation for an abstract HTML authoring application.

As with the other editing systems that we have discussed, we would like to identify the Fundamental Element for an HTML editing system, and by extension, some First Class Citizens. Previous systems that we have examined used characters as their Fundamental Element. However, HTML diverges from these systems by introducing the ability to structure content, which it achieved through a node system, as is shown in Figure 4.4. The nesting nodes, are wrapped around leaf nodes to provide the completed web page with structure. As the presence of structure is a significant change, and the structure is achieved through the use of nesting nodes, it follows that the Fundamental Element to be some form of node, be it a leaf or nesting node. There are three types of candidates to choose from. Table 4.1 lists the three candidates and highlights the differences between each. Each of these is a candidate for Fundamental Element. By examining the differences between them we are able to identify which is suitable for such a designation.

	Can Alter Content	Have Properties
Structural (Nesting)	✓	✓
Non-Text Content (Leaf)		✓
Text Node (Leaf)	✓	

Table 4.1: A comparison of three types of nodes in HTML.

The first candidate is any of the concrete nodes under the category of structural nodes. We will use a *div* node for the purpose of explanation, but similar logic can be applied to any other structural node, such as paragraph, link or body nodes. Structural nodes provide structure to the document and are exclusively nesting nodes. They have the ability to alter their own content, by inserting and removing other nodes from their sequence. They also have properties that can be replaced. Figure 4.7 shows what the Fundamental Element

would look like if it were a structural node, for example, the *div* node. As the ability to edit content is important in an editing environment, a positive of using a structural node as the Fundamental Element is that all First Class Citizens would require the ability to alter their content. For *div* and other structural nodes, this means the ability to insert or remove nodes from their sequence. However, as a *div* node has properties, all other First Class Citizens must also have properties, which would preclude *text* nodes. This is undesirable, therefore, ideally, an alternative should be found.

Div Node

Content:	Seq[Link _{NNode} Pargraph _{NNode} ... Text]
Content.Operations:	Insert or Remove Nodes
Type:	Div _{NNode}
Properties:	Set[Align ]
Properties.Operations:	Replace Property Value

Figure 4.7: Proposed Fundamental Element – paragraph node.

The second candidate is any of the concrete nodes under the category of non-text content nodes—such as images and audio. The ability to edit non-text content is not typical of functionality that would be expected in an HTML authoring environment—typically they are edited in an external application, such as a pixel image editor. While we are discussing a theoretical HTML authoring environment, we choose to retain this restriction. As we have established that editing is important to an authoring environment, it follows that non-text content is not suitable as the Fundamental Element.

The third candidate is text nodes. Like structural nodes, text nodes can edit their content. We can think of text nodes as a ‘mini-world’ traditional text editor as specified in Figure 4.1. This means that we can consider text nodes to be a sequence of characters with the ability to add, remove or replace characters in this sequence. Unlike structural nodes, text nodes do not contain properties. Figure 4.8 shows the Fundamental Element of an HTML authoring environment if a text node is used as the Fundamental Element.

Text Node

Content:	Seq[Character]
Content.Operations:	Insert, Remove or Replace Characters

Figure 4.8: Proposed (& accepted) Fundamental Element – text node.

By using text node as the Fundamental Element, we overcome the issue we had when using a structural node instead. That is, now both text nodes and structural nodes can be considered First Class Citizens. This is because they are both editable—where text nodes can have their sequence of characters altered, structural nodes can have their sequence of nodes altered. While structural nodes contain properties and text nodes do not, our definition for a First Class Citizen uses the requirements presented by the Fundamental Element as a minimum. The nesting nodes are seen as building on top of the requirements specified by the Fundamental Element. As non-editable elements, non-text content nodes are not considered First Class Citizens. This is logical—as mentioned earlier, these elements cannot be edited in an HTML authoring environment and are therefore less mutable than a text node.

As a postscript to the discussion of text nodes, it should be pointed out that while they do not contain properties, this does not preclude them from being themed. It is possible to alter the appearance of text in HTML indirectly through the properties of surrounding nesting nodes. For example, a nesting node may specify the font of a text node that is its child.

Spatial Hypermedia. In Section 4.3 we will discuss specific Spatial Hypermedia systems and will provide a Fundamental Element and System Representation for each. In this section, we will list the minimal requirements for a Spatial Hypermedia Fundamental Element. As Spatial Hypermedia subscribes to the Newtonian view of space, the position of each Item in the system must be recorded. A logical way to achieve this is for each citizen to store its own position. Figure 4.9 shows the minimum requirements for a Fundamental Element (left) and System Representation (right) of a Spatial Hypermedia System. We assume the minimum require of two dimensions of space. Specific Spatial Hypermedia systems may contain additional dimensions; which they may choose to represent as an additional coordinate in their position property.

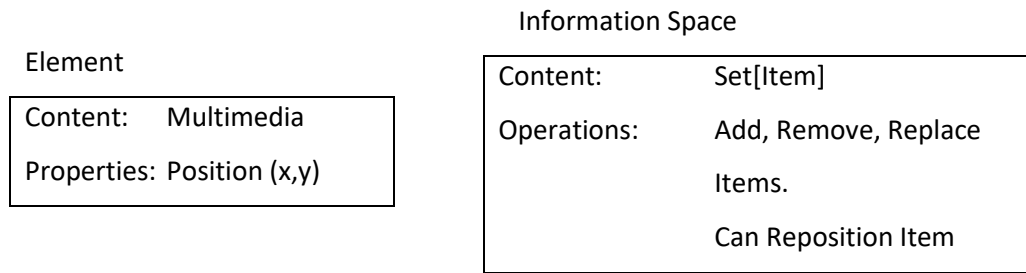


Figure 4.9: The Minimum Requirements for a Fundamental Element and System Representation of a Spatial Hypermedia System.

In contrast to the Fundamental Element and System Representation of the specific form of Hypermedia previously reviewed (HTML), the Fundamental Element and System Representation of this minimalistic and theoretical Spatial Hypermedia system seem quite simplistic. This is partly because hierarchical structure is not a necessity in Spatial Hypermedia, and partly because the additional operations and properties that specific Spatial Hypermedia systems may feature are not listed here. The primary change between Spatial Hypermedia and the systems previously reviewed is the notion that the position of an element is recorded. Sections 4.3 and 5.1 review multiple applications with Spatial Hypermedia functionality. In doing so, the above Fundamental Element and System Representation will be expanded to include the properties and operations specific to those applications.

4.2.2 System Representation as Applied to Meaning

The decision of an authoring environment to use relative or absolute space has an impact on how meaning is communicated in the documented produced by the editor. In editing environments that subscribe to the Leibnizian view of space—such as traditional text editors and word processors—content is stored as a series of elements, frequently characters. Regardless of the reality of implementation, users can reason about the content as being stored sequentially—in terms of implementation, this naturally maps to a list data structure. A consequence of this is that the ordering of elements is uniquely defined. Content earlier in the document produced in a traditional text editor can be considered to appear earlier in the data structure and vice-versa for content occurring later. The sequential storage in traditional text editors is evident from the ability to move the cursor forward and backwards through the document with the left and right arrow keys on the keyboard.

A relative-based authoring application often provides the user with the illusion of random access to anywhere in the document. For example, a user may be able to position the mouse at a (x, y) position between any two characters and have the cursor move there by clicking. This is not an atomic operation. When the click occurs, computation must be carried out to relate the absolute positioning of the mouse to a relative position within the System Representation being used for the document. A similar situation occurs when using the up and down arrows on the keyboard to navigate between lines in a file. Evidence of computation, in this case, can be seen by noticing that different applications can produce different results when dealing with non-monospace fonts. This behaviour of having to perform computations occurs frequently when applications move between relative and absolute aspects of their system.

Editors can exploit the fact that the ordering of citizens is uniquely defined. For example, Microsoft Word can scan for grammatical errors because the ordering of content is expected to make sense. This means that the ordering of content is tied to the meaning of that content. A corollary of this is that, in an environment that subscribes to the Leibnizian view of space, adjusting the ordering of citizens is likely to change the associated meaning.

Conversely, in editing environments that subscribe to the Newtonian view of space—Spatial Hypermedia—citizens have no ordering imposed on them by the system. In terms of implementation, this naturally maps to a set data structure. This is because the order in which content is stored has no effect on the associated meaning. Earlier in this section, we stated that the minimal requirements for the Fundamental Element in a Spatial Hypermedia system was to store its content, position and ability to be added, removed or replaced. For the purposes of explanation, we omitted one other required field: a unique ID. To reason about Items being stored in a set, each must be unique. When working in a Leibnizian subscribed authoring environment, the ordering of citizens removes the possibility of two being identical. However, in Spatial Hypermedia, if two citizens have the same content, are formatted identically and are positioned in the exact same spot, we need a unique ID to differentiate them. The revised version of a Spatial Hypermedia Fundamental Representation can be seen in Figure 4.10.

Item
Content: Multimedia
Properties: Position (x,y)
ID

Figure 4.10: The (revised) minimum requirements of the Fundamental Representation in a Spatial Hypermedia system.

Because the ordering of citizens in the set data structure does not influence the meaning of content, the meaning is frequently left for the user to interpret. When it is necessary for a Spatial Hypermedia system to order elements—as it will be in a Spatial IDE, so that content can be serialized for compilation—it is common for the system to have an algorithm that can be initiated on demand by the user to apply meaning.

Once again, consider a slide in Microsoft PowerPoint. A single element on that slide can be thought of as belonging to a set of all elements in the slide. The author spatially arranges content so those viewing a presentation of the slideshow will infer the ordering that the author wants. The system does not impose an order of the elements. However, at the author's direction, Microsoft PowerPoint can control the order in which elements appear on the screen through animation; an example of an algorithm being used to apply meaning.

4.2.3 Content and Meaning Discussion

By way of summary, Table 4.2 shows a range of document authoring systems. This table covers and expands upon the systems already discussed in this section. For each application, we specify:

1. How content is positioned in the information space: either relative to existing content or absolutely positioned.
2. The type of data structure that is used to store its citizens. For clarity, we consider: sequences to be variable in length and ordered; arrays to be fixed in length and ordered and sets to be variable in length, unordered and with no repeated members. We also specify if nesting is permitted in the System Representation.
3. How ordering (or lack of) affects the meaning of the content. An application may enforce that the ordering of citizens determines the meaning, which we refer to as 'Sequential'. An application may leave the ordering—and therefore meaning—unstated, allowing those viewing the document to decide on the meaning of

content, which we refer to as 'User Inferred'. An application may allow the user to initiate an algorithm to establish an ordering and therefore meaning, which we refer to as 'Algorithmic'.

4. What the Fundamental Element of each system reviewed is. This is decided by treating the application as a black box—that is, we do not examine the code and instead decide based on interactions with the GUI of the system.
5. What other First Class Citizens are in the system. The Fundamental Element is always a First Class Citizen.

Points 1-3 are interconnected. For example, the relative positioning of citizens in the information space, an ordered data structure and sequential relationship between ordering and meaning are all suggestive of each other. Points 4 and 5 are also connected, the Fundamental Element is required to decide which citizens are First Class.

Microsoft OneNote—one of the more recent Desktop Applications in the Microsoft Office Suite—is included in the table. It is marketed as an application for note-taking. While it has not been mentioned up to this point, it has similarities to the general purpose Spatial Hypermedia systems that will be reviewed in Section 4.3. We therefore include it here for subsequent comparison. We also split Microsoft PowerPoint and Microsoft Word into two entries to illustrate the notion that applications have dominant and secondary uses.

	Positioning		Data Structure				Ordering/Meaning			Fundamental Element	Other First Class Citizens	Multimedia
	Relative	Absolute	Sequence	Array	Set	Allows Nesting	Sequential	User Inferred	Algorithmic			
Microsoft Notepad	✓		✓				✓			Character	-	
GNU Emacs	✓		✓				✓			Character	-	
Microsoft Paint	✓			✓			✓			Pixel	-	
Minimal Multimedia Editor †	✓		✓				✓			Character	Image	✓
Microsoft WordPad	✓		✓			✓	✓			Character	Non-Text Media	✓
Microsoft Word												
-Text Authoring (Dominant)	✓		✓			✓	✓			Character	Non-Text Media	✓
- SmartArt/Line Drawing		✓			✓			✓	✓	Shape	Diagrams	
HTML Authoring †	✓		✓			✓	✓			Text Node	Nesting Nodes	✓
Microsoft OneNote		✓			✓			✓		Text Item	Non-Text Items	✓
Microsoft PowerPoint												
-Slide Editor (Dominant)		✓			✓			✓	✓	Text Item	Non-Text Items	✓
-Slide Organiser	✓		✓				✓			Slide	Section	

Table 4.2: A Range of traditional authoring applications. For each: How content is positioned in each and how this relates to meaning. The types of citizens that exist within each authoring application.
† Abstract application for the purpose of discussion.

4.3 General Purpose Spatial Hypermedia

We now review three research-led modern general purpose Spatial Hypermedia systems. While applications such as HyperCard [35] helped popularise Spatial Hypermedia, we choose to focus on more recent applications. Furthermore, the three applications reviewed can be thought of as representative of the two categories of Spatial Hypermedia—those

that use Variable Sized versus those that use Fixed Size spatial interfaces. For each system, we begin by providing a brief introduction to the application before moving on to discussing how the concepts previously discussed in the thesis relate to the application. These concepts are:

- Identifying and describing important interface elements. This includes diagrammatically describing the First Class Citizens of the application, how the Fundamental Element builds on the base requirements established in Section 4.2, and what the System Representation looks like.
- Discussing Spatial Memory considerations: whether it is a Fixed Size or Variable Sized spatial interface, the presence of/ability to author landmarks and the presence or lack of overview.
- How users can author in the system. Specifically, how content can be spatially arranged.

We end the discussion of each application by examining some evaluation that researchers have undertaken using these systems.

Different Spatial Hypermedia systems use different terms to refer to authored content. As we build SpIDER (our Spatial IDE) by extending Expeditee, we adopt the term Expeditee developers use—Item. This is synonymous with the term citizen that we have been using thus far. Individual components of each system will be referred to by the name their developers gave them, but the collection of authored elements will be referred to as Items.

VIKI and VKB are the first two systems reviewed. The first developed by Marshall et al. [36, 37] at Xerox Palo Alto Research Center and the second by Shipman et al. [38, 39, 40] at Texas A&M University. Lessons learned from the development and analysis of VIKI saw VKB developed as its successor, and as such, they share many similarities. Together these systems represent a category of Spatial Hypermedia that utilises a scrollable canvas to contain information, forming a Variable Sized spatial interface. Aspects, such as the scrollable canvas, have parallels with how Microsoft OneNote works.

Expeditee is the third general purpose Spatial Hypermedia system to be reviewed. Developed at The University of Waikato, as the open source successor to KMS [41], Expeditee contrasts with VIKI and VKB by providing a Fixed Size spatial interface and utilising

a linking system to provide limitless space to spatially arrange chunks of content. The result of this is that spatial arrangement happens at a finer level of detail, with some higher-level details getting less on-screen representation.

4.3.1 VIKI

As a Spatial Hypertext system as opposed to a Spatial *Hypermedia* system, VIKI does not support non-textual media such as images. Features in VIKI focus on making it easy to organise information. The information space in VIKI is a scrollable canvas that adjusts in size to accommodate new content as it is added. Information is spatially positioned on this canvas. There are three items that VIKI provides to the user: Objects, Collections and Composites. A screenshot of VIKI taken from [37] can be seen in Figure 4.11. Towards the right-hand side of this screenshot, an example of an Object can be seen. The Objects content is text beginning with “Title: Object description”. Objects can be contained in other items: Collections and Composites. Objects cannot be contained in other Objects. Collections can contain other Collections. This allows for the formation of hierarchical structures.

A series of menus and buttons are arranged along the top of VIKI—outside of the infinite canvas—that can be used to create and theme new Objects, Collections and Composites. Beyond the ability for users to categorise and hierarchically arrange information, one of these menus also provides the user with a spatial parser. This spatial parser—an example of an algorithm being used to order items whose ordering is normally user inferred— can make suggestions to the user for implementing organisational structures based on the layout of content.

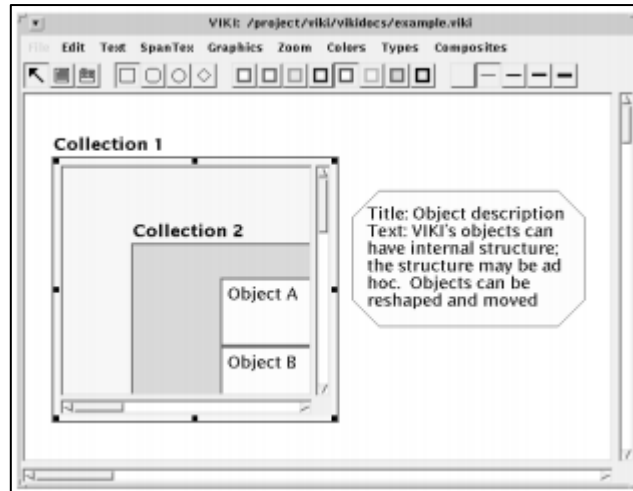


Figure 4.11: A Screenshot of VIKI, showcasing Objects and Collections, taken from [37].

Citizenship. We now expand upon the three items that VIKI provides to its users—the Object, Collection and Composite. As part of this process we specify the Fundamental Element, System Representation and any First Class Citizens. To begin with, we clarify the information space. When starting VIKI, users are initially presented with a scrollable canvas onto which information can be placed—this is the information space. Excluding the menus and buttons present at the

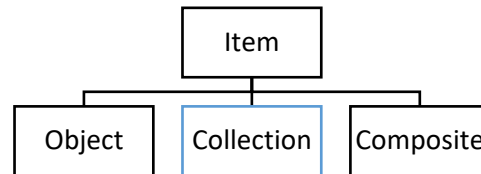


Figure 4.12: The three items of VIKI.

top of the screen, this canvas takes up the whole window. Figure 4.12 shows the three Items in VIKI. As we will explain, the Fundamental Element of VIKI is the Collection. As we have done previously, we have highlighted this by outlining Collection in blue.

Objects are the Items VIKI provides for storing text. They can be spatially positioned inside Collections and can be themed by changing aspects such as colouring, border thickness and shape. When an Object is placed inside a Collection it is at specific coordinates; it is absolutely positioned. A user is able to maximise an Object. This opens a separate window and provides the user with a relative editing environment such as those described in Section 4.2 when discussing traditional text editors. Figure 4.11 contains three examples of Objects. The rightmost object is not contained in a user-created Collection but rather on the initial canvas. This Object is shaped differently from the others for the purpose of exposition; whereas the others are rectangles, this is an 8-sided polygon reminiscent of a curved

cornered rectangle. An array of buttons above the canvas control the shape of created Objects. The other two Objects are contained within a Collection, which itself is contained in another Collection. One of these Objects contains the text “Object A” and the other “Object B”. Figure 4.13 shows a representation of Object using the style established for documenting Fundamental Elements. It extends the minimal requirements for a Spatial Hypermedia Fundamental Element, as described in Section 4.2, by including content, an ID and a position. As such, it is a candidate for being the Fundamental Element. Note that as a Spatial Hypertext system we document its content as text rather than multimedia.

Object

Content:	Seq[Character]
Properties:	Position (x, y)
	ID
	Colour
	Border Thickness
	...
Belongs to:	Composite Nil

Figure 4.13: Representation of a VIKI item: Object.

A Collection is a scrollable canvas—just like the initial canvas provided—which can contain other Collections as well as Composites and Objects. Collections can be used for hierarchical navigation by ‘maximising’ them. When a user maximises a Collection, VIKI performs an action akin to zooming that we will refer to as hierarchical zoom. Hierarchical zoom does not behave in the same way as what is commonly thought of as zooming in a modern desktop environment. Instead of being able to incrementally increase the portion of the screen that is taken up by content, r zoom is all or nothing. That is, when hierarchically zooming into a Collection in VIKI, the entire screen is filled with the Collection that is being entered, effectively replacing the initial canvas or last Collection that was maximised.

Collections cannot contain text but instead store a set of Objects and other Collections. Like Objects, they can also be themed, but not to the same extent. For example, both Collections and Objects can have their colour adjusted, but only Objects can be shaped in one of many ways—Collections are always rectangular. As mentioned earlier, two

Collections can be seen in Figure 4.11, one is titled Collection 1 and the other Collection 2. Collection 2 is contained within Collection 1. As we did with Object, Figure 4.14 shows a representation of Collection using the style established for documenting Fundamental Elements. There are similarities between VIKI's Collection Item and the structural nodes documented in Section 4.2.

Collection	
Content:	Set[Collection Object]
Content.Operations:	Add or Remove Collection/Object Maximise Collection
Properties:	ID
Belongs to:	Composite Nil
Content.Properties:	Set[Position Colour ...]
Content.Properties.Operations:	Replace Property Value

Figure 4.14: Representation of VIKI item: Collection.

Composites exist outside the Collection/Object hierarchy. They cannot be directly created by the author but instead occur as a result of laying content out in a way the system expects. Users are able to specify that certain spatial arrangements are composites and then when the system notices that the author has used this spatial arrangement they will encase the contributing Objects and Collections into a Composite. Composites can then be themed. The user can opt to run VIKI's spatial parser which will then attempt to make suggestions on Composites to adopt. Figure 4.15 shows a representation of VIKI's Composite, comparing it to the representation of VIKI's Collection shows how Composites can be thought of as light-weight Collections.

Composite	
Content:	Set[Collection Object]
Properties:	Set[Colour Border Thickness ...]
Properties.Operations:	Replace Property Value

Figure 4.15: Representation of VIKI item: Composite.

When HTML was discussed in Section 4.2 we recognised that using structural nodes as the Fundamental Element would stop text nodes from being a First Class Citizen due to them not having properties. In the case of VIKI, by comparing their representation, we see that

Collections are similar to structural nodes and that Objects are similar to text nodes. However, Objects in VIKI do have properties. Furthermore, Objects have some properties that Collections do not. It follows then that Collections are the Fundamental Element of VIKI and Objects are a First Class Citizen.

Composites however are not as mutable as Collections; this is due to Composites occurring as a side effect of spatial arrangement instead of direct author intervention. The System Representation for VIKI can be seen in Figure 4.16.

VIKI Information Space

Content:	Set[Collection Object Composite]
Content.Operations:	Add, Remove or Reposition Collection/Object

Figure 4.16: System Representation for VIKI.

Spatial Memory Considerations. VIKI can be considered a Variable Sized spatial interface. Examples of similar spatial interfaces in Chapter 2 used ‘Pan and Zoom’ navigational controls to allow the user to move around the information space. In place of this, VIKI uses scrolling and hierarchical zoom. As Chapter 2 explains, landmarks and the presence of an overview can help a user develop their Spatial Memory of the information space. While landmarks are generally useful in both types of spatial interfaces, they are more useful in Variable Sized spatial interfaces, as the author does not have the edges of the application window to utilise. Overviews however are only appropriate in Viewport spatial interfaces.

In VIKI, each Collection has a title. This title is displayed in bolded text and placed in a stable position at the top left-hand corner of the Collection, making it useful as a landmark. At some level of hierarchical zoom however, no titles will be present. At this point—and possibly at appropriate prior levels of zoom—the author must use the spatial arrangement of content to provide landmarks. Figure 4.17 shows a screenshot of VIKI while inside a Collection. The author has spatially arranged Items to imply groupings and provide headings to each group. These manually created headings can be used as landmarks.

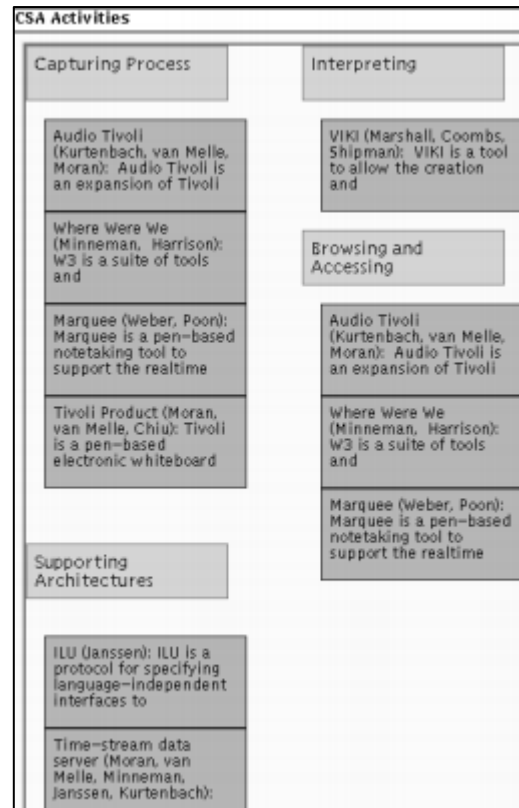


Figure 4.17: VIKI: Inside a Collection, arranged content to provide landmarks. Taken from [37].

Unfortunately, VIKI does not have an overview. An overview must provide users with a low-detailed miniaturised view of the entire information space. The closest functionality that VIKI has to an overview is if the user was to hierarchically zoom out as far as possible. However, even doing this is likely to omit information as some content may be hidden inside a Collection—either not or partially visible until that Collection is maximised. Notice, for example, the Objects in Figure 4.11 are only partially visible.

Authoring. Entering content into VIKI is done by creating an Object and typing in the information you want to store. This is not dissimilar to adding content in an application like Microsoft PowerPoint, substituting Text Boxes for Objects. VIKI's real strength however, is in organising content. Users are able to spatially position Objects and create hierarchical structures using Collections to organise content. When editing content in an Object users are provided with a relative editing environment, like those found in traditional text editors. However, the Item encasing the text, the Object, is still absolutely positioned—to either the edges of the encasing Collection or another Item.

VIKI's spatial parser is able to make suggestions to improve the organisation of information. For example, given the workspace seen in Figure 4.17, the author is able to ask for a suggestion from the spatial parser. VIKI might then notice the pattern of using one Object as a heading, followed by several more spatially indented Objects as members of a list. The suggestion might be to take each spatially organised list and make them each a formal structure—a Composite.

Evaluation. Marshall and Shipman undertook a study to measure the effectiveness of VIKI for information triage [42]. They define information triage to be “the process of sorting through relevant materials, and organising them to meet the needs of a task”. The study used 15 undergraduate students who had just started a course on HCI. The participants had 3.5-17 years of experience with computers and 3.5-11 years of experience with windowing systems. Participants were given 75 articles relating to machine translation packages and asked to recommend one of the packages for use in a fictional company. Participants were randomly sorted into three conditions:

1. Use of the full VIKI system with all 75 articles pre-entered. 10–15 minutes of training to use VIKI was given.
2. Use of VIKI without Collections and with all 75 articles pre-entered. 10–15 minutes of training to use VIKI was given.
3. Use of paper and pen with all 75 articles printed out for them to sort through.

Participants were all given 45 minutes to complete the task on their own. An exit questionnaire was given that confirmed their recommendation to the fictional company and asked their opinion on how their task went.

Many participants were not confident with their results. This was put down to the time constraint of 45 minutes. In general, those using paper were more confident of their recommendation. Two other interesting observations were:

1. Participants created order regardless of their environment. Participants working with paper would create stacks of related material and participants using VIKI without Collections opted to spatially arrange content into groups anyway.

2. Participants less interested in reading all the content and more interested in making the best judgement call in the limited time finished the task with an appreciation of the idea of a system like VIKI.

4.3.2 VKB

Building on top of VIKI, VKB retains most of the same functionality. It retains the use of:

- A scrollable canvas as its primary editing area.
- An item called Collections that allow for hierarchical arrangement of information and hierarchical zoom.
- An item called Objects for storing and spatially arranging information.

It deviates from VIKI by:

- Allowing for the insertion of non-textual data such as images. This promotes VKB to a Spatial Hypermedia system rather than a Spatial Hypertext system as VIKI was.
- Introduces cross-cutting links from one element in the Spatial Hypermedia system to another.

The design of VKB was driven by the observation that individuals using Spatial Hypermedia systems develop their own methods for communicating meaning that others may have trouble interpreting. This causes a problem when the author and the reader of a Spatial Hypermedia system are not the same person. One solution to this problem may have been to provide users pre-built structures for communicating meaning, however this limits the strength of a Spatial Hypermedia system: the flexibility to create your own structures. Instead VKB provides users with the ability to navigate along a time-axis; inspecting the development of the information space as time passes. This gives readers the opportunity to see the author's decisions in the order they are made, hopefully giving the reader the ability to understand the spatial structures the author creates.

Citizenship. The three Items provided by VIKI are retained by VKB—the Object, Collection and Composite. Minor changes have been made, for example, Objects can no longer be multiple shapes. However, the major details discussed when talking about VIKI remain the same. These details include the selection of Fundamental Element, First Class Citizens and System Representation. The addition of the ability to browse through the development of

the information space, however, does provide VKB with some new functionality worth discussing.

Figure 4.18 shows an annotated screenshot of VKB. A significant difference from VIKI is the inclusion of the ‘history toolbar’, which is a slider that is used as the primary method for navigating through time. While the ability to navigate through time and the related controls are not strictly concerned with adding or editing content, they may affect an author’s work by helping them understand the spatial arrangement decisions that previous authors have made. We will discuss the utility of this functionality as well as the problem of different authors and readers in Section 5.2—specifically how these issues relate to the development of a Spatial IDE.

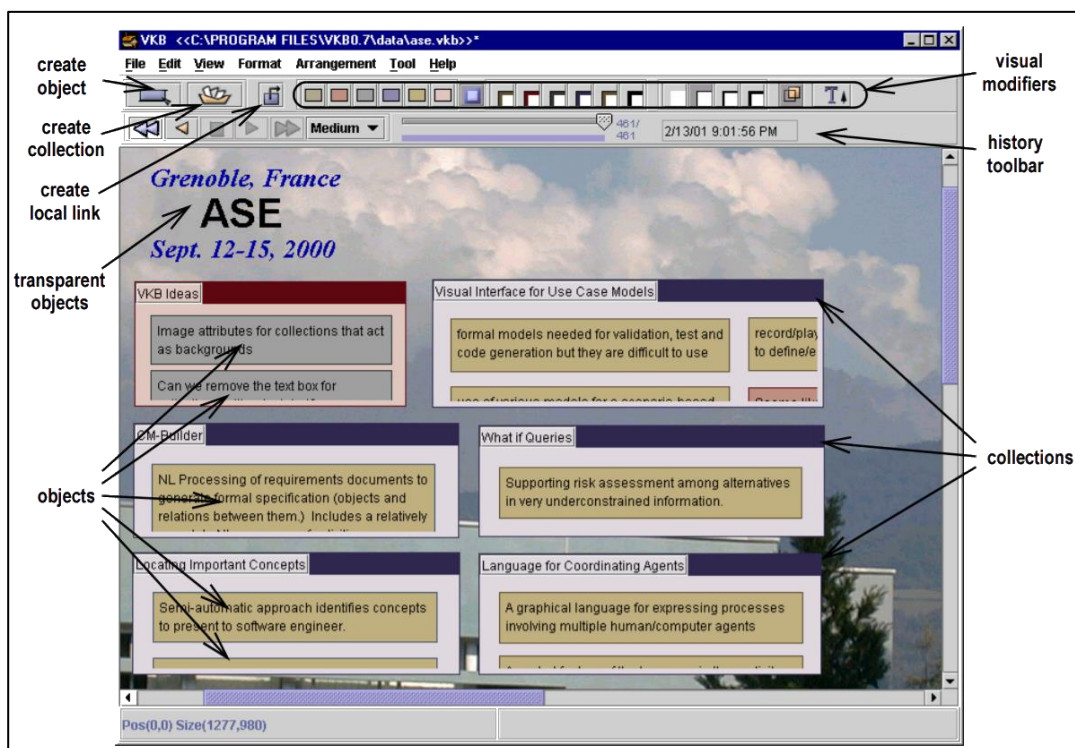


Figure 4.18: An annotated screenshot of VKB, taken from [40].

Spatial Memory Considerations. As with VIKI, VKB provides the user with a Variable Sized spatial interface. The Spatial Memory considerations for VIKI also apply to VKB.

Authoring. Creating content in VKB is very similar to creating content in VIKI. Two points of difference are:

1. Images, not previously available in VIKI, are now supported in VKB. They can be set as the background for a Collection or Object.
2. Improvements have been made to the spatial parser that is used to make suggestions for organisation. The revised algorithm applies some heuristics to Objects based on their similarity to other Objects.

Evaluation. In [39] Shipman et al. present anecdotes concerning the use of VKB. VKB has been used for, the gathering of information by a high school chemistry teacher, report writing by an undergraduate student, project management by research groups consisting of university faculty members and undergraduate students, and organising the ACM Hypertext 2000 conference.

This use has led to refinements to VKB. For example, as a result of complexities discovered by the high school chemistry teacher, VKB was altered to include the ability to create Objects containing information from the clipboard, therefore making it easier to import information from external sources into VKB. Observations concerning the development of a spatial information space in VKB were also obtained by questioning the decisions individuals made. For example, the undergraduate student that used VKB for report writing did so for multiple reports, each taking at least a month. The spatial arrangement that the student came up with for each report differed, suggesting that different layout structures are useful for different tasks/subjects or that increased competence with VKB lead to varied behaviour. This emphasises the importance of retaining the flexibility of Spatial Hypermedia systems.

In [40] Shipman et al. report on an evaluation of VKB that had participants author poems. The goal of the study was to examine how collaborative work differed between a physical environment and a digitally augmented one. Participants, from Texas A&M University, consisted of undergraduate students from the English department and graduate students from the College of Architecture and Department of Management Information Systems. There were eight participants in total and they were divided into four pairs. A commercial product called a Magnetic Poetry set was used by the groups working in a physical environment and VKB by those working in the digital environment. All pairs were instructed to create a poem out of an identical set of words (or word parts): those that were included

in the Magnetic Poetry set. VKB was initially set up with individual Objects each containing the words from the Magnetic Poetry set.

Participants were given 90 minutes to create a poem. This time was divided into four parts. The first and last part had the pairs work collaboratively while the second and third parts had only one member of the pair work. Participants were unable to verbally communicate with their pair when working individually, though they were able to leave each other notes.

Results from a survey taken by participants revealed:

- All participants were familiar with using Windows on computers.
- Three of the four undergraduate English majors had Magnetic Poetry sets at home and the fourth was familiar with it. None of the other participants were familiar with Magnetic Poetry.
- The majority of students were happy with the time limitations of the task. One reported that more than enough time had been given and one reported that not enough time had been given.
- Three of the four participants using VKB found that the most time-consuming aspect of the task was finding specific words. One participant using Magnetic Poetry also reported this to be the case.

All participants were at least 'somewhat satisfied' with the poem they built. These results suggest that a digital interface is a suitable environment for assembling text, but that limited practise time hinders the ability to organise disjoint information in a searchable fashion.

4.3.3 Expeditee

Expeditee is a Spatial Hypermedia system developed as an open source implementation of work previously done by Akscyn et al. [41]. Expeditee supports a wide variety of multimedia forms including text, images and diagrams (line art). Extensions to Expeditee exist to add support for audio authoring [43] and office applications such as spreadsheets [44]. In contrast to previous applications reviewed, Expeditee provides the user with a Fixed Size Spatial Interface. In order to explain how Expeditee maintains a Fixed Size Spatial Interface and still allows for an unbounded quantity of information, we must first examine Expeditee's information space.

Figure 4.19 shows a screenshot of Expeditee designed to showcase commonly used media elements (Items). In Expeditee parlance, we refer to the information space that they are placed on as a Frame. Content is positioned at specific coordinates on a Frame and no scrollbars exist. Together, these two characteristics allow the edges of the application window to be used as anchor points for developing Spatial Memory. Positioned prominently in the centre-top of the Frame is a title graphic that makes use of all the frequently used Expeditee Items. Dismantling the graphic, we see the following Items:

- Three Text Items: “Commonly Used”, “Expeditee” and “Items”. All three of these Text Items happen to use the same font but the middle Text Item is given a larger font size.
- Two Polygons: a pink/red triangle that encases all the Text Items and a yellow rectangle that encases the Text Item with a larger font. The yellow rectangle spans the width of the red triangle.
- An Image that is a caricature of the fictional goddess Expeditee.
- Two Polylines, each flanking one side of the graphic. The right Polyline features an arrowhead whereas the one on the left does not.

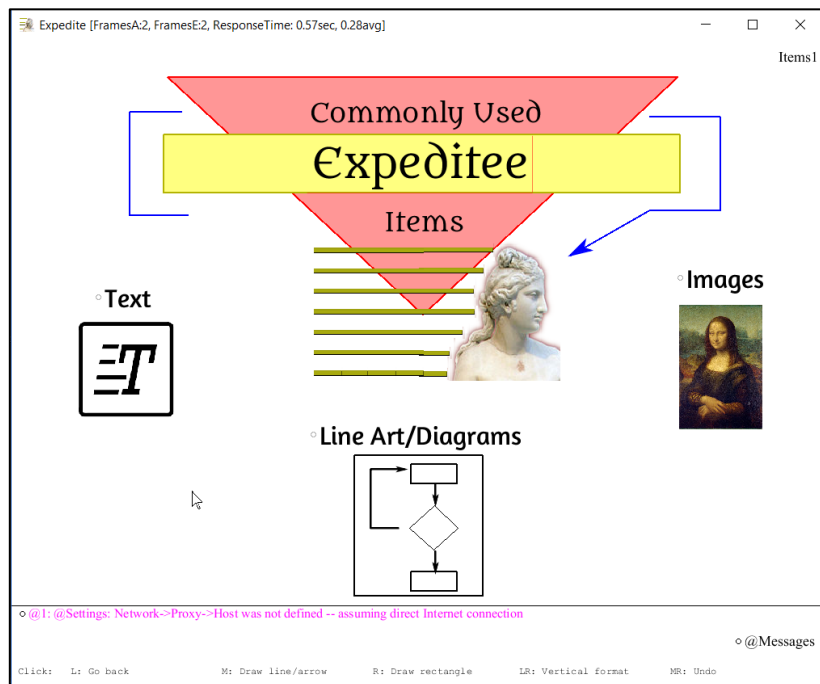


Figure 4.19: A screenshot of an Expeditee Frame showcasing commonly used Items.

Citizenship. Even without extensions to include support for audio content authoring and office documents, Expeditee has a large collection of Item types. For the purposes of explanation, we will discuss some of the more commonly used Items that can be placed on a Frame. For each Item type we will discuss how they fit into the citizenship of Expeditee, culminating by describing the Fundamental Element and First Class Citizens. The Item types that will be discussed are shown in Figure 4.20. An author is able to use these Item types to add text, images, polygons (with additional support for rectangles) and polylines (with or without arrowheads) onto the information space.

The last previous systems reviewed—HTML authoring, VIKI and VKB—all featured some form of nesting. VIKI and VKB used Collections to achieve this whereas HTML had nesting nodes. In contrast, no Expeditee Item explicitly supports nesting. It should be noted however, that Expeditee does feature algorithms that allow nesting to be emulated when the user executes certain actions.

As we will shortly explain, the Text Item is the Fundamental Element of Expeditee and as such—in keeping with previously reviewed systems—we have highlighted this with a blue outline.

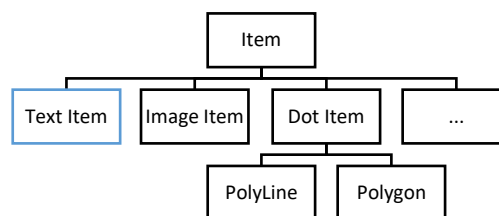


Figure 4.20: The Items of Expeditee

A Dot Item is simply a visual dot on the screen. On their own they are not particularly useful—an author may decide to use them to communicate some specific meaning or use them aesthetically, but this is not their principal role. Rather, Dot Items can be connected by constraints (visualised as a line) to other Dot Items. This allows for the creation of PolyLines and Polygons. A series of Dot Items connected with constraints that does not create an enclosure forms a Polyline. PolyLines can optionally have an arrowhead attached. If an enclosure is formed, then the author has created a Polygon and the enclosure is colour filled in to signify the state change. PolyLines and Polygons share many of the same properties that other Items do, such as colour and size (in this case thickness of line). Figure 4.21 shows a representation of the Dot Item.

Dot

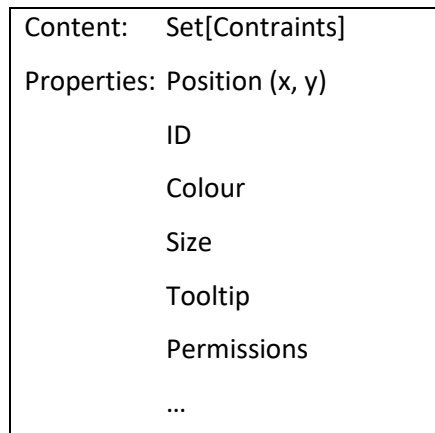


Figure 4.21: Representation of Expeditee Item, Dot—used for Polylines and Polygons.

The capability of creating Polylines and Polygons is useful for the production of diagrams, tables and categories. As rectangles are commonly used in diagrams and other structural components, the right mouse button—in free space—has been set aside as a quick way to create them. Rectangles are also commonly used to temporarily group Items together, allowing them to be moved as a single unit.

As with Object in VIKI/VKB and Text Nodes in HTML, a Text Item can be considered a ‘mini-world’ traditional text editor. When we were discussing the concept of a HTML authoring environment we identified an issue with Text Nodes in terms of uniform editing: their lack of properties. As text is the primary way of communicating information, it is desirable that the citizen representing text be a First Class Citizen. By our definitions of Fundamental Element and First Class Citizen, for Text Nodes to be First Class Citizens, the Fundamental Element of a HTML authoring system must not feature properties. As Text Nodes are the only citizen in HTML that did not contain properties, it follows that the Fundamental Element must be the Text Node. Therefore, the lack of properties on Text Nodes leads to weaker requirements for First Class Citizens. As Expeditee Text Items include properties, they do not cause this same weakening in the requirements for First Class Citizens. Figure 4.22 shows a representation of the Text Item.

Text Item

Content:	Seq[Characters]
Properties:	Position (x, y)
	ID
	Colour
	Size
	Tooltip
	Permissions
	Font-Family
	Font-Style
	...
Belongs to:	Frame

Figure 4.22: Representation of Expeditee Text Item.

Figure 4.23 shows a representation of the Image Item. Whilst not an Image Pixel Editor, Expeditee does support some operations for manipulating images. Notably, the ability to copy a region of the pixels—providing the ability to create a cropped copy of the image.

Image Item

Content:	Array2D[Pixels,Pixels]
Operations:	Move, Scale, Copy region...
Properties:	Position (x,y)
	ID
	Size
	Tooltip
	Permissions
	...
Belongs to:	Frame

Figure 4.23: Representation of an Expeditee Image Item.

Because properties are present in Expeditee Text Items, we are able to identify this type of Item as the Fundamental Element of Expeditee without lessening the requirements for being a First Class Citizen. Furthermore, all other Item types in Expeditee have the same level of editability, making them First Class Citizens. Not only do they all feature a similar set of properties and data structure operations, but they are also manipulated consistently

by mouse controls. Further discussion of Expeditee in this chapter will show other benefits of Text Items being the Fundamental Element in Expeditee.

The system representation for Expeditee can be seen in Figure 4.24. Subscript is used to show that Polygons and Polylines are created out of Dot Items and that Images are created out of Text Items; a detail we expand upon in Chapter 6.

Expeditee Frame

Content:	Set[Dot Polygon _{Dot} PolyLine _{Dot} Text Image _{Text} ...]
Content.Operations:	Add, Remove or Reposition Items

Figure 4.24: System Representation of Expeditee.

Spatial Memory Considerations. We classify Expeditee as a Fixed Size spatial interface because—due to the lack of the possibility for scrolling or hierarchical zoom—an author is guaranteed the ability to relate the position of content to the edges of the screen. To allow data sets larger than one screen, Expeditee uses a Frame and Linking system. Figure 4.25 shows how the Frame and link structure in Expeditee can be visualised. Miniaturized copies of Frame screenshots previously seen in this review are arranged showing how they connect. Arrows are used to show the connections that are formed by linked Items—one Frame contains a link to each of the other Frames. An enlarged section of the image is shown in Figure 4.26 so that the circle that appears beside linked Items can be seen.

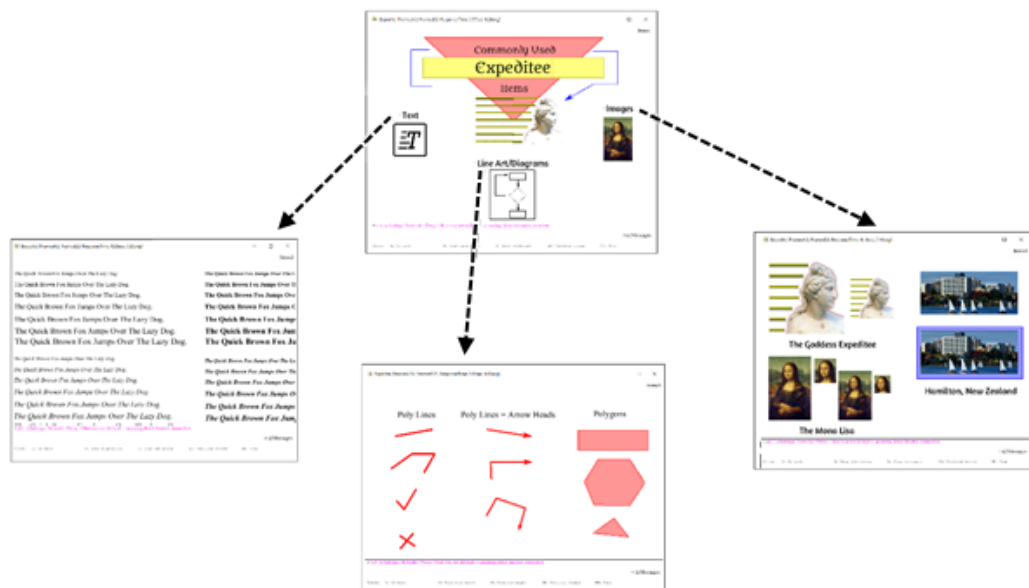


Figure 4.25: Marked up Frame structure of previous examples.



Figure 4.26: Zoomed in Section of Frame structure example.

In comparison to authoring applications that use a variable sized spatial interface, Expeditee's use of a Frame and Linking system allows it to maintain a Fixed Sized Spatial Interface at the cost of likely occluding a greater percent of the total information at any given moment. This trade-off will be discussed in Section 5.2.

On the subject of landmarks and overviews: the ability to author landmarks in Expeditee is roughly comparable to doing so in VIKI and VKB. As stated in the review of VIKI, the titles of Collections have limited use as landmarks. Similarly, Expeditee Frames are generated with a title. At some level of hierarchical zoom, a VIKI information space relies on the spatial positioning of its Objects and Collections to provide landmarks. As VIKI allows users to spatially arrange Objects to author landmarks, so too does Expeditee allow authors to arrange Items. While we have a Fixed Size Spatial Interface in Expeditee, as mentioned in the previous paragraph, not all content will be visible at once. This is another issue to be discussed in Section 5.2.

Authoring. The subject of authoring in Expeditee is large. For the purpose of section, we will focus on a single aspect of authoring that amplifies the classification of Text Items as First Class Citizens in Expeditee: property injection. The subject of authoring in Expeditee is expanded on in Chapter 6.

Figure 4.27 shows a fragment of an Expeditee screenshot. A Text Item is towards the left of the image. By left and right mouse clicking at the same time on this Text Item, the user has obtained a list of its common properties—seen on the right. This same action can be

performed on any Item. The list produced is not all of the properties that the specified Item has, but rather those that the developers have deemed as being used frequently. The process of property injection—soon to be explained—can be used to adjust any property that an Item has. Adjusting an unlisted property on a specific Item will cause it to be included in any list of properties requested from that Item in the future.

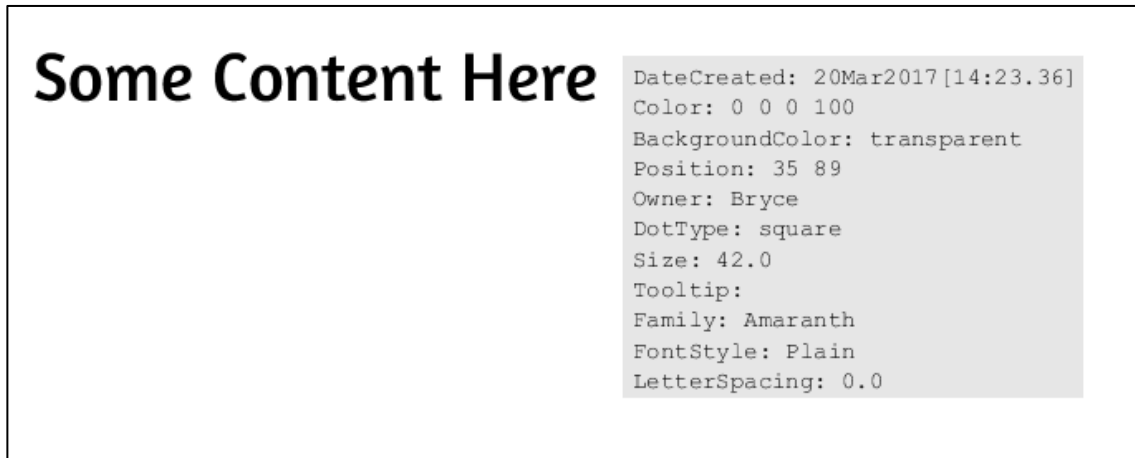


Figure 4.27: A cropped screenshot of Expeditee showcasing the common properties of a Text Item.

Notice that each property is listed as a name-value pair. Property injection is the process of injecting a Text Item that is formatted as one of these name-value pairs into an existing Item, thereby changing that property. This is done by:

1. Creating a Text Item with content that is formatted as a name-value pair where the name is a valid property and the value is a valid setting for that property. For example: *FontStyle: BoldItalic*
2. Picking that Text Item up by attaching it to the cursor with middle click.
3. Hovering the cursor over the Item whose property you wish to change with the previously created Text Item and middle clicking to inject that property.

Figure 4.28 shows a before (left) and after (right) example of the example explained above. It should be noted that, similarly to a link node in HTML, the destination of a link is stored as a property on the linked Item. This means that property injection can be used to create and alter links.

<p>Some Content Here</p> <p>FontStyle: ItalicBold</p>	<p><i>Some Content Here</i></p>
-------------------------------------------------------	---------------------------------

Figure 4.28: Property injection in Expeditee. Before and after.

Property injection gives Expeditee Items a reflective quality. The Fundamental Element—Text Items—can be used to adjust other Items, all of which are First Class Citizens. As we drew on programming literature for the term First Class Citizen, we now also adopt the term Reflection from the same literature [45]. In a programming language, reflection is the process that code uses to analyse and modify itself during runtime. This is analogous to the process of property injection that Expeditee uses to modify itself while running. Due to the fact that we have established that Expeditee has the ability to modify itself through its Fundamental Element we can now strengthen our definition of a First Class Citizen.

Expeditee features Reflective First Class Citizens:

A citizen of the system that can be manipulated in similar ways and be modified by the Fundamental Element of that system.

Evaluation. Built upon the success of two predecessors, initially ZOG [41] and subsequently KMS [46], Expeditee is the result of a long period of iterative development. Developed at Carnegie-Mellon University from 1972 to 1983, ZOG, Expeditee’s initial predecessor, was sponsored by the American Navy Office of Naval Research. Thus, Expeditee, itself the result of nearly a decade of development, in addition to building on the lineage of direct predecessors with their three decades of development, has been subject to forty years of refinement. Furthermore, Akscyn has been a principal architect of all three systems, meaning that the lessons learned from each previous iteration are being applied with first-hand knowledge.

Upon the completion of development, ZOG was deployed on the carrier class ship, the USS Carl Vinson to act as a collaborative-capable system—spanning 28 workstations—to assist with document authoring, viewing and task tracking. For example, the ship policy manual could be viewed and edited and logs of ship maintenance were kept using ZOG. Following

this, another year of collaboration between the ZOG team and crew members of the USS Carl Vinson occurred. This final collaboration both helped refine ZOG (and latter KMS) and gave the ZOG team insights into the benefits of iterative software development that included regular client feedback [47].

In 1981, towards the end of the development of ZOG, a company named Knowledge Systems was formed to create a commercial variant of ZOG. The resulting product, named KMS (short for Knowledge Management System) was initially released in 1983. Used internally for a wide variety of tasks, the developers of KMS estimated their collective usage of the system to be (as of 1988) 10 thousand hours and 50 thousand frames created [41]. These values only represent time spent using the system as a user, notably excluding the time spent developing and testing. Several other companies also worked with Knowledge Systems to utilise KMS in the running of their business; General Electric, Martin Marietta, Tennessee Eastman, GTE and the US NSA [48].

Two significant areas of design in KMS are collaboration and extensibility. In [49] Yoder et al. discuss aspects of KMS designed to assist with collaboration. Topics covered include:

- The simultaneous access and authoring of information by multiple end users.
- The ability to annotate Frames.
- View previous versions of Frames.
- Permission models so that an author may dictate how their content can be edited by others.
- How Frames (and FrameSets) can be used as communication platforms with simultaneous access and authoring [41].

All of this functionality would later be used to direct the development of Expeditee.

Having not identified the personal nature of Spatial Memory as a potential issue (as Marshall et al. do a number of years later, see Section 4.3.2), they do not attempt to address this directly, however the recorded version history featured in KMS and Expeditee has similarities with the solution provided by the developers of VKB.

Having seen the benefits of working closely with the crew on the USS Carl Vinson, the team at Knowledge Systems felt it important that KMS be extensible so that end-users would be able to add or manipulate existing functionality to suit their needs. To this end, a scripting

language named ‘Action Language’ was developed [50]. Described as being block structured and using a simple ‘command line’ syntax, Action Language could be used to create independent chunks of code capable of manipulating and interacting with Frames. These scripts could be authored on and executed from KMS Frames. The fabrication of hundreds of these scripts within KMS, and the experience maintaining them, led Akscyn to believe that a Spatial Hypermedia system may not only be appropriate for programming, but beneficial [48]. Incentivised further by more recent attempts at using Expeditee to program by researchers at the University of Waikato, this observation was what initially spurred this thesis and the development of SpIDER.

When developing Expeditee, Action Language was re-implemented as ‘SIMPLE’. Rather than a direct port of Action Language, SIMPLE prioritised implementation of functionality to that which had proven useful. Authoring and executing SIMPLE is achieved using Expeditee Frames and Text Items. Every statement in SIMPLE is—in Akscyn’s words—“flat” [48]. This essentially means that statements, including those categorised as reserved keywords in other languages, behave as procedures and make use of Expeditee’s Frame and linking system. One notable example is IF statements. When executing, Expeditee will check if the conditional on the IF statement will resolve to TRUE. If it does, and the IF statement Text Item contains a link, Expeditee will move onto executing the SIMPLE code behind that link.

4.3.4 General Purpose Spatial Hypermedia Discussion

As we did when discussing non-spatial hypermedia applications in Section 4.2, we will now present Table 4.3 to summarise Section 4.3. All of the three systems that we have analysed use absolute positioning (a requirement for Spatial Hypermedia) for their Items and algorithms or user inference to apply meaning to their content. For this reason, those attributes are not included in this table. We do however list three new categories:

1. The type of nesting that the application allows. We identify two dimensions of variability:
 - i. How deep Branching can occur. Shallow or Deep.
 - A group of citizens that can contain other citizens, and where at least one of the citizens can contain citizens of the same type is considered to have *Deep Nesting*. A valid example is: given *A*, *B*, *C* are all types of

citizens, if A can contain B , and in turn B can contain C and other instances of B they are considered to have Deep Nesting.

- If the structure of the citizens in question allows (or exhibits) nesting but none of the citizens can contain citizens of the same type then we consider them to have *Shallow Nesting*. The previous example does not exhibit *Shallow Nesting* because the type B can contain other citizens of type B . A valid example of a group of citizens with Shallow Nesting would be: given A, B, C are all types of citizens, they are set up so that A may contain B and B may contain C .

The length of the branch does not affect its classification as either *Shallow* or *Deep Nesting*. Taking the example used to explain Deep Nesting, the number of type B or type C that a type B contains is not considered.

ii. The strength of containment. Strict or Transient.

- If the application has a strong notion of containment it is classified as having *Strict Nesting*. We are primarily concerned with how the application represents nesting in its data structures. An application that stores data to keep track of which citizens are nested within which uses Strict Nesting. However, we conjecture that a good approximation of the strength of containment can be made by examining the application. Typically, Strict Nesting can be identified by examining how nested citizens interact with the citizen they are nested within. If a nested citizen can be occluded through spatial positioning this is a sign that it is subject to Strict Nesting. Another sign that Strict Nesting is occurring would be the presence of a 'snap-to' grid as this suggests the application is monitoring containment.
- Alternatively, an application may use *Transient Nesting*. When Transient Nesting is being utilised the application does not store information related to nesting but rather calculates which citizens are nested within which when specific user actions occur. Like with Strict Nesting, Transient Nesting can be identified by examining the running

application. One way Transient Nesting may be implemented is based on geometry. For example, when picking up a citizen, the application may algorithmically decide which citizens are contained within and pick those up two. If this is the case then minor geometrical changes may change the behaviour. For example, moving a citizen so that it overlaps less with another may cause the two to become 'detached'.

2. The form of Spatial Interface that the application uses. Variable Sized Spatial Interfaces feature techniques that stop the edges of the screen being used to spatially position Items whereas Fixed Size Spatial Interfaces do not.
3. Whether or not the application's First Class Citizens are reflective.
4. How likely and to what extent general use of the application will cause information to be occluded.

Concerning the likelihood of occlusion given general use of an application: It is technically possible to, in all three applications, not have any information occluded. As long as there is space left on screen, an author may choose not to use supported organisational functionality that causes occluded information. In VIKI and VKB this means that a user does not make use of scrollbars or hierarchical zoom. In Expeditee this means that a user does not make use of the Frame and Linking system. Whilst these methods may be suitable under specific circumstances—perhaps for building a poster—they are not generally suitable, and as such, a more meaningful consideration is how likely information is to be occluded when using the supported organisation functionality that is provided. Under general use, all three applications are likely to occlude information.

A cursory exploration into each application reveals that, under normal circumstances, VIKI and VKB are likely to contain more information—when compared to Expeditee—before occlusion becomes likely. Furthermore, at any given time, a larger portion of content is likely to be hidden when using Expeditee. Factors such as the size of textual content and the prevalence of whitespace in Expeditee explain this.

In VIKI/VKB, the presence of scrollbars on Objects provides authors with fine-grained control over the size of an Object. They are able to perform a trade-off—increasing the amount of occlusion whilst reducing the size of an Object or vice versa. On the other hand, in Expeditee, users frequently use a larger than normal font size. It is theorised that the

limited size of an Expeditee Frame, combined with the linking system, encourages users to structure their documents using the Frame abstraction [48]. This in turn provides them with the ability to size content as if they are producing a low-content document—such as a poster—even when they are not.

In VIKI/VKB, whitespace has no special use. An author may leave some whitespace unfilled to communicate meaning such as a division. In contrast, Expeditee uses whitespace for navigation and communicating meaning. Left clicking in whitespace performs a *Back Operation*, navigating the user to the Frame they were on prior to the current. The benefit of this is that navigating between Frames can be done rapidly, with whitespace being used to navigate backwards and links tending to have no disturbance near their hitbox. More details concerning navigation in Expeditee are provided in Chapter 6.

	Data Structure				Nesting Type	Spatial Interface		Fundamental Element	Other First Class Citizens	Multimedia	Reflective First Class Citizens	Level of Occlusion
	Sequence	Array	Set	Allows Nesting		Variable	Fixed					
VIKI			✓	✓	Deep and Strict	✓		Collection	Object			Likely, at least some
VKB			✓	✓	Deep and Strict	✓		Collection	Object	✓		Likely, at least some
Expeditee			✓		Deep and Transient		✓	Text Item	Images, Dot, Line etc.	✓	✓	Very likely, most

Table 4.3: A summary of the three applications reviewed in this section. For each: The type of data structure that can be thought of as being used to store citizens and if nesting is used. The type of Spatial Interface is being used. Details about the types of citizens present.

4.4 Summary

This chapter has thoroughly examined several aspects of the design of authoring environments that have significant effects on their capabilities. We discussed the concept of spatially positioning content and used the ideas of absolute versus relative space to explain how this is achieved. Not satisfied with simply providing a definition, we then set out to use examples to explain how we can identify Spatial Hypermedia applications.

Contrast was drawn on the lines of relative or absolute positioning of content, how (and if) nesting is supported and the issue of occluded content—specifically when it becomes an issue when using absolute positioning. This allowed us to demonstrate how the layout and meaning of content are tied together. It was shown that these aspects influence an application’s suitability for different types of content.

A focus on Spatial Hypermedia authoring considerations, in combination with a review of three modern general purpose Spatial Hypermedia authoring applications, has shown that significant differences arise due to the design goals and decisions made when developing Spatial Hypermedia.

The discussion in Sections 4.1 and 4.2 was undertaken to bring together and explain the relevance of Chapter 2 and 3 to this research. In Chapter 2 we looked at Spatial Memory and how it can be applied to software development. It was in this chapter that we encountered initial evidence showing that programmers are able and willing to use their Spatial Memory whilst programming. We were also able to draw on literature reviewed in this chapter to distinguish between two different types of Spatial Hypermedia—Fixed and Variable Sized. Chapter 3 then examined traditional IDEs, resulting in us noticing a distinct lack of malleability. This observation motivated the development of a formal model of authoring built around the concepts Fundamental Element, System Representation and First Class Citizen. These concepts allow us to identify the level of mutability present in an application and how wide-spread that mutability is over the application’s citizenship. The goal of this work was to help establish a group of citizens designed with consistently high mutability. These citizens can then be used to create a Spatial Hypermedia IDE that does not have the lack of malleability present in traditional IDEs.

Chapter 5

Authoring in Spatial Hypermedia IDE

Environments

This chapter outlines and justifies the development direction that was taken in the development of SpIDER. This is achieved by reviewing four examples in Section 05.1 of IDE authoring from the literature that use a significant amount of Spatial Hypermedia in their design: Code Thumbnails, Code Canvas, Code Bubbles and Debugger Canvas. As will be explained in their appropriate sections, each of these applications approaches the task of using Spatial Hypermedia to utilise Spatial Memory differently. The formal descriptive model described in Section 4.2 is applied to each application. The format of the review mimics the format seen in Section 4.3. In addition we relate each piece of software assessed to the work presented in Chapters 2 and 3.

The aspects of Spatial Memory applicable to computer interfaces that were discussed in detail in Chapter 2 are:

- Spatial Memory can be used for navigation and object location.
- The use of Spatial Memory can be promoted through the positioning of components with respect to the edges of application windows.
- Spatial Memory is better utilised with a Fixed Size spatial interface rather than a Variable Sized spatial interface.
- The gap between Fixed Size and Variable spatial interfaces can be lessened by using overviews and landmarks.

- An overview is a miniaturized and less detailed version of the whole information space.
- Landmarks are prominent components of the information space that people can use as focal points from which to map other elements.
- Landmarks are generally useful—in both Fixed Sized and Variable Sized interfaces.
- Long-term Spatial Memory is arranged hierarchically. Similarly, interfaces may use a hierarchical structure to reveal or hide content.
- It is a challenge to achieve spatial stability in content that is actively being authored.

Chapter 3 discussed common functionality found in IDEs and users' interactions with them.

The purpose of this discussion was three-fold:

1. To identify core functionality that a Spatial IDE would require.
2. To analyse limitations that traditional IDEs exhibit in their implementation of functionality and the ways that Spatial Hypermedia can be used to overcome these limitations.
3. To examine how programmers currently use space in traditional IDEs.

The rigidity of IDE interfaces and functionality was explored as a weakness of traditional IDEs.

Heading towards the completion of this chapter, in Section 5.2, we bring together several discussion points put forward earlier in the chapter (and thesis) so that in Section 5.3 we can explain why we believe the Spatial Hypermedia system Expeditee is the best platform of those reviewed for building a Spatial Hypermedia-based IDE.

5.1 Spatial Hypermedia in IDEs

We now review four research projects that resulted in the construction of Spatial Hypermedia functionality for programming. Firstly, we review Code Thumbnails by DeLine et al. [12], an early investigation performed at Microsoft Research into how programmers are able to use Spatial Memory to navigate around a code base. Code Canvas was subsequently produced by DeLine and associate Rowan [51], and is reviewed second. In this work DeLine and Rowan create a programming environment that allowed Spatial Memory to be utilised for both producing code and navigating around an entire code base. Code Bubbles is the third project to be reviewed [52, 53]. Created by Bragdon et al., Code

Bubbles, like Code Canvas, is a Spatial Hypermedia-based IDE. Unlike Code Canvas, Code Bubbles does not attempt to provide users with the entire code base at once, instead opting to allow the user to work with fragments of the software project.

Extending on our documentation of Code Thumbnails in Section 2.4, we now evaluate it in a fashion suitable for comparison with other Spatial Hypermedia IDEs. For the remaining three applications, we perform the same analysis undertaken in Section 4.3:

- We examine their citizenship, and identify First Class Citizens and its System Representation.
- We discuss the implications that the application's design has for Spatial Memory.
- We review how authoring is performed in the application.
- And finally, the evaluation that researchers undertook on their application.

5.1.1 Code Thumbnails

Code Thumbnails is an extension to Microsoft Visual Studio, created by DeLine et al. at Microsoft Research [12]. We first introduced Code Thumbnails in Section 2.4 where we discussed the study that DeLine et al. undertook to evaluate a programmer's willingness and ability to use Spatial Memory for programming. We now extend that discussion by examining the design and functionality that Code Thumbnails provides as well as the effect the design has on Spatial Memory.

Unlike the other applications reviewed, Code Thumbnails is not directly concerned with editing. Instead it aims to accelerate a user's navigation, thereby benefiting the programming experience. In order to achieve the goal of allowing programmers to leverage Spatial Memory for navigating between code snippets, two interfaces are added: The Code Thumbnails Scrollbar and Code Thumbnails Desktop. In terms of Spatial Memory considerations, they are both Fixed Sized spatial interfaces that are spatially stable. The former resembles an overview as defined in Section 2.2. In other words, it provides a miniaturized and less detailed view of the complete code file. The latter resembles the Space-Filling Thumbnails developed by Cockburn et al. [17].

Code Thumbnails Scrollbar. A screenshot of the Code Thumbnails Scrollbar, taken from [12], can be seen in Figure 5.1. Examining this screenshot, we see that the traditional scrollbar present in Visual Studio, for moving up and down through a code file, has been

augmented. This is the Code Thumbnails Scrollbar. To the left of the augmented scrollbar is the normal text editor for writing C# code. To the right, the complete file is represented in a single view. In order to fit all of the content, the text has been scaled down. This provides text that is indicative of how long the lines of code are and how they are indented (the shape), but is not intended to be read. However, a programmer is able to click on a portion of code in the augmented scrollbar and cause the code editor area on the left to centre on that content. Unable to read the text in the scrollbar, if a programmer wishes to use it for navigation, they must instinctively start to use aspects of Spatial Memory to navigate the file. Namely, the shape and spatial position—with reference to the edge of the file or relative position of other code they can already place (a landmark)—of the code snippet that contains the code they wish to navigate to. In turn, this means that a successful navigation shows that they have utilised their Spatial Memory.

We review the Code Thumbnails Scrollbar because it is a useful example of a tool that allows programmers to use their Spatial Memory to navigate around a source file. However, unlike all the other applications we review in Chapter 5, the Code Thumbnails Scrollbar is not designed for authoring, instead it provides an overview of the authoring area. All user interactions with the Code Thumbnails Scrollbar result in a navigation (to a new location in the main text editor area) and never a manipulation of content. As we distinguish between relatively or absolutely positioned content and identify the type of data structure used to store citizens so that we can understand how authoring effects the application, it does not make sense to discuss these aspects.

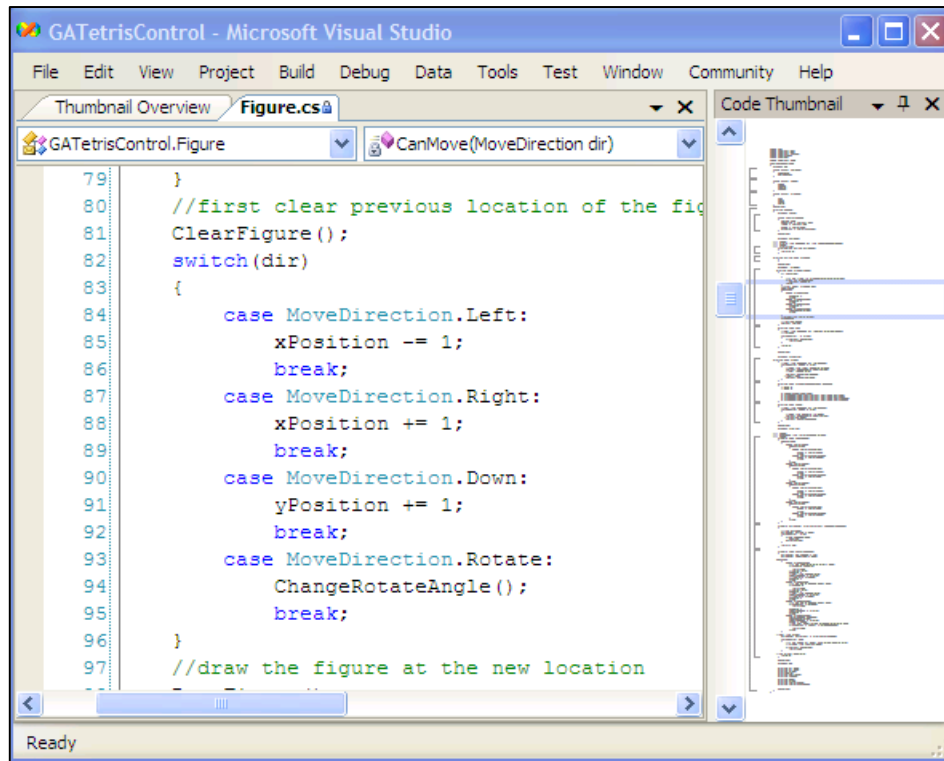


Figure 5.1: Code Thumbnails Scrollbar, taken from [12].

Code Thumbnails Desktop. A screenshot of the Code Thumbnails Desktop, taken from [12], can be seen in Figure 5.2. Examining this screenshot, we see a canvas filled with spatially arranged thumbnails. Each thumbnail is a further scaled down copy of a Code Thumbnails Scrollbar. This means that, in contrast to the Code Thumbnails Scrollbar, which provides navigation within a single file, the Code Thumbnails Desktop provides navigation over all the code files of a complete software project. Programmers are able to manually reposition each thumbnail, but the extension is otherwise completely spatially stable.

Border thickness and shaded blue areas of differing intensity are used to communicate information such as which file was last opened and where the viewport on each file currently resides. As with the Code Thumbnails Scrollbar, programmers are able to click on a thumbnail and cause Visual Studio to navigate to the selected file.

Readable labels are provided on the thumbnails in the Code Thumbnails Desktop which could be used to accurately select the navigation target file. Whilst present, these labels may be not needed. Evaluation done by DeLine et al. (and reviewed in Section 2.4) shows that programmers who have had opportunity to practice using Code Thumbnails Desktop

with a specific software project are still able to use the tool once the entirety of all thumbnails were made invisible (including the titles), providing evidence that Spatial Memory is being utilised.

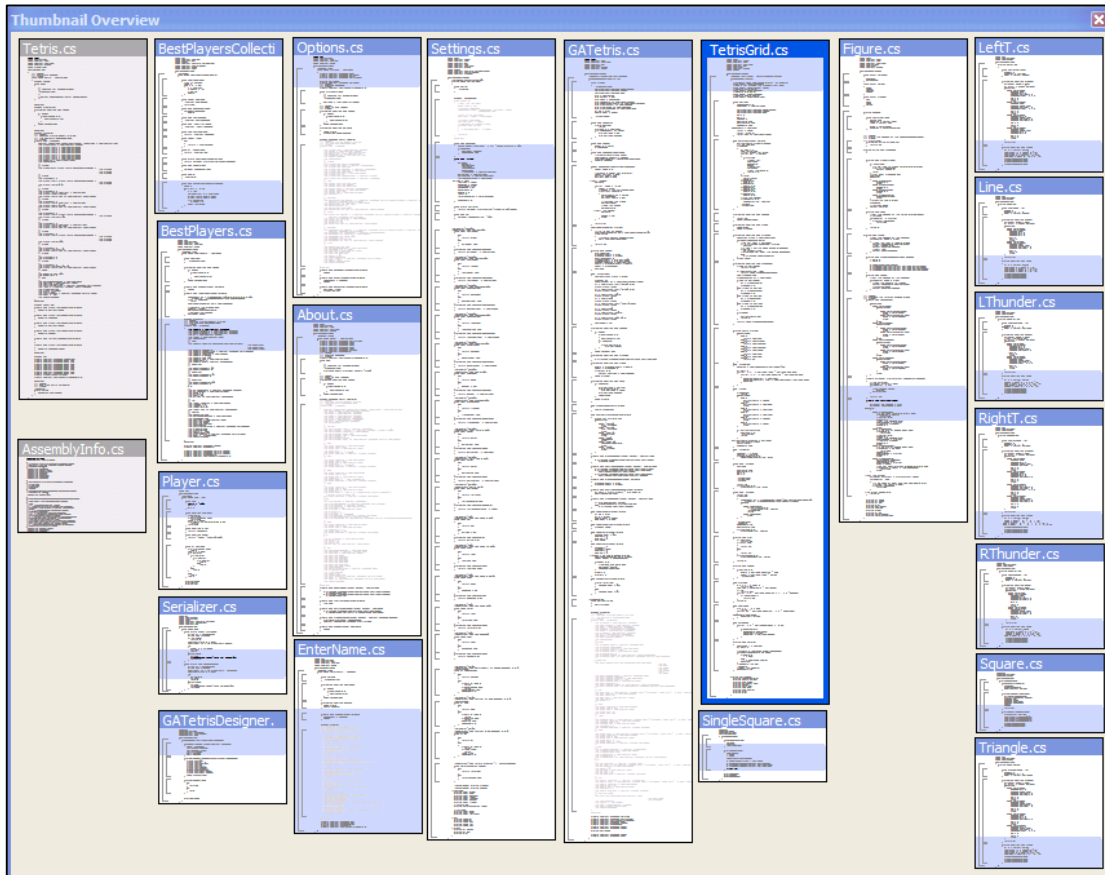


Figure 5.2: Code Thumbnails Desktop, taken from [12].

Parallels. Both CodeMap’s by Kuhn et al. [54] and Code Thumbnails seek to utilise a programmer’s Spatial Memory to assist with navigation. Where Code Thumbnails uses the shape of source code to achieve this, individual CodeMap’s instead generate a spatial visualisation of the code. Results seem to suggest, that for the purpose of accelerating navigation through Spatial Memory, the approach taken by DeLine et al. achieves better results.

5.1.2 Code Canvas

Working with Rowan, DeLine extended research into the use of Spatial Memory for programming with the development of Code Canvas—a significant restructuring of Microsoft Visual Studio [51]. Unlike Code Thumbnails, Code Canvas is concerned with

authoring code. This, combined with its leveraging of Spatial Memory, makes it a Spatial Hypermedia Integrated Development Environment. As such, the developer's approach, implementation and research questions are of interest to this thesis. In their paper, they discuss the overall success of IDEs in general but highlight a level of stagnation in IDE interface design. They make the case for looking to improve on the existing design by comparing the prevalence of high-end personal computers today with the computer systems that were commonplace when IDE interface design began to stagnate. In an effort to address this lack of innovation and leverage Spatial Memory they designed and built a version of Microsoft Visual Studio that features an interface with similarities to that of VIKI/VKB.

Figure 5.3, taken from [51], shows three screenshots of Code Canvas. Each screenshot is at a further level of zoom beyond the former, with the top screenshot displaying the complete software project and the bottom zoomed in enough to allow text to be read comfortably. This form of zooming is dubbed Semantic Zoom by DeLine et al. because at each stage of zoom attempts to show the user an aspect of programming with semantic meaning. Yellow highlighted content shows the results of a previous search and the red arrows show a stack trace of the software project.

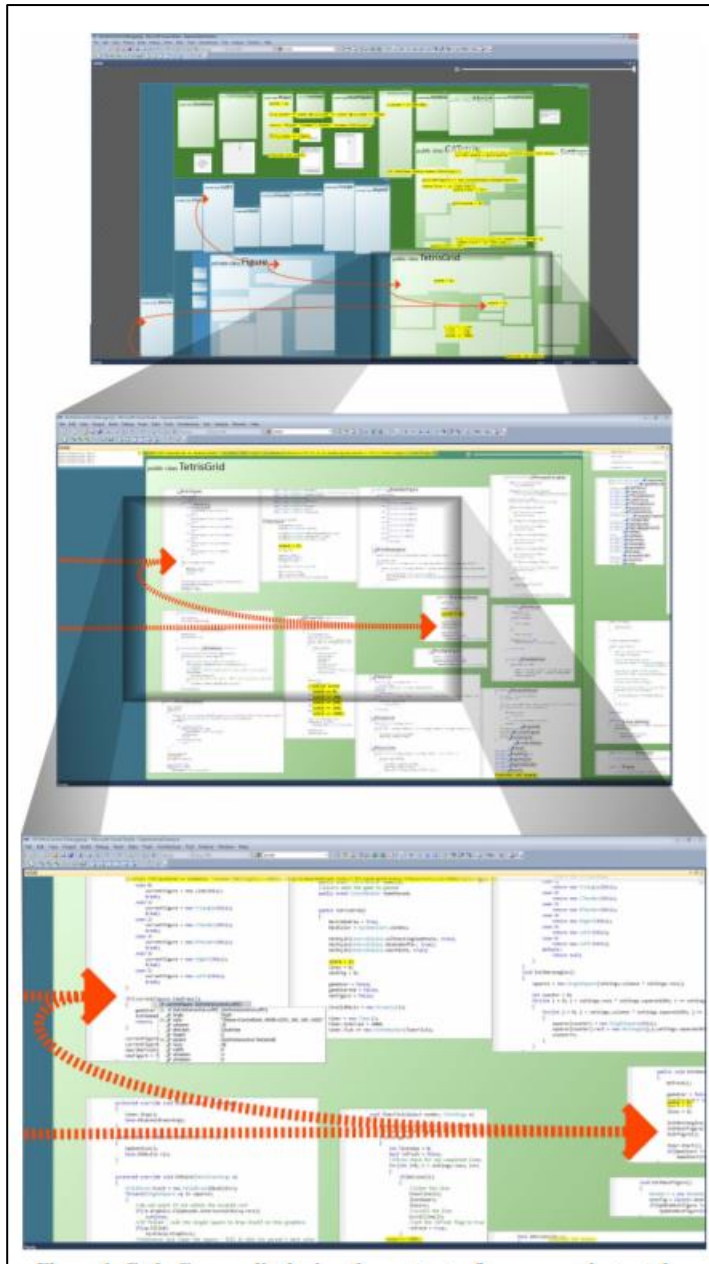


Figure 5.3: Three levels of zoom in Code Canvas. Project view, class view and editing view. Taken from [51].

Citizenship. By examining [35] we are able to map Code Canvas into our on-going discussion and analysis of citizenship in authoring environments. The spatially positionable rectangular container for code fulfils the role of the Fundamental Element. Unnamed in their work, we refer to this citizen as a Semantic Container, as it contains a semantically valid fragment of code as determined by the language. Figure 5.4 explains this citizen diagrammatically. As far as functionality that the author can directly affect, it is identical to a traditional text editor. Additional hypermedia functionality is added to the editor based

on the content that is entered, such as hyperlinks from references to functions to their implementations. Other functionality has visual aspects that may be thought of as citizens, such as the red arrow signifying a stack trace or the highlighting that is the result of a search initiated by the user.

Semantic Container

Content:	Seq[Character]
Operations:	Insert, Remove or Replace Characters

Figure 5.4: The primary citizen of Code Canvas.

Figure 5.5 shows the System Representation of Code Canvas. Supported operations are the ability to add, remove or replace citizens, spatially position Semantic Containers and perform Semantic Zoom. The order of citizens is not important, as such it makes sense to think of them as being stored in a Set.

Code Canvas

Content:	Set[Semantic Container ..]
Operations:	Add, Remove, Replace Citizens. Can Reposition Semantic Containers Can perform Semantic Zoom

Figure 5.5: The System Representation of Code Canvas.

Spatial Memory Considerations. Code Canvas utilises an infinite canvas with pan and zoom controls. This results in the classification of a Variable Sized Spatial Interface. As another example of a Variable Sized Spatial Interface, there are multiple similarities between VIKI/VKB and Code Canvas.

- Content is represented in spatially positioned containers. VIKI/VKB use Objects where Code Canvas uses Semantic Containers.
- A restricted form of zooming is utilised. In VIKI/VKB we referred to this as hierarchical zoom because it allows the user to view a specific part of the tree hierarchy that was formed. The developers of Code Canvas refer to their restricted form of zooming as ‘semantic zoom’, because it uses the semantics of the programming language to determine what each stage of zoom does—for example,

one level of zoom may be designed for writing code and another for viewing the class structure.

- No overview—as defined in Section 2.2—is present. Zooming out as far as possible may or may not provide a complete overview of the information space, depending on the quantity of information present.
- Prominent features, such as class or public variable names, are emphasised and may be used as landmarks. This is similar to Collection titles in VIKI/VKB. Other than this, as with VIKI/VKB, the shape of content must be used to provide landmarks.

Two differences between VIKI/VKB and Code Canvas that are significant for utilising Spatial Memory are:

1. The ability to vary font properties in VIKI/VKB but not in Code Canvas. Code Canvas retains much of the editing functionality that traditional Microsoft Visual Studio provides. This means that a programmer using Code Canvas is restricted in the ways they can emphasise specific pieces of code in a way that users of VIKI/VKB are not.
2. Whilst both VIKI/VKB and Code Canvas feature infinite scrollable canvases, they differ in how they deal with an increasing amount of content. VIKI/VKB use scrollbars on both the entire canvas and on their content containers (Objects). In contrast, Code Canvas only has a scrollbar on the entire canvas. Instead of producing a scrollbar once a container is full of content, the container grows. This is a trade-off. On the one hand, less information is forcibly occluded when interacting with a specific container because the container has grown to accommodate the content. On the other, this same container growth may make it desirable to reposition other containers which may be damaging to Spatial Memory.

Authoring. The authoring in Code Canvas differs from the traditional form of Microsoft Visual Studio by allowing programmers to spatially position semantically determined chunks of code. However, when editing a specific Semantic Container of code, the programmer must use the traditional text editor augmented with hypermedia functionality that is used in the traditional form of Microsoft Visual Studio. The spatial layout of code is also used to

provide programmers with alternative ways of visualising information, such as the red arrows that present a stack trace.

Evaluation. DeLine and Rowan describe the implementation details of Code Canvas by stating research questions. One of which is: “To what extent should Code Canvas be a collaborative space versus a personal space?” [35]. Progress towards answering this question is made in a follow-up publication [55]. This concept of a Code Map is presented. It is a diagrammatic way of presenting all of the relations between information contained in a software project, including code, documentation and planning. A field study is undertaken where a Code Map is initially produced with pen and paper and later transferred to Code Canvas.

A Code Map for a team of programmers is designed and produced. Interviews, observation while the programmers work and analysis of diagrams that the programmers produce are all considered. Several iterations of the Code Map are made and each is provided to the team of programmers by attaching it to the wall of a shared space. Each iteration receives varied opinion from team member to team member. Ultimately, it is reasoned that the lack of interactability and difficulty of modification by the programmers limits the usefulness of the paper Code Map. At this point, a new version of the Code Map is created in Code Canvas, with each member of the team being able to access and modify it. Through continued questioning, further evidence is found that programmers are able to utilise their Spatial Memory and find the Code Map to be a useful addition to their development cycle.

5.1.3 Code Bubbles

Code Bubbles is a Spatial Hypermedia IDE for the Java Programming language developed by Bragdon et al. [52, 53] as an extension of Eclipse. Like Code Canvas, Code Bubbles features an infinite scrollable canvas and uses spatially positionable containers to display code. A container is referred to as a ‘Bubble’. A Bubble contains a semantically meaningful fragment of text, such as a function or piece of documentation. Related Bubbles are arranged into groups referred to as Bubble Groups. For example, a programmer may opt to place all functions that add widgets to a window in a single Bubble Group. In turn, multiple Bubble Groups form a Working Set which is represented in an overview along the top of the screen. This overview is referred to as the Panning Bar. A Working Set is intended to

contain all relevant—and no additional—information to address a specific issue, such as fixing a bug.

Figure 5.6 shows a screenshot of a Code Bubbles workspace taken from a video published by Bragdon.³ The primary difference between Code Canvas and Code Bubbles is illustrated in this screenshot. Where Code Canvas spatially arranges a complete software project, Code Bubbles limits what is displayed to the content you are currently working with. There are four Bubble Groups displayed in the screenshot, each with a different background colour.

- The blue and green Bubble Groups both contain code. The former contains four Bubbles and has been marked with an icon to indicate that it contains a bug. The latter contains three Bubbles.
- The pink and yellow Bubble Groups contain project-related information that is not code. In the case of the yellow working set it is documentation.

Unlike the Bubble Groups that contain code, which are arranged vertically, the yellow working set is arranged horizontally. This is an example of using spatial layout to communicate meaning. The Bubbles are ordered within their Bubble Group—each subsequent Bubble is a result of following a path from the previous.

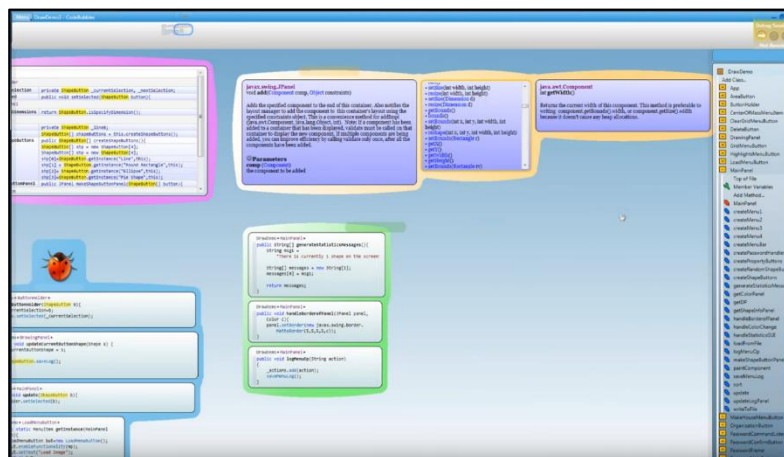


Figure 5.6: A Code Bubbles workspace featuring a single Working Set that contains four Bubble Groups with between one and four Bubbles each.

³ <https://youtu.be/PsPX0nEIJO4> Accessed April 2017

Citizenship. Through examining [52, 53], the previously mentioned video and two Code Bubbles websites,^{4,5} we are able to map Code Canvas into our on-going discussion and analysis of citizenship in authoring environments. The Fundamental Element of Code Bubbles is the Bubble. Acting as a ‘mini-world’ traditional text editor, a Bubble can have characters inserted, removed or replaced. Just as the semantic container from Code Canvas utilises the traditional Visual Studio editor for dealing with the content entered, the Bubble uses Eclipse’s traditional text editor. This has the positive effect of automatically adding hyperlinks between references and implementations, but the negative effect of restricting users from modifying font style or inserting diagrams alongside code. Figure 5.7 shows the diagrammatic representation of the Bubble Item.

Bubble

Content:	Seq[Character]
Operations:	Insert, Remove or Replace Characters

Figure 5.7: The Fundamental Element of Code Bubbles.

Two other Items are worth discussing: Bubble Groups and the Working Set. A Bubble Group is a collection of Bubbles. Bubbles within a Bubble Group are not ordered. Bubbles within a Bubble Group can be spatially positioned as long as they remain in contact with another Bubble in the Bubble Group—if contact is lost, that Bubble forms a new Bubble Group. Entire Bubble Groups can be spatially positioned. Figure 5.8 shows the diagrammatic representation of the Bubble Group Item.

Bubble Group

Content:	Set[Bubbles]
Operations:	Insert or Remove Bubbles

Figure 5.8: The Code Bubbles Bubble Group.

The Working Set is a collection of Bubble Groups that together are used to solve a specific task. All Working Sets appear in the Panning Bar at the top of the application. The Panning

⁴ <http://cs.brown.edu/~spr/codebubbles/> Accessed April 2017

⁵ http://www.andrewbragdon.com/codebubbles_site.asp Accessed April 2017

Bar can be used to save, load or removing existing Working Sets from the application.

Figure 5.9 shows the diagrammatic representation of the Working Set Item.

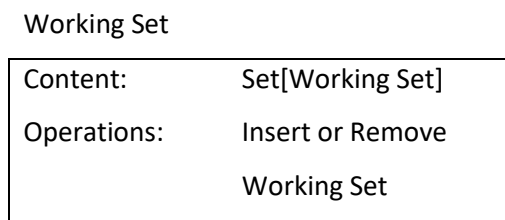


Figure 5.9: The Code Bubbles Working Set.

The System Representation of Code Bubbles must be able to store and manipulate multiple Working Sets and control the viewport based on the level of zoom the user sets as well as the active portion of the application (as specified by the Panning Bar). Figure 5.10 shows the diagrammatic form of the System Representation for Code Bubbles. Working Sets, Bubble Groups and Bubbles represent a thin hierarchy of Items that no other application reviewed has had. This thin hierarchy places a significant amount of functionality on these Items rather than on the System Representation—for example, the containment of Bubbles in a Bubble Group.

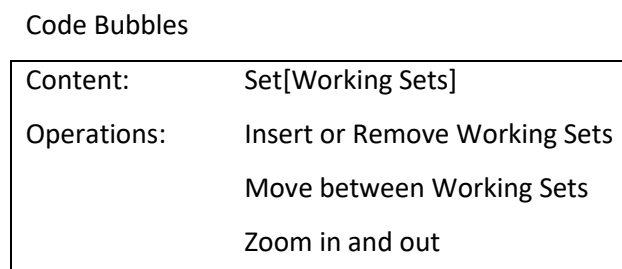


Figure 5.10: The Code Bubbles System Representation.

Spatial Memory Considerations. As another application with an infinite scrollable canvas, the Spatial Memory considerations for Code Bubbles are similar to those of Code Canvas and VIKI/VKB. It is classified as a Variable Sized Spatial Interface due the ability to pan the viewport. Unlike Code Canvas and VIKI/VKB, that use restrictive forms of zooming, semantic and hierarchical zoom respectively, Code Bubbles uses the traditional form of zooming—where the size of content is scaled by a percent specified by the programmer.

Code Bubbles' working sets provide programmers with the ability to group related Bubbles. This is similar to the relation between Objects and Collections in VIKI/VKB but does not have the same hierarchical nature—that is, whilst a Collection can be placed inside another

Collection in VIKI/VKB, a Working Set cannot be placed inside another Working Set in Code Bubbles.

Authoring. As with the Semantic Containers of Code Canvas, a Bubble behaves as a traditional authoring environment—retaining the features and limitations of the Eclipse text editor. It is the spatial positioning of Bubbles and expressiveness of working sets that makes Code Bubbles unique.

Bubbles can be created either by following hyperlinks from existing Bubbles or through the Package Explorer docked to the right of the application (Figure 5.6). New Bubbles join an existing logical Bubble Group (if there is one). For example, if a Bubble is created from clicking a reference to a function, it will join the Bubble Group that the reference belonged to. However, Bubbles can be individually detached from their current Bubble Group and attached to another or left as a new Bubble Group.

Reconsider Figure 5.6. Together, all four Bubble Groups are considered part of a Working Set. Along the top of the window is an area that shows all currently active content. This Working Set is spatially positioned in this overview. A blue rounded rectangle displays where, in the entire information space, the current viewport is. This provides programmers with a technique for spatially separating different tasks. For example, should the programmer be working on fixing the bug in the blue working set, and a colleague asks for help on another part of the code base, they are able to navigate to a spatially separate part of the information space, and bring up the appropriate code fragments there.

Evaluation. In [52] Bragdon et al. discuss qualitative and quantitative evaluation of Code Bubbles that they undertook. For their qualitative evaluation, they used 14 professional programmers (13 male, 1 female with a mean age of 31.85 years). The participants were given a copy of Code Bubbles, and asked to ‘think out loud’ while they operated it. The feedback received was positive. It was noted that participants seemed to appreciate the concept of Working Sets, Bubble Groups and Bubbles and how they may be useful in: staying on task, organising their thoughts and working towards solving specific issues. Whilst participants did not miss the file structure organisation that traditional IDEs provide, there was discussion of how Code Bubbles would fare when dealing with large changes, such as rewriting complete classes. It was suggested that a separate interface—referred to as a *Class Bubble*—would be useful for this.

In their quantitative evaluation Bragdon et al. wished to test their hypothesis that: “Code Bubbles users will be able to understand the code more quickly, take advantage of multiple simultaneous bubbles, and should use significantly fewer navigations/minute on average, and fewer repeated navigations/minute on average. [52]”. Participants for this study were drawn from a pool of graduate and undergraduate Computer Science students at Brown University. There were 20 participants in total (19 male, 1 female with a mean age of 21.95 years). Each participant had experience using Eclipse—the IDE that Code Bubbles was built on.

In order to measure participant’s code understanding, they were given the task of fixing a bug while their actions were recorded. Each participant was randomly assigned to one of two groups: a control group that used Eclipse and an experimental group using Code Bubbles. Participants completed three tasks, the first was a ‘training’ task that they were given 15 minutes to complete and was not reported on. The second (T1) and third (T2) tasks were more complicated bug fixes (more lines of code and members to consider as compared to the training task) and they were given 45 minutes to complete.

Analysis of the results found that participants using Code Bubbles were both more successful in completing their tasks (meaning the bug was fixed)—with 6/4 completions in the control group and 10/7 in the experimental group, T1/T2 respectively—and faster doing so in the first non-training task—with T1 being performed almost twice as fast and T2 being completed slightly faster in the experimental group, compared to the control group. Whilst Code Bubble performed slightly better in T2, it was not a statistically significant result. Adding together the time taken and success rate for both tasks produced a result showing that participants using Code Bubbles performed statistically better. It was also found that users of Code Bubbles spent significantly less time navigating—with almost half as many navigation actions, resulting in significantly less time navigating.

5.1.4 Debugger Canvas

Following Code Bubbles, Bragdon collaborated with DeLine et al. at Microsoft Research on Debugger Canvas [56]. As an extension to Microsoft Visual Studio, Debugger Canvas is intended as a production version of the Code Bubbles paradigm. Debugger Canvas adapts the concept of a Bubble to filter a programmer’s debugging session to show only the relevant information. Figure 5.11 shows a screenshot, taken from [56], of Debugger Canvas

running. In the figure, six Bubbles are currently open. These six Bubbles are divided into four groups. Each group represents a thread of the running application. Each group is distinguished by a different border colour. The topmost group has a teal border. Moving in order downwards, the other border colours are yellow, blue and purple. The Bubble containing the currently active code is the right-most Bubble in the top group. This is signified by a highlighted title to the bubble as well as a secondary thin border inside the existing border.

The topmost group contains three Bubbles. A series of arrows order the Bubbles within this group, showing the stack trace that resulted in those specific Bubbles being opened. The three remaining groups contain a single Bubble each.

In developing Debugger Canvas, DeLine et al. primarily wanted to gather data on programmers' experiences with the Code Bubble paradigm. Debugging was chosen (over an entire IDE overhaul) for a combination of reasons. Debugging is often a cognitively difficult task that tends to last for several minutes, and so provides a good opportunity for data gathering. It is the type of task that the Code Bubbles paradigm should perform well with. Furthermore, the researcher's report that they were hesitant to replace the traditional style of code editing as it was thought that programmers may be unwilling to make such a significant change all at once.

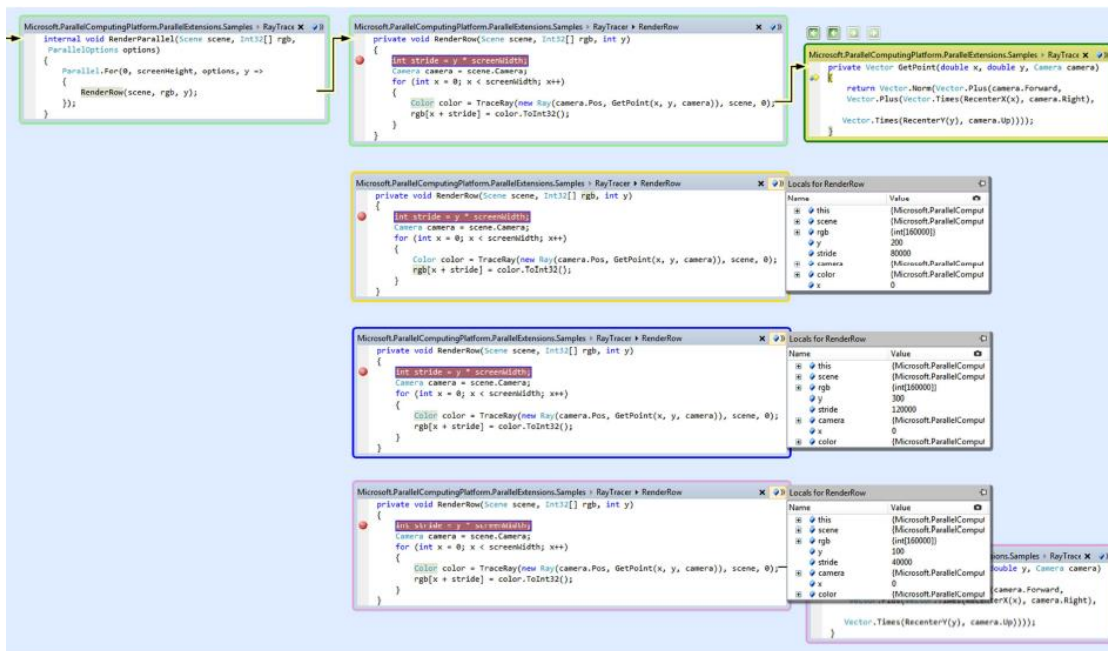


Figure 5.11: A screenshot of Debugger Canvas, showing six bubbles and associated debugging information.

Citizenship. As Debugger Canvas is a variation on the Code Bubble paradigm, its citizenship is similar to Code Bubbles. The Fundamental Element remains the Bubble and continues to act as a ‘mini-world’ traditional text editor. Figure 5.12 shows the diagrammatic representation of the Debugger Canvas Bubble. Whilst Bubbles maintain the ability to author content, they are also augmented with traditional Visual Studio tools to assist with debugging, such as a display showing the current value of variables.

Bubble

Content:	Seq[Character]
Operations:	Insert, Remove or Replace Characters
	Debugging Specific Tools

Figure 5.12: The Debugger Canvas Fundamental Element.

Debugger Canvas arranges Bubbles based on their thread of execution. This is comparable with the Bubble Group citizen in Code Bubbles. We will refer to this citizen as a Thread Group. Figure 5.13 shows the diagrammatic representation of the Thread Group citizen. Unlike in Code Bubbles where spatially positioning Bubbles close to each other caused Bubble Groups to form, Debugger Canvases Thread Groups are formed based on the semantics of the programming language—specifically, the thread of execution that a Bubble belongs to determines which group it belongs to.

Thread Group

Content:	Set[Bubbles]
Operations:	Insert or Remove Bubbles

Figure 5.13: The Debugger Canvas Thread Group.

Unlike the Bubble Group citizen from Code Bubbles, that has a direct parallel in the Thread Group citizen, there is no equivalent to the Code Bubbles Working Set. Whilst programmers using Code Bubbles are able to specify which Working Set new Bubbles populate, in Debugger Canvas, new Bubbles always join the newest canvas that the programmer has created. If a user does not create a new canvas for a new debugging session, Debugger Canvas does not visually separate the new session from the previous. Furthermore, interaction with each canvas occurs in isolation from the others. For example, each canvas keeps track of its own level of zoom and viewport position. For these reasons, only one conceptual structure is needed to represent both the theoretical equivalent of the Code Bubbles Working Set and the Debugger Canvas System Representation, as can be seen in Figure 5.14.

Canvas

Content:	Set[Thread Group]
Operations:	Insert or Remove Thread Groups Zoom and Pan

Figure 5.14: The Debugger Canvas System Representation.

Spatial Memory Considerations. The Spatial Memory considerations for Debugger Canvas are—unsurprisingly—similar to those of Code Bubbles. Both use a Variable Sized spatial interface supported by traditional pan and zoom functionality.

The primary visual difference between Debugger Canvas and Code Bubbles is in their approach to displaying groups of Bubbles. Whilst Code Bubbles uses the close positioning of Bubbles to form Bubble Groups, Debugger Canvas instead utilises Thread Groups, based on the thread the Bubble belongs to. The freedom to spatially position Bubbles distinct from each other and still have their relationships recorded gives programmers flexibility. This should result in more opportunities to visually communicate additional information for human interpretation. However, this divergence from the Code Bubbles paradigm does

mean that programmers have lost the ability to determine which Bubbles belong to which group.

The impact of this change is interesting to consider. Another way to summarise this change is to say that programmers have gained more avenues for using space to communicate information directly to humans but have lost the ability to specify groupings to the computer, therefore decreasing the potential avenues for algorithmic assistance. This appears to be a win for spatial communication, but—with the details reported in [56]—it is not clear if this is a net win overall.

Generally speaking, it seems logical that when comparing two applications, the one with the wider scope should provide more flexibility to the user. This should allow the user to deal with issues that the developer of the application did not foresee when designing the application. In this specific case, we have Code Bubbles, designed to assist with programming as a whole, and Debugger Canvas, designed to assist with only debugging. It follows that we should expect Code Bubbles to be more flexible, which in this case would mean more freedom to spatially position content.

However, through the process of adapting the Code Bubbles paradigm for debugging, it seems the developers have increased the potential for spatial communication. This observation is consistent with a comment made in [56] where DeLine et al. discuss having to re-evaluate the design decisions made when producing Code Bubbles.

Authoring. The authoring environment of Debugger Canvas does not differ from that of Code Bubbles, notwithstanding those already highlighted. Namely:

- The formation of groups of Bubbles being based on threads rather than proximity,
- The merging of the Debugging Canvas System Representation and theoretical Working Set and
- The addition of Debugging Specific Tools—such as the state of variables.

Evaluation. DeLine et al. performed multiple evaluations during and after development aimed at releasing Debugger Canvas as a production level tool. They accomplished these evaluations by gathering both qualitative and quantitative user feedback and measuring the performance (execution speed) of Visual Studio with and without Debugger Canvas installed.

When discussing their evaluation of the performance of Debugger Canvas in [56], DeLine et al. delineate the development chronologically over two releases. For our purposes, it is sufficient to discuss the final results that they obtained. It was found that panning a canvas remains responsive with upwards of 100 Bubbles on the canvas. This number of code fragments is more than enough to necessitate panning and is likely to be more code fragments than will typically be used in a single debug session. DeLine et al. put this result down to the use of existing libraries that they were using.

Other performance measurements reported showed a minor cost in execution speed when using Debugger Canvas. After having hit a breakpoint, stepping through code on a canvas was shown to take 100 ms longer per step. However, under normal circumstances, this speed decrease is regardless of the number of Bubbles present, therefore making this a negligible result. Exceptional circumstances can occur however. When multiple duplicate Bubbles are present, the time per step would increase linearly with the number of duplicate Bubbles. Users are provided with an option to reuse Bubbles when the content is the same in order to avoid this issue.

Another speed penalty was present when starting a new debugging session. To measure this difference, the time between the 'Start Debugging' button being pressed and the UI being ready for use was measured:

- i. Without Debugging Canvas installed. 1.5 seconds measured.
- ii. With Debugging Canvas installed but not being used. 2.2 seconds measured.
- iii. With Debugging Canvas being used. 3 seconds measured.

With the baseline established in (i), it was found that using Debugging Canvas (iii) doubled the load time.

Prior to the public release of Debugger Canvas, 10 participants were given the goal of completing three tasks in Debugger Canvas in an hour. This allowed DeLine et al. to gather early feedback and make appropriate changes. It was from this trial that the decision to provide users with the ability to create a new canvas was made. Prior to this, Visual Studio automatically made a new canvas for every debug session, and it was discovered that sometimes users wanted to continue using an existing canvas.

Following the public release, download data was recorded. During the first week, six thousand instances of Debugger Canvas were installed. Whilst this dropped significantly over the next couple of weeks, the number of new downloads did stabilise quickly. DeLine et al. take this as a sign that word of mouth is causing adoption as no advertising attempt was made. Over the first 40 weeks: weeks 1 to 3 saw 55% of the total downloads, leaving 45% of downloads to occur between week 3 and 40. Data was also collected through the Microsoft Cursor Experience Improvement Program. The relative use of different commands was recorded. One interesting comparison that can be made from this data is that for every new Bubble that was stepped into, there were 0.11 new canvases created. Or in other words, the average canvas used contained slightly less than 10 Bubbles.

5.1.5 Spatial Hypermedia in IDEs Discussion

Table 5.1 shows a summary of the applications covered in this section. We group the two components of Code Thumbnails together. We also group Code Bubbles and Debugger Canvas under the heading of Code Bubbles Paradigm. In addition to many of the categories we have encountered in Sections 4.2.3 and 4.3.4, a new column is present: ‘Level of Detail’, which differentiates the applications reviewed based on the quantity of information they present to users.

- As the Code Thumbnails Scrollbar provides an overview of a single code file, we state its level of detail as: code file. The Code Thumbnails Desktop incorporates all of the code in a software project by providing thumbnails for each code file, we therefore state its level of detail as: code base
- Code Canvas goes a step further by expanding the level of detail to: entire software project. Not only does it allow the author to interact with the entire code base (through different levels of zoom) but also incorporates the ability to have non-code content included alongside code files.
- Code Bubbles and Debugger Canvas both have the level of detail: working set. Instead of attempting to provide programmers with the entire software project, these applications instead allow users to limit the amount of content displayed to them so that only pertinent information can be seen.

Of all the applications reviewed in Section 0, only the Code Thumbnails paradigm utilises nesting. We omit this detail from Table 5.1 in favour of stating the details here—simplifying

the tables layout. The two orthogonal forms of nesting (as defined in Section 4.3.4) for the Code Bubbles paradigm are Shallow and Strict. Shallow Nesting refers to the fact that there is no recursive nesting behaviour; that is: none of the citizens are able to contain another citizen of the same type. Strict Nesting refers to the fact that the application determines which citizens are contained within which.

	Data Structure				Spatial Interface		Primary Citizen	Multimedia	Level of Occlusion	Level of Detail
	Sequence	Array	Set	Allows Nesting	Variable	Fixed				
Code Thumbnails										
-Scrollbar		N/A				✓	Thumbnail		None, but not intended to be read.	Code File
-Desktop			✓			✓	Thumbnail		None, but not intended to be read.	Code Base
Code Canvas			✓		✓		Semantic Container	✓	Likely, depending on zoom.	Software Project
Code Bubbles Paradigm										
-Code Bubbles			✓	✓	✓		Bubble	✓	Very likely. Pertinent information likely visible.	Working Set
-Debugger Canvas			✓	✓	✓		Bubble		Very likely. Pertinent information likely visible.	Working Set

Table 5.1: Summary of applications discussed in Section 0. For each: The type of data structure that can be thought of as being used to store citizens and if nesting is used. The type of Spatial Interface is being used. Details about the types of citizens present.

5.2 Discussion of Relevant Matters Arising

Throughout the thesis there have been threads of discussion that we have not addressed in detail. This was because dealing with them as they came up would have distracted from the main thrust of the earlier sections. We now complete those discussions prior to the conclusion of the chapter. The conclusion to the chapter in Section 5.3 uses the details we have enumerated to discuss the direction of our development of SpIDER.

The topics for discussion in this section are summarised in the following sentences:

- The Spatial Memory considerations of reviewed applications have been mentioned throughout Chapters 4 and 5 but we have not detailed how Spatial Memory considerations will affect the design of our Spatial Hypermedia IDE.

- Multimedia is well utilised in general purpose Spatial Hypermedia but is less common in programming environments utilising Spatial Hypermedia. This would appear to be to the detriment of the usefulness of these applications.
- Our review of traditional IDEs in Chapter 3 revealed a high level of rigidity in interface elements and the way in which content (code) is authored. This limits a programmer's ability to customise their environment for their task. This observation leads us to investigate what happens when steps are taken to reduce rigidity.
- It is infeasible for IDEs to show the entirety of a software project at once and therefore occlusion must be addressed. Different applications choose to hide information in different ways, how may each of these techniques impact a programmer?
- It is reported that Spatial Hypermedia can introduce new difficulties for collaboration [55, 39, 40]. How can these difficulties be handled?

5.2.1 Spatial Memory

Designing and building a programming environment that maximises the opportunities for building, utilising and refining Spatial Memory is attractive because of the “automatic nature” of Spatial Memory [3]. Easily remembering the location of specific content, such as a function, and the relationships it has with other content will enable programmers to focus on producing the logic they require. However, designing an authoring environment to maximise the use of Spatial Memory has some unique challenges. The absolute positioning featured in Spatial Hypermedia applications provides the author with more opportunities to create relationships which can be utilised by Spatial Memory [4, 9, 13] and is therefore a good starting point. This section is split into four parts, each addressing a group of related concepts important to Spatial Memory and the associated challenges. First we address the size of the information space in an authoring environment. Second we look at spatial stability. Third we discuss the parallels between long-term Spatial Memory and programming. Fourth and finally we address the authoring of landmarks.

Size of Information Space. The potentially limitless size of the content in an authoring environment is the primary obstacle. In Chapter 2 we explained the distinction between Single View interfaces and Viewport interfaces [5]. To summarise, a Single View interface

provides access to the entirety of the information space at once whereas a Viewport interface limits what can be seen. A Single View interface is better for forming and utilising Spatial Memory. This is because Single View interfaces allow the user to relate the position of content to the edges of the screen as well as other content. However, the unbounded information in an authoring system means that a Single View interface is not feasible. This leads to many authoring applications utilising panning and scrollbars to access content which can lead to an increase in time spent navigating [16]. In an effort to retain the ability to use the edges of the screen we created the terms Fixed Size spatial interface and Variable Sized spatial interface in Section 2.5.1. The latter refers to authoring applications that utilise scrollbars such as Microsoft Word. The former makes use of a navigational tool, such as surrogates, that splits content up over multiple separate views. Expeditee is notable in that it is the only general purpose Spatial Hypermedia application reviewed that uses a Fixed Size spatial interface.

Activating a surrogate replaces the current view with another view that contains the underlying information. The most common example of a surrogate is a hyperlink, such as those seen on web pages. This example also demonstrates an important distinction. In HTML, a hyperlink can not only be used to navigate between web pages, but also within a single web page. For our purposes, we consider a hyperlink to be a surrogate if it is used to navigate between web pages as this requires a change of view. However, when a hyperlink is used to navigate within a single web page, it is simply moving the viewport and therefore is not a surrogate.

In Spatial Hypermedia—especially Fixed Sized spatial interfaces; surrogates can be exploited to make more efficient use of limited space. An example we have used repeatedly to demonstrate rigidity in traditional IDEs is the relative positioning of functions. In a traditional IDE there is only one ‘degree of freedom’, therefore programmers are limited to positioning two (at most three if both above and below can be used) functions side-by-side. The spatial positioning in Spatial Hypermedia IDE would then provide programmers with several degrees of freedom, allowing for one function to be spatially positioned close to several others. For example, *functionA()* may call *functionB()*, *functionC()*, *functionD()* and *functionE()*. In a traditional IDE, a programmer could place *functionB()* below *functionA()*, and possibly *functionC()* above, but the two remaining functions complicate the positioning.

In a Spatial Hypermedia IDE, it is perfectly feasible for functions *B* through *E* to be positioned so that they circle *functionA()*. However, this approach can only scale until 2-dimensional screen space becomes an issue. This is where surrogates can be used. By replacing the content of one or more of the encircling functions with a surrogate, more screen space is made available. VIKI, VKB and Expeditee all make use of surrogates, with the latter using them exclusively.

Spatial Stability. Another challenge of producing an authoring environment that utilises Spatial Memory is addressing spatial stability. As discussed in Section 2.3, spatial stable content is content that does not alter its position over time. Non-spatially stable content is a hindrance to Spatial Memory [5, 16, 17]. For the purpose of user interface design, this means that components should not move. For example, the items in a drop-down list should not alter their order. However, this becomes more complicated when considering authored content. Applications such as traditional text editors shuffle content up and down a page when new content is added or removed, retaining the order specific tokens appear in, but not the tokens position on screen. Spatial Hypermedia applications, such as many of those reviewed in Sections 4.3 and 5.1, allow authors to spatially position blocks of content. As these blocks of content each contain a portion of the overall document, and as these blocks are spatially stable with respect to each other, this reduces the level of spatial instability when compared with traditional text editors. However, inside these blocks, a traditional style text box is used for storing the content. Therefore, spatial instability is still an issue at this smaller scale.

There is a trade-off to be made. As a spatially positionable block does not interfere with the positioning of other spatially positionable blocks, and the chance of spatial instability occurring within a block increases as the quantity of content increases with respect to the dimensions of the block, the overall spatial stability of the information space can be improved by limiting the quantity of content in each block. Extending this notion, and with the development of a Spatial Hypermedia IDE in mind, if each block contains a single token from a programming language, we should be able to minimise spatial instability. In order to make this trade-off, the design of spatially positionable blocks needs to be considered. One example of a spatially positionable block design that does not lend itself to this trade-off is the citizen type Object in VKB. The presence of a title and several pixel thick border means

that Objects take significantly more screen space than their content alone would require. These design decisions—made by the developers of VKB—communicates the intended use of the application and is appropriate for this use. When designing an implementation of the VKB citizenship framework for programming, the Object equivalent would need to be minimalistic if each Object were to contain a single token. An example of a minimalistic spatially positionable block can be seen in Expeditee. Expeditee's Text Item provides minimal padding between the enclosed content and the edges. Furthermore, its border automatically adjusts to fit the content and becomes invisible when not selected.

One final note on the spatial stability of authored content is worth mentioning. Practice has been shown to have a positive effect on Spatial Memory [14]. This is a moderating factor if we limit the reshuffling of text to occur only as a result of a user action.

Long-Term Spatial Memory Parallels. As discussed in Section 2.1.2, long-term Spatial Memory is arranged hierarchically. Arriving at a node on the hierarchy hastens the recall of possible next steps. Whilst this may have application in other forms of authoring, there is an exaggerated parallel with programming. While it can differ from language to language, programming frequently produces hierarchical structures. At the macro level, software projects, packages and classes exist—each being contained within the last. Even at the class level hierarchical structures exist: inner classes, functions, enclosing statements such as conditionals and loops, assignment and mathematical statements etc. Furthermore, programming code requires maintenance and will be revisited after it was initially written, accentuating the usefulness of long-term memory. The challenge then, is to further emphasise this hierarchical quality of code. VIKI/VKB provide a set of citizens specifically designed to allow the building of hierarchies. Expeditee on the other hand can achieve hierarchical layout through its use of surrogates.

Landmarks. Landmarks tend to be prominent citizens, normally due to their size or positioning. Authors in a Spatial Hypermedia system are able to use landmarks to assist in building a spatial map of their environment. They can use the position of a landmark to relatively position other content in their information space. This is similar to the way in which the edges of the application can be used in Fixed Sized spatial interfaces—however they are not a complete replacement [4]. The goal therefore, is to design applications that maximise the potential for creating landmarks. The option to position content in Spatial

Hypermedia naturally helps with this and allows for prominent positioning. The ability to adjust the font size, style and colour are also important aspects. Finally, the ability to create images and other forms of non-textual media also provides opportunity to create landmarks as these citizens tend to be significant and receive thoughtful positioning by the author.

5.2.2 Multimedia

As just mentioned, non-textual media is well suited for use as landmarks. This is one reason for providing programmers with the ability to insert multimedia into their coding environment. Another is that we know programmers make frequent use diagrams—both during the development [55] and study of software [57, 58]. A benefit of working in an IDE is that they integrate functionality that a programmer may find useful into a single application. It follows then, that programmers should be able to include useful diagrams side-by-side with their code. Code Canvas, as reviewed in Section 5.1.2, is notable for providing programmers with this ability.

In his seminal paper, *Literate Programming* [59], Knuth describes a form of programming that aims to allow programmers to focus on describing their logic to other humans rather than to a computer. This is achieved by blurring the divide between program code and documentation. That is, instead of settling for program code and documentation existing side-by-side, he aims to have them exist as one integrated entity. Reminiscent of Knuth's aims to integrate program code and documentation, we aim to allow for the integration of program code with non-textual media. As we have previously stated, by leveraging Spatial Memory, it is hoped that programmers are able to spend more time crafting their logic and less time on tangential tasks. An example of this integration in use might be the ability to insert a design sketch of the user interface and spatially position code snippets in appropriate places—such as code constructing a specific text field being spatially positioned over that text field in the sketch.

This goal further specifies our design requirements. Our Spatial IDE, SpIDER, must provide users with the ability to spatially layout code in a non-linear fashion. In other words, it must be possible for two programming statements that should execute subsequently in the compiled application be spatially separated on screen.

5.2.3 Rigidity

A Spatial Hypermedia-based IDE will allow programmers to utilise their Spatial Memory for programming. This is the primary goal of the thesis. By examining traditional IDEs (Chapter 3) we arrived upon a secondary goal for the thesis: minimising the rigidity that our Spatial IDE provides as a way of giving a programmer more choice over how they develop software.

We distinguished between two types of abstraction: rigorously defined and informal. IDEs lacked, to their detriment, informal abstractions. This led to rigidity in the interface components and text areas. Previously we cited the Eclipse Outline component as an example of a rigid interface. We consider content to be rigid when the application imposes an ordering that limits the relationships that two fragments of content can have. An example we gave for this was the relative positioning of Java functions in the text editor pane. When a user is unable to achieve a specific action due to the rigidity of the application they likely must accept this limitation as they are unable to easily effect change.

The more rigidity an IDE exhibits, the more important it becomes for an IDE to contain functionality that its users will require to mitigate the loss of flexibility. In the process of adapting the Code Bubbles paradigm to debugging, DeLine et al. were required to significantly alter the design of Bubble Groups to suit the task. However, attempting to provide functionality for everything a user may need, without placing limits on the intended functionality, is an unwinnable battle. The list of required functionality is never-ending. Instead, by providing flexibility (reducing rigidity), users are able to adapt their environment to suit their task, this however, comes at the cost of a steeper learning curve than traditional authoring applications. This learning curve necessitates familiarising participants with Spatial Hypermedia aspects of an application prior to evaluation—as is frequently done through a learning period [56, 12, 40, 51]. In the evaluation of VKB we discussed an undergraduate student who was able to approach report writing differently based on the topic they were writing on. They were able to do this because of the level of flexibility provided by VKB.

This anecdote from the previous paragraph demonstrates that it is important to promote the flexibility of an IDE. It is unreasonable to expect that an IDE could contain functionality for handling every possible scenario. Furthermore, even if you could provide a lot of the functionality that a user desires, then you risk creating a difficult to use UI where only a

small portion of the functionality is used by any single person. By maintaining flexibility and allowing functionality to be quickly built, this issue is avoided.

In order to measure flexibility of a system we defined the terms Fundamental Element, First Class Citizen and System Representation. Using these terms we have reviewed several applications. The spatial positioning of content in Spatial Hypermedia helps address the issue of rigidity. Of the eight Spatial Hypermedia applications we reviewed, there is a varied amount of support for reducing rigidity overall. However, one stands out above the rest. Expeditee's citizens not only achieve a high level of mutability but they also have the added benefit of reflective citizens. The ability to reasonably spatially position content as small as a character is also only available to Expeditee. We believe that Expeditee citizens are best suited to help minimise both content and interface rigidity through its Reflective First Class Citizens.

Other citizens systems are potential candidates as well. We believe that a system with citizens similar to VKB provides the potential to build flexible interfaces. A modification to the visual aspects of VKB Objects would allow smaller amounts of content to be spatially positioned without too much window chrome. This is similar to the approach Code Bubbles took.

5.2.4 Occlusion

When we introduced the terms Fixed Sized and Variable Sized spatial interface we were making a compromise. Whilst both are a sub-category of Viewport interfaces, we chose to distinguish between interfaces where the edges of the application can be used to map the position of content and those that cannot. However, as discussed in Section 2.2, there are other benefits to Single View interfaces that Fixed Sized spatial interfaces may not have, such as the bottleneck caused by locating non-visible components. It follows then, that we should attempt to limit use of viewports when possible. In other words, we should limit occlusion.

In summarising Sections 4.3 and 5.1 we noted a level of occlusion for each of the applications we reviewed—see Table 4.3 and Table 5.1. The level of occlusion an application exhibits refers to the likelihood that there are portions of content that are not visible at a given time. This value estimated by examining the application, inferring (from

publications covering the application) the intended style of use of the application, and assuming that this style of use was applied by a user when interacting with an information space sufficiently large to cause occlusion to occur in the application. When comparing the level of occlusion in multiple applications, we take care to use information spaces of similar size, whilst ensuring that occlusion is occurring in all applications being compared.

Analysis of General Purpose Spatial Hypermedia Occlusion. Consider the level of occlusion in VIKI/VKB. There are three opportunities for occlusion to occur and each can be balanced against the rest.

1. The scrollbars present on Collections/Objects allow authors to position/type more content than can be visually displayed given their dimensions, thus creating occluded content. However, the author is also able to increase the size of the Collection/Object in question, therefore removing occlusion.
2. The scrollbars present on the Canvas behave in the same fashion. Authors are able to position Collections and Objects such that scrolling must be used to see occluded information. In order to remove this occlusion, authors are able to either decrease the size of existing Collections and Objects or re-evaluate the hierarchical containment present in the information space.
3. Authors are able to maximise a given collection such that it fills the screen. This occludes any information that is not in the given collection or any of its sub-collections. The information inside the given collection is subject to points 1 and 2.

The fact that VIKI/VKB provides multiple ways of avoiding occlusion—or at least allowing you to choose which type of occlusion is occurring—combined with the large size of the Canvas suggests that whilst occlusion is likely to occur, it will do so in a controlled manner to a moderate degree.

Consider the level of occlusion in Expeditee. There is only one opportunity for occlusion in Expeditee, and that is through its use of surrogates. Just as the Canvas in VIKI and VKB fills the entire screen, so too does Expeditee's Frame. However, differing from the VIKI and VKB, Expeditee's Frame does not have scrollbar. This provides a hard limit of the information that can be placed on a single Frame. Individual pieces of information may be turned into links which in turn can then be used as a surrogates to navigate to other Frames. Any content not on the currently active Frame is occluded. The expected quantity of content on

a specific Frame is likely to be roughly the same amount as found in a well populated VIKI/VKB Object. Therefore, when compared to VIKI/VKB, Expeditee is likely to have more occluded content.

Occlusion Techniques for Program Code. When designing a Spatial Hypermedia IDE we must pay attention to the style of occlusion our design encourages. We can examine program code in order to help decide what form of occlusion technique (scrolling, surrogates for example) we should include. The hierarchical structure of programming code produces boundaries between fragments of code. For example, whilst editing a function, content within the scope of that function is of high importance. However, once built, that function is ideally treated as a ‘black box’. A similar abstraction occurs when dealing with objects. These boundaries are suggestive places for occlusion mechanics to occur. This analysis does not disqualify any occlusion technique from consideration. It does however suggest that the higher levels of occlusion caused by surrogates is manageable.

This rationalisation ignores the fact that programmers wish to, on occasion, view the high-level connections between fragments of program code—as evidenced by the frequent use of diagrams in software development [55]. Notably, Code Canvas achieves this high-level view through the use of a zoom function [51]. While zoomed out, the programmer is able to use spatial positioning to communicate connections between code fragments. However, they are also unable to read the content inside of functions. A similar concession is made in Code Thumbnails [12]. While Code Bubbles [53] and Debugger Canvas [56] do not make this concession, they also do not attempt to provide this high-level view. This suggests that the prevailing assumption is that an overview is sufficient for this task.

5.2.5 Collaboration

VKB, as a successor to VIKI, attempted to address a potential negative with Spatial Hypermedia. The downside of providing authors with the freedom to spatially arrange content, building their own structures to communicate meaning, is that other individuals viewing the information space may not intuitively understand these structures. Shipman et al. describe this challenge as arising when the “*authors and readers are not the same set of people*” [38]. As we will explain, this is a concern for programming.

Programming is frequently a collaborative effort. While a team of programmers is a team of authors working in the same information space, steps are frequently taken to help them work together. Evidence exists showing that even a well-established team of programmers, who have worked together for an extended period of time, find it necessary to adopt practices that allow them to collaborate [60]. To reiterate, these practices are undertaken even when all authors and readers are from the same group of people—an exaggeration of the stated issue in [38].

More in line with the issue as Shipman et al. states it, is what happens when a new member joins an existing programming team. Brook's law is a frequently used software development philosophy outlined in *The Mythical Man-Month* [61] that states that adding additional team members to a late software project will likely make it later. This is justified by discussing the necessity for new members to be 'brought up to speed'. Regardless of whether the software project is late or not, this same introductory period will be present and should ideally be minimised.

These issues occur without introducing the ability to spatially lay code out. While the ability for individuals to leverage Spatial Memory in a Spatial Hypermedia-based IDE should be beneficial, the addition of this functionality will also cause the above issue concerning teams to be more pronounced. For example, if a team agrees to use a certain spatially based pattern to communicate what code has been tested, then any new member must learn of the pattern and then learn how to apply it.

The Solution in VKB. VKB provides a solution to this issue in the form of a new tool.

Authors in VKB are not only able to navigate an information space as it currently stands, but are able to navigate back and forwards along a timeline. This allows authors to familiarise themselves with the evolution of the information space. This means, that should a new author join in on creating a collaborative VKB workspace, they are able to watch the decision-making process of the authors prior to them, hopefully understanding their intent better and therefore reducing the introductory period.

A theoretical Spatial IDE with the ability to navigate throughout time can be applied to our earlier example where, a team of programmers has used a specific spatial pattern to communicate what code has been tested. When a new programmer is assigned to this team, they are able to look through the development of code from the start. They will be

able to see the first instance of the spatial pattern being produced, any refinements that are made at a later date and have access to a pool of previous examples.

Design and Evaluation of Spatially Laid out Code. Regardless of whether we include the ability to manipulate time in our Spatial IDE, we acknowledge that issues exist around collaborative work when using Spatial Memory [38, 51, 55]. Therefore, it is important that care is taken designing how code can be spatially arranged in a Spatial IDE. We address this issue with some evaluation in Chapter 8.

5.3 Development Direction

Having reviewed and discussed several Spatial Hypermedia systems and their features relevant to the development of a Spatial Hypermedia-based IDE we now explain our course of action for the design and development of SpIDER. We begin by presenting two broad paths for potential development—either adding Spatial Hypermedia functionality to an existing IDE or vice versa. We then explain the advantages and disadvantages of each option, reaching the conclusion that the most pertinent approach, given the aims of the thesis, is to add IDE functionality to a Spatial Hypermedia framework.

The first part of this section addresses our primary goal of creating an IDE that maximises the use of Spatial Memory. Subsequently we address our secondary development goal of minimising the rigidity that the produced environment provides. Each of these parts uses our review of existing Spatial Hypermedia systems and our discussion from Section 5.2 to identify which of the reviewed applications provides the best framework for our needs. In the final part of this section, we conclude by providing a summary of the chapter.

5.3.1 IDE into Spatial Hypermedia

Through our analysis in this chapter we have identified distinguishing features of various Spatial Hypermedia applications. Some of these distinguishing features are novel; others represent a design decision between multiple alternatives. For each, a judgement has to be made about their utility to programming and the scope of this research. When considering alternatives and our goal of building a Spatial Hypermedia IDE, one question that stands out is: Should we work to integrate Spatial Hypermedia functionality into existing IDEs? Or, should we work to integrate IDE functionality into existing a Spatial Hypermedia framework?

From a technical standpoint, both of these options are feasible. Ideally, and with limitless time, this decision would not affect the finished product from the user's perspective. However, realistically, this decision will have repercussions throughout the entire development of the Spatial Hypermedia IDE. All of the applications we reviewed in Section 5.1 opted to include Spatial Hypermedia functionality into an existing IDE. This was achieved by building extensions to existing IDEs: Code Thumbnails, Code Canvas and Debugger Canvas extended Microsoft Visual Studio, and Code Bubbles extended Eclipse. The degree of integration varied, from Code Thumbnails that added new views, to Code Bubbles that heavily modified how code interaction was implemented. A clear advantage of this approach is that the developers of the research application do not need to reinvent various IDE functionality. In other words, functionality such as compile and run, content assist and syntax highlighting are already provided.

When we consider our goal to maximise the flexibility of a programming environment however, this option starts to look less promising. Much of rigidity we identified in Chapter 3 is present in those applications reviewed in Section 0. These applications have not only made use of the provided IDE functionality but also the applicable interfaces, which leads to the rigidity being maintained. For example, because Code Canvas, Code Bubbles and Debugger Canvas make use of the editor provided by the underlying IDE, they are provided with syntax highlighting. However, they are also unable to adjust the font size or style of individual tokens—a consequence of using that specific editor. It may be possible to remove, on a case by case basis, the rigidity from the interfaces provided by the underlying IDE, however this removes the benefit of choosing this option. This is not to say that the approach of introducing Spatial Hypermedia functionality is not able to reduce the rigidity: the concept of the Bubble in Code Bubbles and Debugger Canvas have allowed for the spatial arrangement of functions. However, these benefits are also available when using a Spatial Hypermedia application as the base system.

We now consider the other approach—adding IDE functionality to a Spatial Hypermedia System. This can be achieved by augmenting a general purpose Spatial Hypermedia system with the libraries provided by IDE plugin systems, such as the Eclipse Java Development Tools (Eclipse JDT). This option allows us to programmatically obtain the results of calls to IDE functionality without using the user interface components provided by the IDE.

Unfortunately this means that we will have to design and build these graphical user elements ourselves; but in doing so, we can make a conscious effort to build flexible spatially aware interfaces.

We decided to add IDE functionality to an existing Spatial Hypermedia framework. Specifically, we decided to develop a Spatial Hypermedia IDE whose Spatial Memory considerations and Citizenship would be similar to one of the applications reviewed in Section 0. This allowed us to design a system with the goals of maximising flexibility and the use of Spatial Memory. Further decisions were based around choosing a particular framework from these applications and whether we would have to build an entirely new Spatial Hypermedia application (based on that framework) or extend an existing application.

5.3.2 Maximising the use of Spatial Memory

The learning curve associated with traditional editing applications is assisted by the similarities each has with others. For example, a relative editing environment featuring a scrollbar is ubiquitous across multiple word processing applications; Microsoft Office and Libre Office to name two. Furthermore, the functional behaviours associated with these features are minimally different. The aforementioned scrollbar for example is present for all but the most trivial documents.

Spatial Hypermedia functionality is more subtle. Whilst some mainstream applications can be classified as Spatial Hypermedia, they tend to include fewer spatial solutions in their design, particularly for control mechanisms. The use of tabs in Microsoft OneNote is an example of such a decision. The four applications reviewed in Section 4.3 also follow this path—though perhaps coincidentally. The result of using traditional style components when feasible is that ‘computer literate’ individuals are able to transition to your application with only moderate cognitive effort. However, the downside is that opportunities for leveraging Spatial Memory are being missed.

We do not intend miss opportunities when designing SpIDER. This means that we are prepared for SpIDER to be difficult for people to pick up, but for this cost we are able to explore the potential of an environment that makes the most out of Spatial Hypermedia and Spatial Memory. This is not to say, however, that we do not intend to focus on making SpIDER accessible.

SpIDER Accessibility and Collaboration. In Section 5.2.2 we argued for the importance of including the ability to insert non-textual content into a software development environment. We stated that we wished to achieve integration between code and multimedia, and gave the example of GUI code being spatially positioned on-top of a design sketch. In order to achieve this we need to provide programmers with the ability to spatially lay out individual statements. In Section 5.2.1 we evaluated both styles of Spatial Hypermedia framework against the goal of maximising spatial stability and we concluded that both Expeditee Items and VKB Objects have the potential to limit spatial instability. However, the Text Items provided by Expeditee have the benefit of a minimalist design, making them more suitable for small quantities of content, which in turn makes them the better choice for designing a technique for non-linear spatially laid out code.

In Section 5.2.5 we discussed the challenges associated with collaboration in Spatial Hypermedia. We reviewed the solution provided by Shipman et al. in the design of VKB [38, 40]. While we acknowledge that there are questions that need to be asked concerning collaboration in Spatial Hypermedia, we do not intend to address this issue directly in this thesis. Instead, we conjecture that by providing programmers the ability to spatially layout code, they will develop spatial development patterns that can be used to communicate between members of a team, thereby assisting with collaboration indirectly. Additional evaluation will be needed to support this claim.

Maximising Flexibility. In Section 5.2.3 we discussed the issue of rigidity and how we evaluated various Spatial Hypermedia applications throughout Sections 4.3 and 5.1. We came to the conclusion that the prevalence of First Class Citizens in Expeditee and their reflective nature provide a suitable framework for limiting rigidity. In Section 5.3.1 we then outlined our reasoning for integrating IDE functionality into a Spatial Hypermedia system—so as to not retain the rigidity present in traditional IDEs. This leaves us with the task of re-implementing the user interface functionality that we would have retained if we had used the other approach. This includes: syntax highlighting, content assist, executing and debugging. In order to maximise flexibility we will implement this functionality with Expeditee citizens.

Looking Forward. In the order they are mentioned in this section, we:

- Cover the design of our system for spatially laying out code in Section 7.1.
- Evaluate this system in Chapter 8.
- Engage in a thought experiment in Chapter 9 where we explore a variety of possible spatial development patterns.
- Detail the design of IDE functionality in Sections 7.2 before its implementation details in Sections 7.4 and 7.5.

5.3.3 Summary

This chapter reviewed four instances of IDEs utilising Spatial Hypermedia. We found further evidence that programmers are able to utilise Spatial Memory to help them while programming. We also identified an emerging trend of incorporating multimedia into the IDE due to the desire of programmers to easily see relationships over entire software projects.

In Section 5.2 we brought together several points of discussion so that we could explain our decision on how to move forward on designing and building a Spatial IDE. Having completed this discussion, we then explained our choice in development direction—using Expeditee as the framework for building our Spatial Hypermedia IDE. Our explanation was divided into two parts. We first explained the merits of beginning with a Spatial Hypermedia environment and adding IDE functionality. This was followed by evaluating which general purpose Spatial Hypermedia framework allowed us to maximise the use of Spatial Memory whilst limiting rigidity—we found that Expeditee’s citizens and framework best suited our goals.

Expeditee is an open source application under the GNU General Public Licence. This makes it easier to directly extend Expeditee rather than design and build a new application mimicking its functionality. Furthermore, as Expeditee was developed (and is the subject of work by other researchers) at the University of Waikato, the potential to query colleagues on its design decisions was an additional boon.

Chapter 6

Expeditee In-Depth

This point marks a transition in the thesis. Prior discussion has been focused on the theoretical concepts that should be considered when designing a Spatial Hypermedia IDE. We now move on to more practical matters. In this chapter we expand on the review of Expeditee that began in Section 4.3.3. Subsequently, Chapter 7 documents the development of our Spatial IDE (SpIDER) and how it expands on the existing Expeditee support for authoring code [62] by providing an environment rich in its utilisation of Spatial Hypermedia to allow for novel spatial code layouts and Spatial Hypermedia appropriate implementations of traditional IDE functionality. Notably, Chapter 7 includes a description of the design and implementation of an algorithm dubbed the flow walker, which is used to interpret the novel spatial code layouts produced by programmers and transform them into a serialised string suitable for compilation. The flow walker algorithm is then evaluated in Chapter 8.

In Chapter 5 we explained our choice to use Expeditee as the framework for developing our Spatial IDE. Previous discussion of Expeditee (Section 4.3.3) was limited to pertinent information useful for making this decision. This limitation resulted in a significant amount of detail, especially detail concerning authoring, being omitted. We now present this detail. Whilst we do explain a significant portion of Expeditee in this chapter, this explanation is not intended to be used as a complete reference guide. The content presented is instead intended to help the reader understand specific Expeditee functionality that will be used in the development of SpIDER.

We begin by explaining what the developers of Expeditee aimed to achieve by building the system (Section 6.1). These goals provide perspective and help understand the design

decisions that were made in building Expeditee. Of particular important to the design of SpIDER is Expeditee's innovative form of direct manipulation, which is described next. Following this we cover the structure of Expeditee's user interface (Section 6.2). We then expand on the explanation of the surrogate system Expeditee uses (Section 6.3). Three subsequent topics are then covered over two sections (Sections 6.4 and 6.5) and each explain a different aspect of frame authoring. They are:

- *Textual Content*. How authors create and manipulate text on Frames.
- *Annotations*. How textual content can be tagged to interact with the information space. Including the creation of non-textual content.
- *Polylines and Polygons*. How authors can create shapes and boundaries.

6.1 The Goal of Expeditee

Expeditee directly follows from the development of another Spatial Hypermedia system called Knowledge Management System (KMS) [41], which in turn built on an early hypertext system named ZOG [63]. Following his involvement with ZOG and KMS, Akscyn used his experience to design and develop Expeditee as a modern and open source reimplementation of KMS.

Developed at the University of Waikato in New Zealand, the goal of Expeditee was to be an environment that allowed authors to quickly and logically organise information [41, 47]. In order to achieve this goal, the development subscribed to the following principles:

1. Users shall be able to enter and modify content into the system quickly.
Functionality for organisation and theming of entered content shall be easily accessible; thus allowing users to enter content in an unrestrained fashion, and subsequently reorganising.
2. A minimalist environment along with tools to customise and extend it will be provided to the user. Such tools thus allow the environment to be moulded to each task the user undertakes rather than having to mould the content to the environment.
3. All content should be treated as uniformly as possible so as to allow the same interaction methods to be applied to different types of content. For example,

having learned how to alter the size of text, a user should be able to alter the size of images with the same method.

4. Multiple ways may be provided to achieve a variety of actions so that users can interact with the system as is fit for the task. For example, some tasks require speed to be primary over organisation—such as note taking—and for other tasks—such as restricting existing content—the opposite is true. Providing multiple ways to position content leaves the decision to the author.
5. The system shall support the manipulation of large quantities of information. Users shall be able to build hierarchical structure to work with information at multiple levels of aggregation.

As shown in the remainder of this chapter, much thought was given to the style of interaction in order to achieve the above principles. Notably, this results in Expeditee's efficient style of direct manipulation.

Direct Manipulation Style. Expeditee's manipulation of Items differs in four key ways from the direct manipulation common in other applications: immediate text entry, accelerated access to commonly used interactions through the promotion of mouse buttons and function keys, the ability to pick up and put down content as opposed to the more conventional drag and drop interaction which is implemented by attaching content to the user's cursor. Expeditee makes use of a 3-button mouse.

Expeditee does not require the user to create a text area prior to adding text content to a Frame. Instead, Text Items and their bounding boxes are automatically generated and maintained as the user types. Characters typed appear at the position of the mouse cursor on the screen. The generated bounding box expands as a user adds content to the Text Item. When the cursor is over an existing Text Item, newly typed content appears in that Text Item rather than causing a new Text Item to be created.

This implementation of text entry follows the first principle by allowing authors to produce content first, and organise it as is appropriate. Parnin et al. note that programmers often suffer from disrupted workflow as they attempt to work with two areas of source code at once [64]. We believe that this ability to enter content in an unrestrained fashion will help alleviate this issue.

As will be further explained in Section 6.2.2, the interaction caused by a mouse click depends on the Item the cursor is pointing at. The developers of Expeditee made explicit effort to logically pair related interactions using the same button. For instance, navigating between Frames is a common action. Left clicking on a linked Item follows the link, changing the currently active Frame; similar to the navigation action seen in web browsers. Left clicking in blank space is logically paired, causing a 'back' action to execute, taking the user back to their prior Frame. Additional to this pairing is the effect of left clicking on an Item that, at the time of clicking, is not linked to another Frame. Doing so will still perform a navigation action, creating a new blank Frame and taking the user to it. If the user adds any content to the new Frame, then the Item used to gain access to it is altered to link to the new Frame. Assigning the most easily used gesture (left mouse clicking with one's index finger) to these actions demonstrates the importance Akscyn places on navigation and Frame creation.

The design goal of uniform interaction across object types can also be seen to pervade the utilisation of function keys in Expeditee. For example, the F1 key can be used to increase the size of Items. When the cursor is over a Text Item and F1 is pressed, the font size of that Item is increased. When the cursor is over a line, the thickness of the line increases. Other Expeditee Item types also respond to the instruction to increase their size. Similarly, the F2 key can be used to decrease the size of Items. The colour of Items can be adjusted with the F3 key which cycles through a colour wheel specified by the user.

Commonly seen in many applications is the ability to move elements through drag and drop style operations. In contrast to this approach, Expeditee instead provides operations to pick up and put down Items. When an Item is 'picked up', it is attached to the user's cursor rather than the Frame. The user is now able to move their cursor, thus moving the attached Item (or set of Items), subsequently putting the Item down in its new location. Performing operations (other than putting content down) while content is attached to the cursor does not cause the content to be put down. A notable usage of this is performing navigation operations while content is attached to the cursor, allowing the user to pick up content from any one Frame, travel with it, and easily put it down on another. In a more conventional authoring environment, a user would instead use clipboard functionality to move a non-trivial amount of content a non-trivial distance; this demonstrates that

Expeditee's pick up and put down operations fill the roles of both drag and drop and cut and paste operations prevalent elsewhere.

Like navigation actions, the ability to pick up and put down Items is a commonly used operation. Clicking the middle mouse button while over an Item, without any Items attached to the cursor, will pick the Item up. Clicking the middle mouse button with an Item attached to the cursor will put the Item back down. If a group of Items enclosed in a shape (such as a rectangle), then picking up that shape will also pick up all the Items within it.

Whereas the middle mouse button picks an Item up, and is therefore similar in function to a cut operation, the right mouse button instead attaches a copy of the Item it is activated over, making it similar to a copy operation in other systems. Having attached a copy of an Item (or set of Items) to the cursor, the user is then able to use the middle mouse button to put the copy down. Alternatively, right clicking with content attached to the cursor will put a copy of the content attached to the cursor down, retaining the content on the cursor.

Two other commonly seen interface features in other applications are reserved areas for specific content—such as error message components—and tooltips. These features provide the application with methods to communicate information to the user in an unobtrusive way. While Expeditee does contain a reserved area for messages to the user (MessageBay, Section 6.2.2), it also is able to utilise its ability to attach content to the cursor—a concept that works due to Expeditee's functionality allowing Items to be picked up and put down. For example, when an Expeditee user performs a search operation (an Expeditee Action, see Section 6.4.3), the executed script may place the results of the completed search on the user's cursor so that they can immediately be used or placed in a convenient place on the Frame for further consideration. Regardless of which method is used to provide the Expeditee user with feedback—MessageBay or content attached to cursor—they are provided with First Class Citizens, suitable for manipulation as the user sees fit.

6.2 User Interface Structure

Expeditee's user interface is built out of two Frames as seen in Figure 6.1. One of these Frames is the primary editing area of Expeditee and as such takes up the majority of the screen real estate. The Frame title, Frame name and any other authored content is spatially

positioned in this area. The other Frame is docked beneath the primary editing area. It is called the Message Bay and is used as an output console for Expeditee.



Figure 6.1: A screenshot of a newly created Frame in Expeditee.

6.2.1 Frame Title and Name

With default settings, when a new Frame is created in Expeditee, users are provided with a minimally populated space. Figure 6.1 is an example of a typical newly created Frame. Two text Items are created for the author: a title and a Frame name. The title is positioned in the top left-hand corner and is provided with some initial user-determined content. Other than the fact that Expeditee generates the title, it is functionally identical to a normal Text Item. This means that the author is free to edit, reposition, resize, manipulate or even delete the title to fit their needs.

The Frame name is positioned in the top right-hand corner of the Frame. Like the title, it is also a Text Item. Unlike the title however, the content of the Frame name is designated by the system; calculated sequentially. It has its mutability significantly restricted and can therefore not be altered. The restrictions ensure that the Frame name Item cannot be repositioned, resized, manipulated or deleted. This allows code within Expeditee to reliably access it. A copy of the Frame name can be made and is subject to the full range of manipulation normal Text Items are. These restrictions are enforced by a permissions model implemented into Expeditee. Whilst this same permissions model can be used by

authors to restrict their own content, documenting how it works is unnecessary for this thesis.

6.2.2 Message Bay

A reserved area at the bottom of the application window contains the Message Bay. Figure 6.2 shows a magnified screenshot of the Message Bay. One message is present in the screenshot: it notifies the user that no proxy settings are detected and will therefore assume a direct connection to the Internet if Expeditee needs access to online content for one reason or another.

The Message Bay is one of the ways Expeditee can communicate with the user without using specially designed widgets such as dialog boxes and is itself a Frame. Text content that appears in the Message Bay is set to have the same limited mutability that the Frame name does. The Message Bay is attached to the application window; meaning that messages that appear in the Message Bay are consistent across Frames.

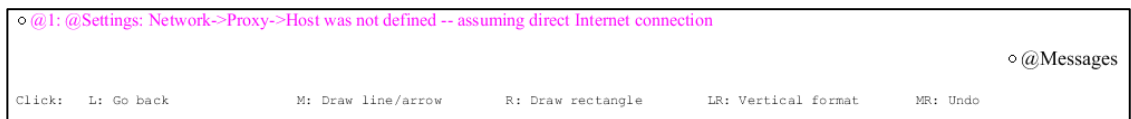


Figure 6.2: A screenshot of the Expeditee Message Bay.

A final feature can be seen at the very bottom of Figure 6.2—some context-sensitive help text. As Expeditee makes heavy use of the mouse for interacting with Items on a Frame, this help text serves to remind a novice user of what each mouse button. This help text updates based on the position of the cursor. The same mouse button will do different actions depending on the type of Item that it is positioned over. For example, in Figure 6.2, the mouse cursor can be seen hovering in blank space. The help text informs us that—while in blank space:

- Left clicking will cause a back navigation to happen. A back navigation in Expeditee is synonymous with a back navigation in a web browser.
- Middle clicking will begin to draw a line.
- Right clicking will begin to draw a rectangle.
- Left and right clicking at the same time will perform a vertical alignment action. This is a pre-defined algorithm that will cause a selected group of Text Items to

reposition themselves slightly so that the vertical distance between them is consistent.

- Middle and right clicking at the same time will restore the last Item that was deleted.

If the mouse was instead hovering over a linked Item, then what each mouse button does would change and the help text would update to reflect this. For example, left clicking would follow the link on the Item, changing which Frame is visible.

The mouse button(s) assigned to do a specific action provide insight into the importance that the developers have placed on that action. For example, the action *vertical format* requires two mouse buttons to execute. This suggests that the developers see this action as being used less frequently than actions requiring only one mouse button. The action *Undo* also requires two mouse buttons. In this case however, the same mouse button combination, when over an Item performs a *Delete*. As these actions are counterpoints to each other, they share a mouse button combination. As accidental deletion is a hindrance, it is assigned a 2-mouse button combination to increase the chance that deletions are a result of direct intent by the user.

6.3 Creating and Manipulating the Frame and Linking System

Expeditee's surrogate system allows a specific piece of content—from the collection of all content on the Frame—to link to a specific Frame, thus allowing for a many-to-one relationship for Frames. Clicking on an Item that contains a link adjusts the viewport so that the target Frame is being shown, thereby allowing access to previously occluded information.

In Section 6.3.1 we will document the basic method for creating links to existing Frames before moving on to show how new Frames can be created in Section 6.3.2. We detail the creation of links to existing Frames first because it demonstrates that the surrogate system is able to create cross-cutting links. In contrast to link creation, creating new Frames is done sequentially, where the new Frame (destination Frame) is automatically given a unique identifier based on the unique identifier on the origin Frame. Frames created using this method are conceptually stored in the same collection. This collection is referred to as a

FrameSet—a mechanic covered in Section 6.3.3. Finally, Section 6.3.4 looks at two methods for navigating between Frames.

6.3.1 Linking to Existing Frames

In order to create a link to a specific Frame, the name of the target Frame must be known. The screenshot shown in Figure 6.3 shows a frame with the name *home1*—this is the initial Frame that the user is shown when Expeditee is started.

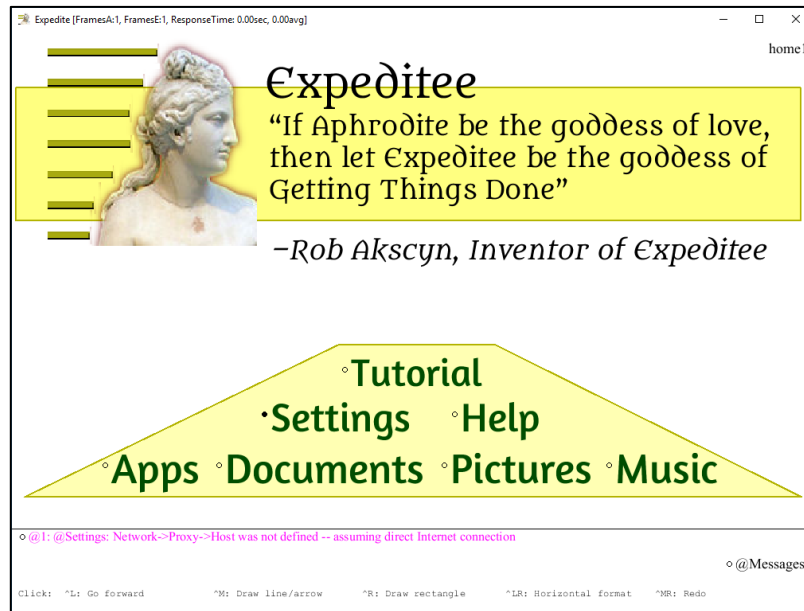


Figure 6.3: A screenshot of the initial Frame that Expeditee presents to the user on startup.

By taking a copy of this name—or manually typing the Frame name—and using property injection (Section 4.3.3), a user is able to have any Item link to the Frame with that name. For example, a user may want to create a 'Home Button'—an image that when clicked, navigated to *home1*. This can be constructed in two steps. First the user is required to import the image into Expeditee. This can be achieved by dragging the image they want from their computers file system. The second step is to have the newly created image link to the appropriate Frame. The user can inject the following property into that image: *Link: home1*. As mentioned earlier, an Item that links to another Frame features a small circle to the left of the Item. In this example—creating a home button—it is reasonable that the user might not want this circle. Another property can be altered to make this circle invisible: the link mark. Injecting the following name-value pair into the image will achieve this:

LinkMark: false. *LinkMark: true* can be used to bring the circle back. This same process can be used to create links from other types of content such as Text Items.

6.3.2 Creating New Frames

Before an Item can be linked to a Frame, it must exist. If a user wishes to create a new Frame, rather than link to an existing one, they simply need to left click on the Item they wish to be the link. As long as that Item is not already linked to a Frame, a new Frame will be created. Figure 6.1 shows what a Frame looks like when initially created. A Text Item is generated to act as the title for the new frame, the content of the title is the same as the content of the link used to create the Frame. Other than the title and the Frame name, the Frame is left blank so that the user can customise it to suit their task.

6.3.3 FrameSets

Users are able to create collections of Frames under a specific name. These collections are called FrameSets. Pressing F6 while the cursor is over a Text Item creates a new FrameSet that is named after the content in that Text Item. The screenshot from Figure 6.3 shows a Frame from the *home* FrameSet. When a new FrameSet is created it begins by creating the first frame in that FrameSet. For example, pressing F6 on a Text Item with the content “thesisnotes” will create the FrameSet *thesisnotes*, with the first Frame being *thesisnotes1*. Beyond the ability for users to categorise their work, this also provides other benefits. One such benefit is the shorthand this provides when using property injection to create links. If a user wishes to transform an existing Item on a Frame to a link to another Frame in the same FrameSet then the name of the FrameSet does not need to be specified in the name-value pair. For example, under those circumstances just stated, *Link: thesisnotes2* and *Link: 2* are equivalent. Links between frames in different FrameSets require the frameset name prefix to be present.

There are two methods that allow users to pre-populate newly created Frames. One of these is the *Zero-Frame* system. The other makes use of annotations and will be described in Section 6.4.2. Every FrameSet has a special Frame called the zeroth Frame that can be used to provide content to new Frames created in that FrameSet. Navigating to the zeroth Frame in a FrameSet, the user is able to set up a template for new Frames created in that FrameSet. Extending our example from earlier, the home button that the user created could be spatially positioned in the bottom-left corner of the zeroth Frame. Once this is

done, any new Frames that are created in that FrameSet will feature this home button at the specified position.

6.3.4 Frame Navigation

Navigation actions between Frames can be split into two styles, random access and ordered pagination. Each style can be achieved in multiple ways. We will document one from each style. Random access Frame navigation is achieved using a surrogate system. If an Item is linked to another Frame then left clicking on it will follow that link. Once on that Frame, left clicking in white space (where no Items are present) will navigate back to the prior Frame.

This provides similar functionality to that seen in internet and file system browsers.

Alternatively, ordered pagination is based on the Frame name—specifically the suffix of the Frame name, the Frame number—and is similar to the style of navigation seen in application such as Microsoft PowerPoint. By using the left and right directional keys on the keyboard users are able to look through the Frames in a FrameSet in order. For example, if the user was on *thesisnotes5* and pressed the left directional arrow then they would be on *thesisnotes4*. This technique allows authors to view pages in the order they were created, and can also be used to access the zeroth Frame.

6.4 Text Entry in Expeditee

As with most other authoring applications, Text is the primary type of content for communicating information. The four guiding principles detailed in Section 6.1 significantly influence how Text is authored, modified and used in Expeditee. Section 6.4.1 documents how textual content can be easily added and efficiently arranged on an Expeditee Frame. Section 6.4.2 and 6.4.3 shows how special Text Items (annotations) can be used to both customise the information space and help provide uniformity for non-textual content.

6.4.1 Adding and Modifying Text Content

Users are able to create text by positioning their mouse cursor where they want content to appear and begin typing. Unlike in most other authoring applications, clicking the mouse to gain focus is not required. As the user is typing a Text Item is created. Moving the mouse cursor to another position will cause that Text Item to stop being edited. Moving the mouse cursor back to a Text Item will re-enable the editing of that Text Item. As with Object in VIKI/VKB and Text Nodes in HTML, a Text Item can be considered a ‘mini-world’ traditional text editor.

As text is frequently the primary way of communicating information, several hotkeys have been assigned to help manipulate text. All of these hotkeys adjust a property that could alternatively be adjusted using property injection. Furthermore, many perform similar functions on non-textual content.

- F1 and F2 increase and decrease the font size of a Text Item respectively.
- F3 alters the font colour of a Text Item by cycling through a colour wheel that is customizable by the author.
- F4 cycles between three bullet point options on a Text Item: no bullet point, annotation Item (discussed shortly) and asterisk.
- F7 alters the font style of a Text Item by cycling through a font style wheel. Options are: normal, bold, italic, bold italic.
- F8 alters the font family of a Text Item by cycling through a font wheel that is customizable by the author.

Some of the properties that Text Items contain do not make sense in other Items—such as font style for example. However, the majority of properties that text supports are also supported by other items.

6.4.2 Annotations

By prepending a '@' to the beginning of a Text Item, an author tags it as an annotation. This communicates to the Expeditee system, and other users, that it is not to be considered a normal Text Item but instead indicates it is for the system to use, or piece of documentation—such as a note. The system is able to use annotation to alter the information space. For example, the annotation '@LogDir' is used to control where on the file system log files will be written. There are two categories of annotations: system reserved and user-defined. In this section we will document three prominent system reserved annotations: templates, overlays and images. Section 6.4.3 will document user-defined annotations.

Templates. In Section 6.3.3 we discussed the Zero-Frame technique and discussed how it could be used to create a *template* for Frames in a FrameSet. We used this language for the purpose of explanation. However, Expeditee also uses the word template to describe a different set of functionality. We wish to distinguish between Frame templates, like the

Zero-Frame technique, and content templates. Content templates provide a method for authors to provide default properties to certain content. One prominent content template is the item template (`@ItemTemplate`).

By placing a `@ItemTemplate` annotation on a Frame, the author is able to set the initial font properties for all newly created Text Items on that Frame. When a new Text Item is being created, the system checks for the presence of `@ItemTemplate` on the Frame and sets the properties of the new Text Item to match those of the template. The font style, size, colour etc. of the provided template are examined and used to populate the properties of the new Text Item.

When multiple templates are provided, Expeditee will use the top-left-most positioned one. This provides authors with the ability to have multiple templates set up and to switch them in and out as desired. A variety of content templates exist, each with a provided default which can be altered on a settings Frame. One example is the annotation template (`@AnnotationTemplate`) which allows the author to set the default font properties of Text Items that are created by typing '@'.

Overlays. Earlier in Section 6.3.3 we stated that there are at least two ways of pre-populating newly created Frames and went on to explain the *Zero-Frame* system. Figures 6.4 and 6.5 demonstrates another method for achieving pre-population. In Figure 6.4, some useful links are provided through the use of an active overlay. When the active overlay annotation is (`@ao`) deleted, as it is in the latter, these links are removed with it.



Figure 6.4: Screenshot snippet of Expeditee documentation home Frame **with** useful links present along the top.



Figure 6.5: Screenshot snippet of Expeditee documentation home Frame **without** useful links present along the top.

An overlay is an annotation—and another form of Frame template—that causes the content of one Frame to be overlaid on another, the primary of which is the active overlay, created

with the annotation *@ao*. When the Expeditee system encounters an active overlay annotation, it checks if it has a link property. If it does, it dereferences this link and overlays the content found on the destination Frame over the Frame that the annotation is on.

Both of the Frame template methods we have covered have some potential shortfalls. As stated in Section 6.3.3, the contents of the zeroth Frame will be used to populate newly created Frames. However, any existing Frames in the FrameSet will not have the effect applied. This can make it tedious to make updates to the Frame template of the FrameSet; requiring manual updates to existing Frames. Overlays however only apply to the Frame that their annotation is placed on, therefore requiring the overlay link to be inserted on each new Frame.

Combining the two methods can be an effective way of overcoming the shortcomings of each. Using the home button example from earlier we can devise a way to get the flexibility of *@ao* with the reliability of the *Zero-Frame* system. Instead of placing the home button directly on the zeroth Frame of the FrameSet, follow these instructions.

1. Create a FrameSet especially for Frame templates, say: *FrameTemplates*.
2. Place the home button control on the first available Frame in the FrameSet, say: *FrameTemplates1*.
3. Create a *@ao* and link it to *FrameTemplates1*.
4. Place the created *@ao* on the zeroth Frame of the Frameset you wish to contain the home button.

This technique works by ensuring that the *@ao* will be present on all new Frames in the FrameSet and allows modifying the Frame template by modifying *FrameTemplates1*.

Adding the template to existing Frames simply requires adding the *@ao*. Two other benefits of this technique are, the reusability—other FrameSets are able to use the Frame Template, and scaling—multiple Frame templates can be created in this fashion and be used together when appropriate.

Non-textual Content. Expeditee provides authors with the ability to insert non-textual content such as images. Perhaps surprisingly, it achieves this by representing images as specially formatted text. When we introduced the worked example of creating a home button in Section 6.3.1 we stated the first step was to import a desired image into

Expeditee. We achieved this by using drag and drop functionality on an image file from the file system. However, behind the scenes, Expeditee is simply referencing this image on the file system rather than truly importing it, this is similar to the way in which images are handled in HTML.

When the function key F10 is pressed, Expeditee enters a special state that is referred to as 'X-Ray Mode'. Pressing F10 again returns Expeditee to normal functioning. By entering X-Ray Mode, the author is asking the system to stop interpreting certain annotations, and instead show the underlying Text Item. Figure 6.6 repeats the screenshot of the home Frame of Expeditee previously featured in Figure 6.3, but this time with X-Ray Mode enabled. In place of the image of the goddess Expeditee, an annotation is present, this annotation directs the system—when outside of X-Ray Mode—to visually replace the annotation with the referenced image.

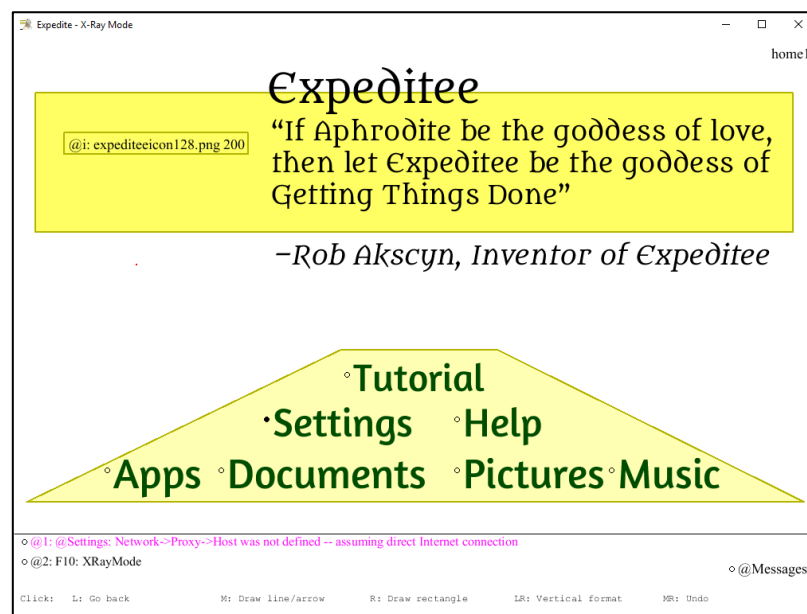


Figure 6.6: A Screenshot of the Expeditee home screen with X-Ray Mode enabled.

Authors are able to manually construct these annotations as an alternative method for adding images to an Expeditee Frame. They may want to use this method in order to have more fine-grained control over the resulting image. We can divide the annotation in Figure 6.6 into three parts in order to help understand how it functions. The first part is the annotation tag, in this case '@i:'. The lowercase i informs an algorithm running over Expeditee Frames that it has encountered an image and the colon acts as a separator

between the tag and the remaining content (the following space is optional). The second part is the file name of the image: “expediteeicon128.png”. Normally an author would be required to provide a full path to the appropriate image, however, a ‘images’ folder is created on installation of Expeditee and the system knows to check there for resources. Finally, the third part of the annotation is the set of properties to assign to the image. In this case, the number 200 specifies the desired width of the image. The inclusion of the optional third part will cause the imported image to scale. As a height is not provided, the width to height ratio is maintained. Other non-textual content also uses the same annotation system, each with its own annotation tag and set of properties.

6.4.3 User Scripting and Actions

Running Actions. Action is the name given to an Expeditee script that runs at the demand of the user. In contrast to the algorithms that interpret system reserved annotations such as those described in Section 6.4.2, actions are able to react to user-defined annotations. Consider a theoretical Action called ‘SlideShow’. When the user executes this action it will produce a series of image files, each a screenshot of a Frame in a specified FrameSet. The user may wish to omit some Frames from the finished product; perhaps they are note pages. This Action is then able to monitor Frames for the ‘@notes’ annotation and skip any Frames it finds with this annotation. Furthermore, the ‘Slide Show’ Action could omit any annotations from the produced images, thereby removing any minor notes made on pages that should otherwise be in the finished product.

An Action can be created by injecting a property into an existing Item. Extending our ‘SlideShow’ example, creating an Item to run this Action is a two-step process. The first step is for the author to create a Text Item to act as the control, ideally with an appropriate label such as “Export Slide Show”. The second step uses property injection. Injecting a Text Item with the content

“a: SlideShow”—we will call this the Action string—into the created control sets it to run the ‘SlideShow’ action when left clicked. In the same way as linked Items gain a hollow circle, Items with actions gain a filled in black circle. Figure 6.6 has an example of a Text Item with an associated action: the Text Item with content “Settings”.

Creating Actions. Expeditee has two ways for a user to create their own Actions. The first is a combination of the fact that the code is open source and uses a plugin system, thus

allowing programmers to write their own Actions in Java. These Actions are then compiled and executed through Java Reflection by the Expeditee runtime. The second method allows programmers to write scripts inside of Expeditee using Text Items on Frames. Multiple languages are available for this: SIMPLE, JavaScript and Python. These scripts have access to library functions that allow for reading and manipulation of Frame content. Tutorials are provided in the form of Expeditee Frames in the standard download of Expeditee. With either of these methods, the programmer is free to specify new annotations (acting as reserved keywords) that are specific to the algorithm they are creating.

Providing Input. There are two ways that Actions can be programmed to accept input parameters. The first is that parameters can follow the name of the action in a space-separated list. For example, if our 'SlideShow' example was extended to accept the desired dimensions for its generated images, the Action string could look like this: "a: SlideShow 1024 1080". This technique requires the parameters to be set prior to the Action string being injected into the control. The second option allows parameters to be decided upon running the action. This is achieved by creating a Text Item with the content of the space-separated parameter list and injecting it into the control. Injecting the Text Item with content "1024 1080" would achieve the same result as our example from the first method, but would require the property injection each time the action was executed.

6.5 Creating Polylines and Polygons

Most of the functionality concerning Polylines and Polygons was explained earlier in Section 4.3. As an amendment to this prior discussion the spot welding operation should be mentioned. Spot welding is a process that visually connects two Polylines or Polygons together and creates an additional Dot Item if necessary. Figure 6.7 shows the starting point for two examples of spot welding. A rectangle and polyline are present and separate from each other. Spot welding can be used to merge them together—either completely, resulting in a single Item or visually, retaining each Item as a separate entity.

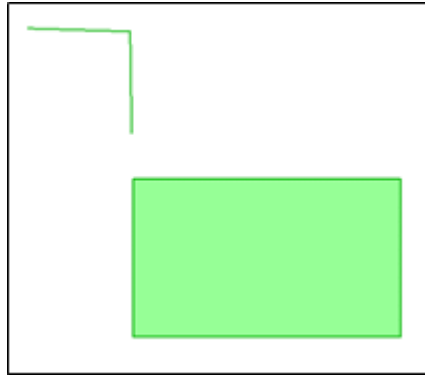


Figure 6.7: A Polygon and Polyline that are not spot-welded.

A user is able to pick up one of the Dot Items that make up either the rectangle or Polyline and spot weld it to another Dot. This will connect the two Items and results in something similar to the screenshot in Figure 6.8. The resulting Item is a Polyline as opposed to a Polygon as not all the Dot Items form a single enclosure.

Alternatively, once the user has picked up one of the Dot Items they are able to spot-weld to any place on a line produced by two Dot Items. When the user does is, Expeditee will insert a new Dot Item into the Polyline. Figure 6.9 shows a screenshot of what this might look like when using our example from Figure 6.7. When spot welding between two Dot Items, Expeditee does not merge the two Items. As a result, any Polygon remains filled in. This gives authors the ability to use split alternations in the position of weld points to produce different effects.

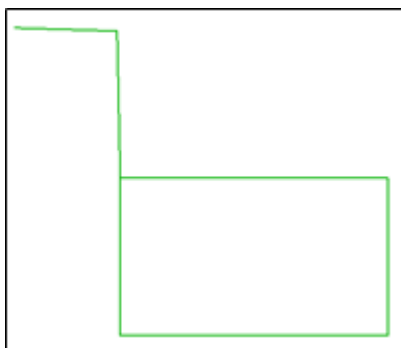


Figure 6.8: Spot-welding to an existing Dot Item.

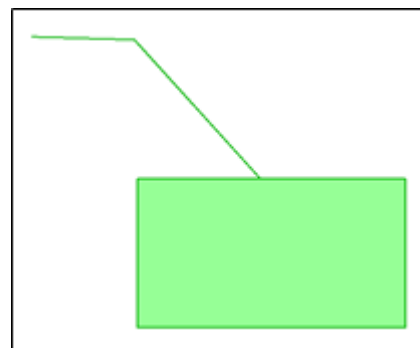


Figure 6.9: Spot-welding between existing Dot Items.

6.6 Summary: Expeditee In-Depth

In Section 4.3.3 performed a cursory review of Expeditee functionality and history, focusing on the aspects that contributed to our decision to use it as the base from which to develop

SplDER. In Chapter 6 we have performed a more in-depth analysis of Expeditee. This involved discussing its interface, expanding on the Frame and Linking system, and documenting some advanced features such as annotations, user scripting and spot-wielding. The terms we have introduced in the process of performing this analysis will be used throughout the remainder of the thesis.

Chapter 7

Usage and Implementation of SpIDER

In this chapter we document how programs are expressed using SpIDER along with novel aspects of its implementation. Integrating IDE functionality into a Spatial Hypermedia system, rather than modifying an existing IDE to include Spatial Hypermedia functionality, provides us with the best opportunity to design flexible interactivity that promotes the use of Spatial Memory. As explained in Chapter 5, Expeditee was chosen because of its reflective first class citizens, multimedia integration and fixed size spatial interface offered the best potential for maximising flexibility and use of Spatial Memory. The chapter also explains the rationale behind key design decisions, especially those consequent on the decision to use Expeditee as the base.

The usage of SpIDER is presented over the first three sections. Section 7.1 documents the way in which code is expressed in SpIDER. Section 7.2 then addresses how IDE functionality such as content assist is presented to the user. Section 7.3 documents how SpIDER provides support for running and debugging Java programs. Each of these sections explain how the aspect of SpIDER documented contributes to the final product. The chapter then transitions into documenting two important aspects of the implementation of SpIDER. Section 7.4 documents the implementation details of the flow walker algorithm. Section 7.5 details a collection of properties and functions referred to as the magnet system whose purpose is used to support a programmer in editing code. We conclude the chapter in Section 7.6 with broad details of the overall architecture of the SpIDER implementation.

7.1 Conveying Meaning in SpIDER

This section documents the aspects of SpIDER that allow programmers to lay their code out in two-dimensional space. Section 7.1.1 explains SpIDER's concept of lines as they exist in a

freely laid out two-dimensional space. Following this, Sections 7.1.2-7.1.4 each introduce a different technique that can be used to spatially lay out code. Section 7.1.5 presents an example demonstrating that these techniques—lines, boxes and arrows—can be used in a variety of ways to achieve the same result, thus providing the author with the freedom of expression required to customise their code layout to suit a wide range of intentions. Section 7.1.6 introduces a third dimension by documenting the Frame and Linking system. SpIDER expands Expeditee’s Frame and Linking system by augmenting the design of Frames to be suitable for authoring code; specific sections of the Frame are reserved for standard Expeditee Items that are not interpreted as code. Each section uses an example to explain its point. The ordering of the sections is such that examples begin simply then progress to include more sophisticated functionality.

Behind the scenes, the flow walker algorithm provides the functionality to support two-dimensional expression of code. The algorithm produces a one-dimensional text stream, suitable for presentation to the compiler, i.e., it serialises the code. The implementation of this algorithm is covered in Section 7.4.

7.1.1 Inferring Lines

To deliver on the envisioned way of writing code in a free-form 2D layout, two aspects of Expeditee’s core functionality needed to be carefully considered: Expeditee’s ability to **theme** Text Items and the standard **spatial ordering** provided by Expeditee.

- **Theming.** Syntax highlighting and error reporting require that specific tokens in a Text Item be coloured. While an Expeditee Text Item can be themed by adjusting its properties, it is not possible to theme a specific set of characters within a Text Item. In other words, if a Text Item contained a token that needs to be highlighted as a reserved keyword, then the entire Text Item must have its colour specified; not just the characters relating specifically to the keyword token.

- **Spatial Ordering.** Standard to Expeditee, when content is serialised through an action—such as when exporting to PDF or a text file—Text Items are ordered first by their y-coordinate, and then, when a conflict occurs, by their x-coordinate. This technique results in an issue that is infrequently a problem in standard Expeditee: two Text Items can appear to have the same y-coordinate, when in reality one is a single pixel higher, potentially resulting in unexpected ordering.

Additional to these considerations, the requirements for interacting with **Eclipse JDT** (the library providing IDE Functionality) needs to be considered: IDE functionality such as content assist must be able to relate a Text Item on screen with the tokens it represents in the serialised Java file. If each Text Item can contain multiple tokens, then additional computation would be required to resolve this relation.

Another way of looking at this issue is by comparing the Fundamental Elements of Expeditee and code. As specified in Section 4.3.3, Expeditee's Fundamental Element is the Text Item. The Fundamental Element of code is the token. Text Items are a collection of characters, including whitespace characters, whereas a token is a collection of non-whitespace characters (with the exception of string literals). The theming and interactions with Eclipse JDT can be simplified by limiting the use of Expeditee Text Items in SpIDER, to a collection of non-whitespace characters. We get the best match by always using one Expeditee Text Item to represent each token.

Adopting the solution of using one Text Item per token helps deliver on the envisioned spatial freedom desired for SpIDER. Figure 7.1 shows an example of spatial freedom gained by limiting Text Items to one token each; not only are some tokens coloured, but one token is raised above the others in its line. Out of context it is impossible to say why the author has chosen to do this—it is possible that it was a mistake—but it does demonstrate utility not possible if all tokens on the first line were contained within a single Text Item. Instead of raising the position of a token, the programmer could have chosen to adjust the tokens size or font.

```
int quantity = Integer.parseInt ( args [ 0 ] ) ;  
System.out.println ( "Producing a list of size: " + quantity ) ;
```

Figure 7.1: Two lines of Java code whose serialisation requires additional processing.

While the issue of spatial ordering in standard Expeditee was an infrequent issue, by limiting the size of Text Items to an individual token, the number of Text Items has been substantially increased, thus increasing the chance that standard spatial ordering causing a problem has also increased. Notably, the ordering of the tokens in Figure 7.1, under standard Expeditee ordering, would place the *Integer* token prior to the tokens of the statement it belongs to. An algorithm—named the flow walker—has been developed to serialise spatially laid out code under the limitation of one token per Text Item. The implementation of this algorithm is documented in Section 7.4. Prior to this, the remaining content of this section documents additional spatial layout that the flow walker provides.

7.1.2 Boxing

Boxes can be drawn around code to separate one set of statements from another. SpIDER will interpret content within a box to be on a separate logical group from the content outside of the box. A programmer may use boxes to help visually communicate something about the content within. For example, consider Figure 7.2. The programmer has chosen to enclose two loops in a box. This has been done to show that they are functionally important compared to other statements in the function.

This example, and subsequent examples from the following sections, contains linked Text Items and annotations. Section 7.1.6 will explain how SpIDER uses these.

```

private static int [ ] merge ( int [ ] left , int [ ] right ) {
    final int [ ] merged = new int [ left. length + right. length ] ;

    int i = 0 , o = 0 , p = 0 ;
    ◦//merge in at least one list completely @merge in remaining items
    for ( ; i < left. length ; i ++ ) merged [ p ++ ] = left [ i ] ;
    for ( ; o < right. length ; o ++ ) merged [ p ++ ] = right [ o ] ;

    return merged ;
}

```

Figure 7.2: Boxing used to emphasise a set of statements.

A programmer can also use boxes to cause SpIDER to interpret code in a specific way; an example of this can be seen in Figure 7.3. Two functions: *getRelativeX* and *getRelativeY* are displayed on the same Frame, side-by-side to highlight their similarity. Boxes have been used to separate them, forcing the SpIDER to process each separately. If the boxes were not present then SpIDER would form lines spanning the screen. This would interweave the functions together, producing an incorrect serialisation.

<pre> private int getRelativeX (int square , int orientation) { switch (orientation % 4) { case 0 : return shapeX [square] ; case 1 : return - shapeY [square] ; case 2 : return - shapeX [square] ; case 3 : return shapeY [square] ; default : return 0 ; } } </pre>	<pre> private int getRelativeY (int square , int orientation) { switch (orientation % 4) { case 0 : return shapeY [square] ; case 1 : return shapeX [square] ; case 2 : return - shapeY [square] ; case 3 : return - shapeX [square] ; default : return 0 ; } } </pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 7.3: Boxing used to separate one function from the other.

Boxes can be placed within other boxes. Figure 7.4 shows a class and function definition. Inside the *start* function, several pieces of code are hidden behind linked Text Items. Each of these linked Text Items is placed in its own green box, each positioned within a blue box.

By placing boxes within boxes, the programmer has been able to communicate that there exists three distinct but related code fragments. Section 7.1.6 covers linking in more detail.

```
package GUI ;
◦ imports
public class AdderFX extends Application {
    private static int currentValue ;
    public static void main ( final String [ ] args ) {    ◦ //start program    }

    public void start ( final Stage primaryStage ) {
        ◦ //Create form buttons and set title
        ◦ //Print Current Value    ◦ //Add    ◦ //Subtract
        ◦ //Pack and show
    }
}
```

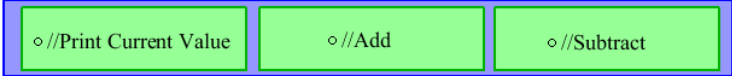


Figure 7.4: Boxes contained within boxes.

7.1.3 Out of Flow

By combining arrows and boxes a programmer is able to position code *out of flow*. This introduces a new form of abstraction to the code and allows for more horizontal space to be used. Figure 7.5 shows an example of using arrows and boxes to reposition some code out of the general flow. In this example, this repositioning has had the effect of emphasising the base case and recursive call from the rest of the content in the *sort* function. When SpIDER encounters an arrow, it finds the box that the arrowhead is in, processes it, and includes the result back at the tail end of the arrow.

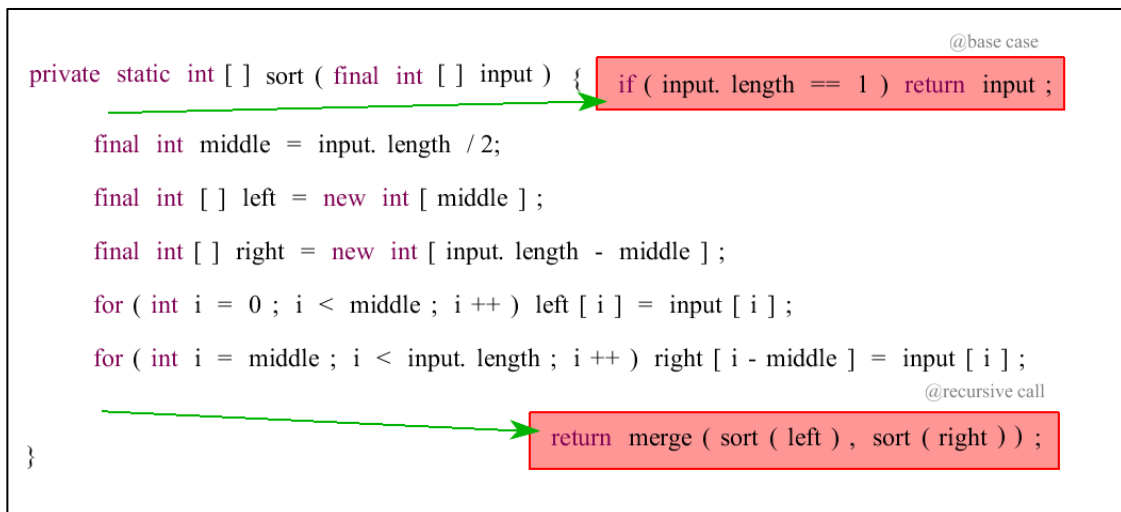


Figure 7.5: Base and recursive case of a function taken out of flow.

Precise positioning of the tail of an arrow can be used to improve the visibility or ease of modification of specific tokens. Figure 7.6 shows a toy example with two functions to illustrate the point. The *addPrint* function which adds and prints the integer parameters it receives, and the *main* function that call the *addPrint* function. The parameters passed to the *addPrint* function have been placed out of flow. This results in the parameter values being emphasised and easily editable.

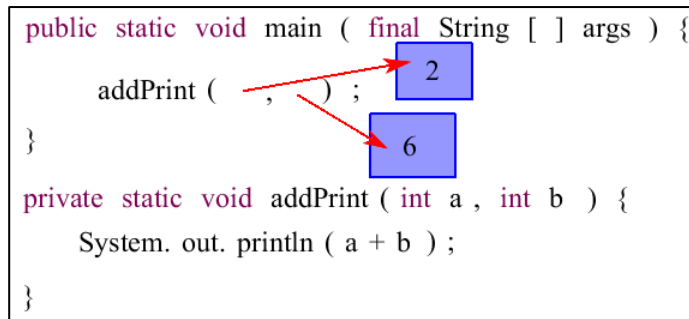


Figure 7.6: Demonstrating the positioning of arrow tails.

7.1.4 Out of Flow Chaining

The tail of an arrow can be placed inside a box. This allows the programmer to create a series of arrows and boxes that produce a chain. A programmer may use this to show the order a process executes in, or alternatively, visualise the structure of a code fragment.

Figure 7.7 shows an example of using chaining to express an anonymous Java class. This has had the effect of separating the code fragment into several pieces; exaggerating the content in the red box which the programmer deems more important than the other code on the Frame.

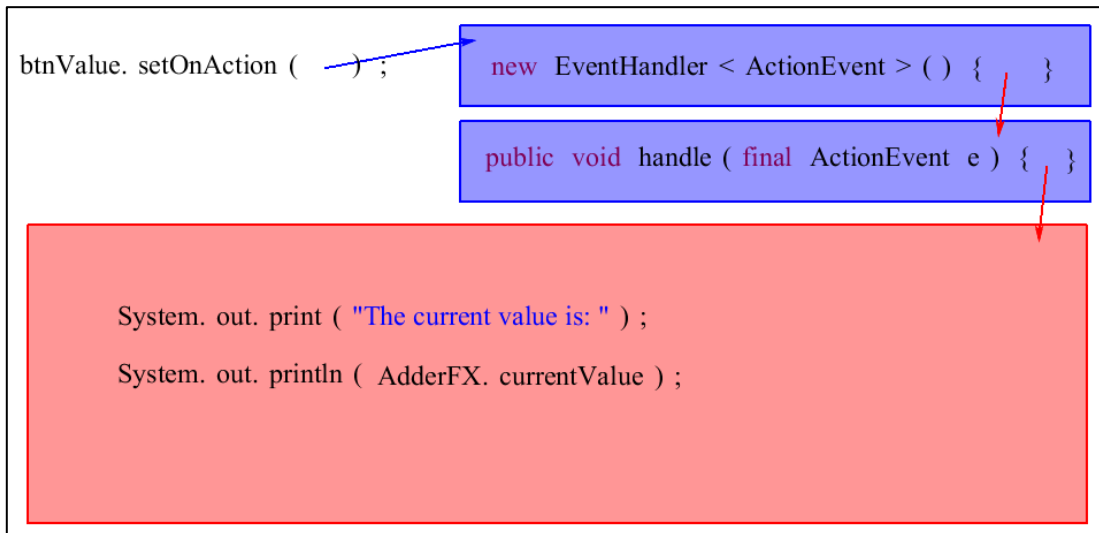


Figure 7.7: Deconstructing an anonymous class with Chaining.

In the example of chaining shown in Figure 7.8 we see a way of using annotations (@) to control the output the flow walker produces. The Java class *Random* (located at *java.util.Random*) optionally takes a parameter to its constructor; when provided it is used as a seed for generating random numbers. By moving the end point of the arrow backwards

and forwards between the red and blue box, and toggling the annotation on the box (an annotated box is ignored when producing serialised code) we can optionally include the parameter. This can be used by the programmer for testing purposes.

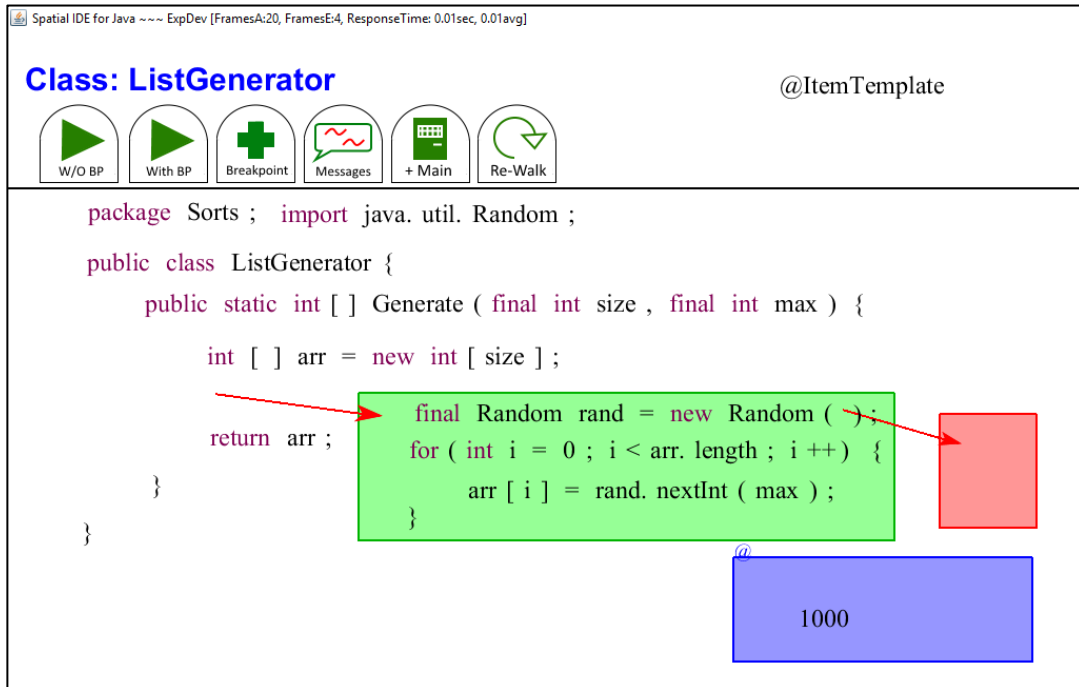


Figure 7.8: Optional Chaining.

7.1.5 A Multitude of Ways

Having seen a way that a programmer might use chained arrows and boxes in combination with annotations to optionally include a parameter to the constructor of *Random* we will now demonstrate that there are many other ways to achieve the same thing. The first to notice is that the red box is optional; the programmer has included it to reinforce the visual of not including a parameter. Figure 7.9 shows you the red box can be removed: while the annotation on the blue box is present it is treated as an empty box, and therefore no parameter is present. Toggling the annotation controls whether or not SpIDER includes the *1000* in the serialisation.

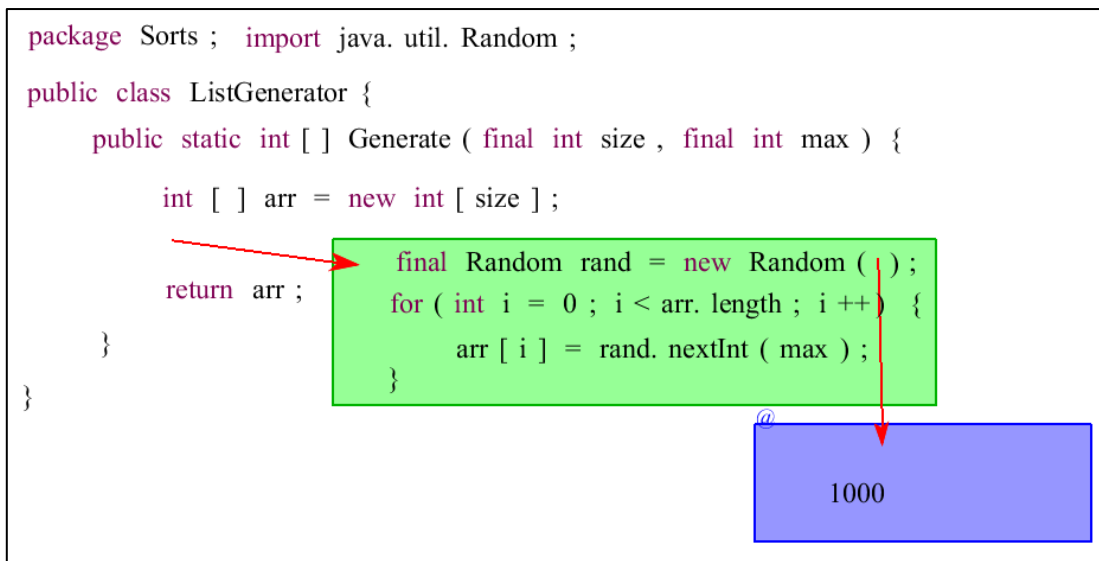


Figure 7.9: Optional Chaining; no red box.

Figure 7.10 shows that the annotation can be on the *1000* rather than the box. When applied to a specific item, annotations will hide only that item from the serialisation process. In this example, the position of the annotation, whether it be on the box or the Text Item directly has no effect on the serialisation. However, if the programmer were to include a more complex statement in the blue box, then the positioning of their annotation gives the programmer fine grained control over the compiled product.

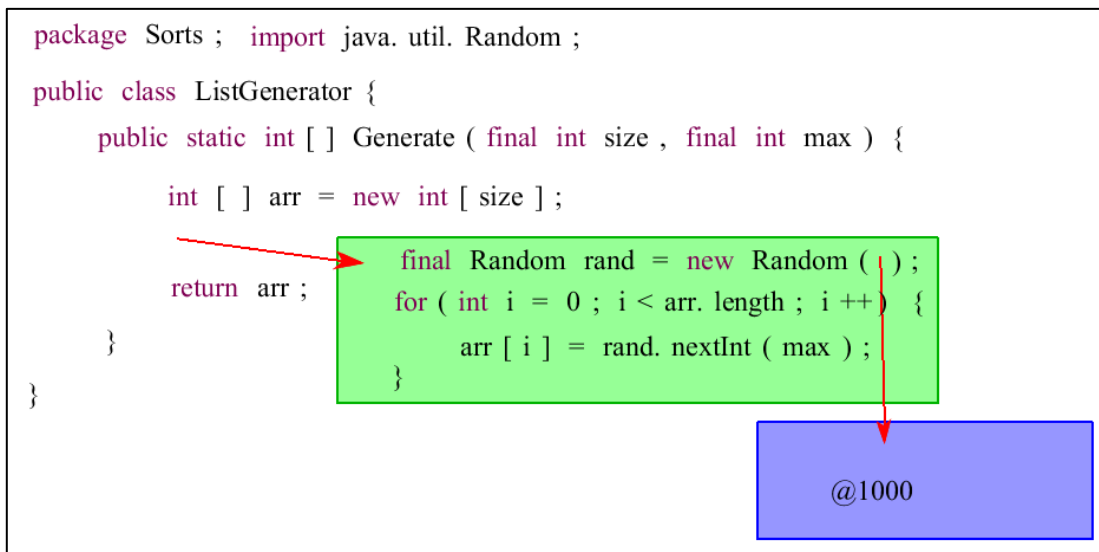


Figure 7.10: Optional Chaining; no red box; annotation on item.

In this hypothetical situation the programmer was wanting to have an optional parameter for testing purposes. This allowed them to have the same set of random numbers being generated each time they run their application. Instead of removing the red box there is another option. Figure 7.11 once again includes the red box, however now there is some code in there for the serialisation process to find. By optionally including the code: *System.nanoTime()* the programmer can produce behaviour similar to using the empty constructor.

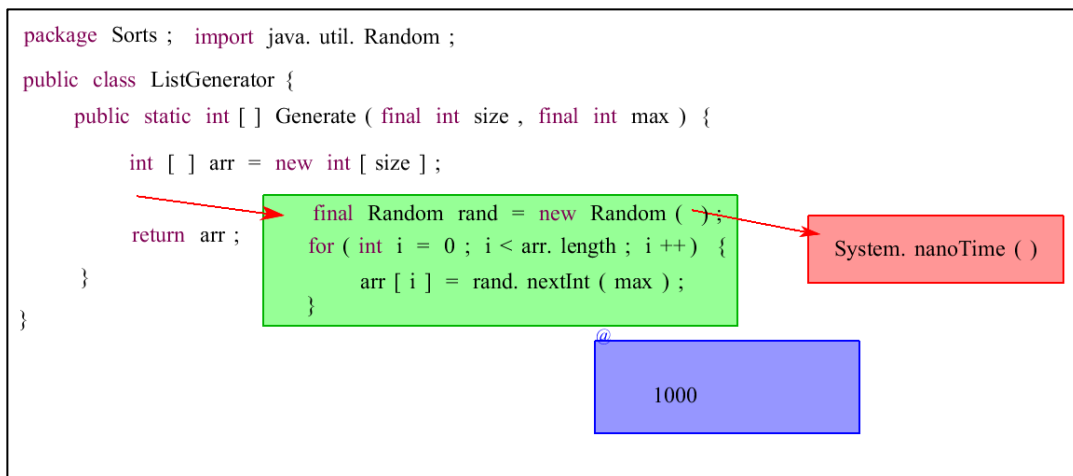


Figure 7.11: Optional Chaining; *System.nanoTime()*.

We have demonstrated four different ways of achieving the same hypothetical task, and programmers are likely to find more. The example we have given is simple; showing that there is wide flexibility even in small cases.

7.1.6 Frames and Linking

Section 6.3 detailed how Expeditee's Frame and Linking system functions. To illustrate its use in SpIDER, Figure 7.12 repeats a previously featured application; a simple calculator program. The package declaration, class declaration, a single field and the skeleton of the *main* and *start* functions are directly visible on screen. The menu area spanning the top of the screen is now visible, we refer to this as the non-code area. Implementation detail is hidden behind linked Text Items. An annotation is used to communicate some supplementary detail about a set of code fragments.

Class: AdderFX

W/O BP

With BP

Breakpoint

Messages

+ Main

Re-Walk

0

10

100

1000

@ItemTemplate

```

package GUI ;
◦imports
public class AdderFX extends Application {
    private static int currentValue ;
    public static void main ( final String [ ] args ) { ◦//start program }

    public void start ( final Stage primaryStage ) {
        ◦//Create form buttons and set title
        ◦//Print Current Value ◦//Add ◦//Subtract
        ◦//Pack and show
    }
}

```

@Actions

◦//Print Current Value

◦//Add

◦//Subtract

Figure 7.12: The skeleton of a simple calculator program, complete with linked Text Items, the non-code area and annotations.

Linked Text Items. When the SpIDER serialisation process encounters a linked Item it follows that link, processes the content on the resulting Frame and then inserts the result into the serialised file in place of the linked Item. This process is recursive, if a Frame reached by following a link, also contains a link, then that link is also followed. Prior to serialisation, SpIDER performs a sanity check, ensuring that no series of linked Items and Frames produce a loop. If a loop is detected, serialisation will not occur and the programmer will be notified.

The content of a linked Item is not included in the resulting serialisation. This allows a programmer to use the linked Item as a form of documentation.

Non-code Area. Every code Frame has a dividing line that separates the coding area below from a non-code area above. Controls for creating and running Java projects are placed in the non-code area. Any authored content positioned above the dividing line is not considered code. This means that it is not subject to syntax highlighting, tokenization, inclusion in the serialised Java file or any other IDE functionality. Essentially, it is treated as

a set of Expeditee Items rather than a set of SpIDER Items. Furthermore, any Item in the non-code area that is converted to a link leads to a non-code Frame.

Annotations. Annotated Items are treated as non-code Items regardless of their position on the screen. This means that: they can be used to provide supplementary documentation that does not appear in the serialised Java file; converting one to a link will cause the system to treat the destination as a non-code Frame. As seen in Section 7.1.5, an annotation can also be applied to a box, causing all Items inside the box to be treated as non-code Items.

Non-code Frames. A non-code Frame is created when a non-code Item, or Item in a non-code area, is used to create a link. A non-code Frame is simply a normal Expeditee Frame. Links created on non-code Frames lead to non-code Frames. This allows a programmer to create a set of Frames specifically for documentation.

7.2 Authoring IDE Functionality

This section documents the design of the SpIDER's IDE functionality. Section 7.2.1 discusses syntax highlighting and provides a code example showing both keyword highlighting and string highlighting. Section 7.2.2 then documents SpIDER's warning and error system. Section 7.2.3 shows SpIDER's version of content assist. In contrast to traditional IDEs where content assist is ephemeral, SpIDER provides the user with a set of Text Items. These Text Items are placed in a box that is marked as an annotation (@) so that the flow walker does attempt to include them in the serialised output. We refer to such Text Items as non-code Text Items. Section 7.2.4 documents the process of creating Java projects and the supporting state system that is used to display an appropriate set of controls based on the Frame the programmer is currently viewing.

7.2.1 Syntax Highlighting

It is natural to consider syntax highlighting a tool for reducing the intellectual effort of understanding code [23, 28]. Figure 7.13 reproduces a screenshot featured earlier when discussing out of flow chaining, which also demonstrates syntax highlighting. SpIDER highlights content in the same fashion as Eclipse; strings are coloured blue, keywords are coloured red.

SpIDER has an additional challenge in supporting syntax highlighting as compared to traditional IDEs, other content on the screen can be coloured. Consider the colour of the

boxes in Figure 7.13. Both keywords and the bottom box are coloured red. Furthermore, both strings and the top two boxes are coloured blue. Whilst the shades differ between box colour and token colour, the programmer has chosen to use a red box to contain the code that includes a string and a blue box to contain code with several keywords. If the programmer wished, they could adjust the boxes to be a wide range of other colours. The *F3* key can be used to cycle through a selection of colours on a colour wheel. Using SpIDER's *settings* Frameset, the programmer is able to adjust what colours appear on the colour wheel. Furthermore, property injection can be used to set the colour of the box to any valid RGB mix. Choice of colour for boxes is left to the whim and discretion of the programmer; avoiding colours that make Text difficult to see due to contrast is assisted by the default set of colours accessed through the *F3* key.

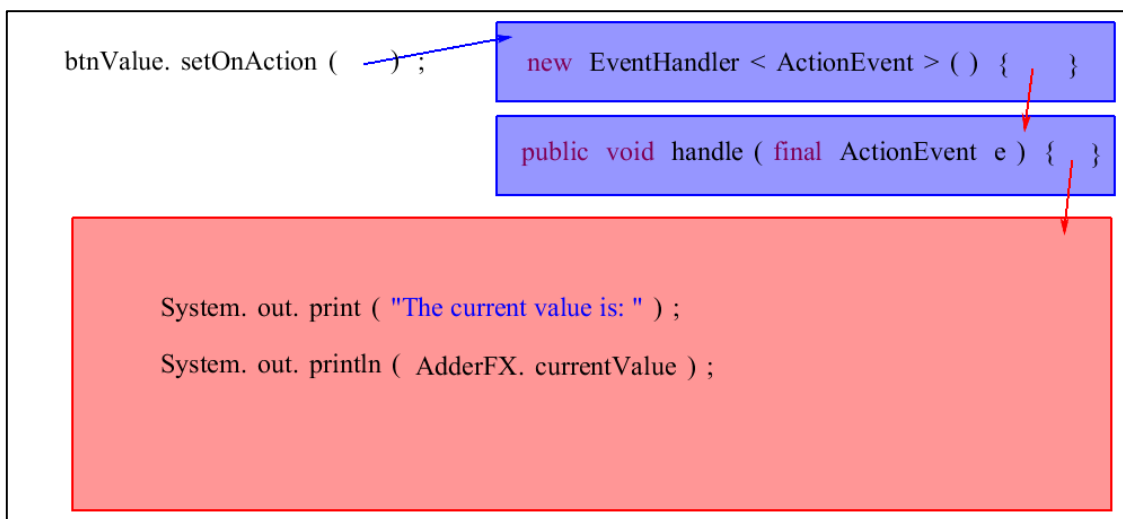


Figure 7.13: Showing keyword and string syntax highlighting.

7.2.2 Warnings and Errors

Another form of content highlighting comes in the form of a warning and error system. Constant compilation allows for timely user feedback concerning the state of the code being developed. Figure 7.14 shows an example of a simple syntax error in SpIDER. In this case, the error is caused by the incorrect capitalisation of the type *String*.

```

package spiderdefault ;

public class Test {

    public static void main ( final string [ ] args ) {
        }
    }
}

```

Figure 7.14: Syntax error cause by incorrect capitalisation.

Figure 7.15 shows the declaration of the string *Name*. Note that the declaration capitalises the first letter. Figure 7.16 shows the Frame reached by following the linked Text Item with content ‘Say Hi’. It is on this Frame that the variable *Name* is meant to be used. Unfortunately, the programmer has written the variable entirely in lower case. This is an error. This example demonstrates an additional challenge SpIDER must overcome. Whilst the two statements are logically next to each other, they are placed on separate Frames. Each Frame is not self-contained. As with a traditional IDE, the logic and correctness of a specific fragment of code is reliant on any code that uses it. However, compared to a traditional IDE, less context is likely to be visible at any given time.

```

package spiderdefault ;

public class Test {

    public static void main ( final String [ ] args ) {
        String Name = "Bryce" ;

        ◦ Say Hi
    }
}

```

Figure 7.15: Declaring a variable to be used on a child Frame.

```

System. out. println ( "Hello " + name ) ;

```

Figure 7.16: Erroneous attempt at referencing the declared variable.

SpIDER solves this problem in two parts. The first part of the solution is a button provided in the non-code area that when pressed will list all the problems that the current project has. Figure 7.17 shows an example of this button being used. Clicking on the warning or error message in the Message Bay navigates the programmer to the Frame that the error occurs on.

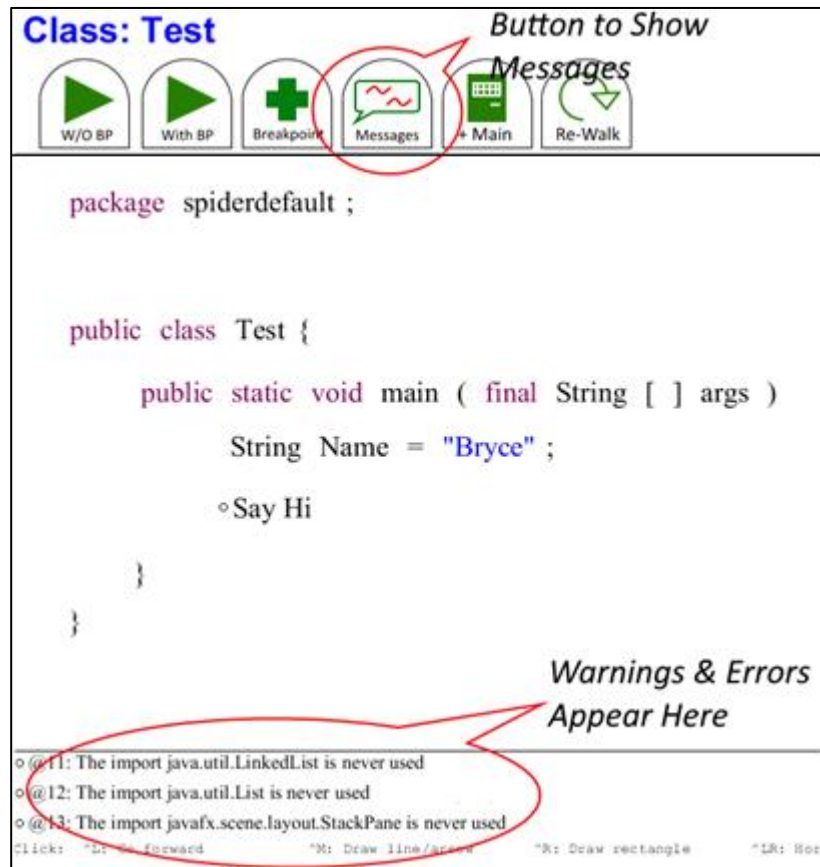


Figure 7.17: Displaying problems in the current project.

The second part of the solution is to highlight linked Text Items either yellow or red if following them will lead to a warning or error respectively. This would allow a programmer to begin at a high-point in a hierarchy of Frames that contain a warning or error and follow a trail or highlighted links to narrow down the cause of the problem. The second part of the solution is not currently implemented (as of 2018), however, Figure 7.18 shows a mock-up of how this might look if implemented using a modified version of Figure 7.15.

```

package spiderdefault ;

public class Test {

    public static void main ( final String [ ] args ) {

        String Name = "Bryce" ;

        Say Hi

    }

}

```

Figure 7.18: A mock-up of link highlighting.

7.2.3 Content Assist

In traditional IDEs such as Eclipse and Visual Studio, content assist is ephemeral and always reflects the latest actions taken by the programmer. For example, having typed *'this.'* will show the programmer all members from the current class. Typing another letter refines the content assist results—*'this.a'* will limit the results to those starting with the letter *'a'*. This refinement is consistently happening as the user types, without the need for the user to request an update. When the programmer selects an option from the currently open content assist, their selection is inserted and the results from their request for content assist disappears.

In contrast, content assist in SpIDER produces a set of first class citizens and is tied to a specific instance of the request for assistance. A content assist request where the programmer has typed *'this.'* produces one set of results, constructed out of Expeditee Items that have the full range of manipulation of any other Expeditee Item. If the user refines the text to be *'this.a'* then a new request for content assist must be made. A content assist request is made by using the keyboard command CTRL + Space with the results initially appearing attached to the programmer's cursor.

Figure 7.19 shows the results of a request for content assist in SpIDER. The programmer has specified a function starting with *'next'* on the object *'rand'* which is of type *java.util.Random*. An annotated box has been created and populated. This means that, the content within the box is not considered code until it is moved outside of the box. The content is a set of Text Items arranged into lines. Each line represents a valid completion

suggestion. When a suggestion is overloaded, such as *nextInt*, an additional linked Text Item, leading to the alternatives, is created. As the result of the content assist being created out of Expeditee Items, the programmer is able to manipulate them as to suit their current task. They may:

- Decide to place the suggestions on the far right of the page (or place them on another page and keep a link to them) so that they are out of the way but can be reused.
- Manually delete or filter options as they decide.
- Spatially rearrange and layout the options as they would code.
- Simply delete the whole box.

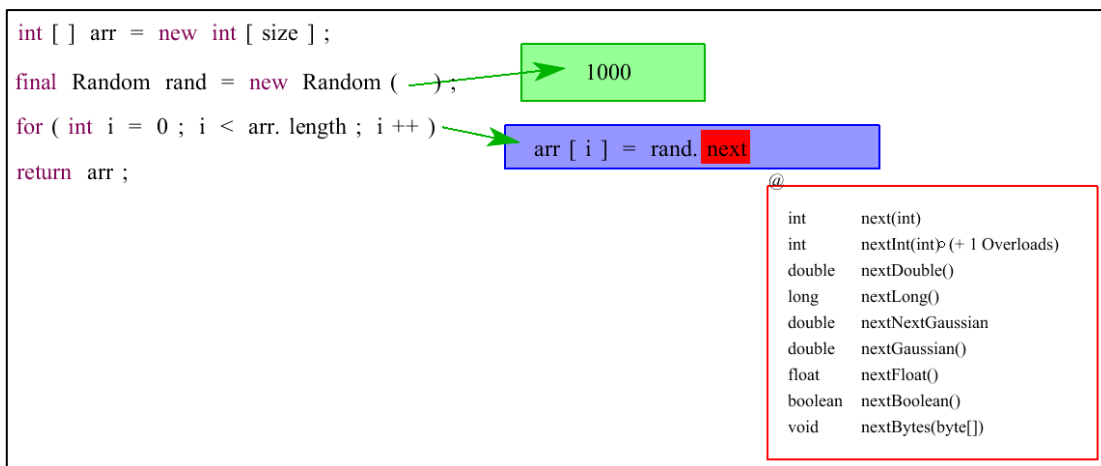


Figure 7.19: Example of Content Assist.

7.2.4 Java Projects and SpIDER State

SpIDER arranges projects into a hierarchy of Frames. When starting SpIDER, after selecting the workspace the programmer wishes to work with, the programmer is presented with a Frame listing all the existing Java projects in that workspace. This Frame also features two buttons that execute actions to create new or import existing Java projects. We refer to this Frame as the Project List Frame. Figure 7.20 shows a Project List Frame in SpIDER for the workspace *Ch8Projects*.

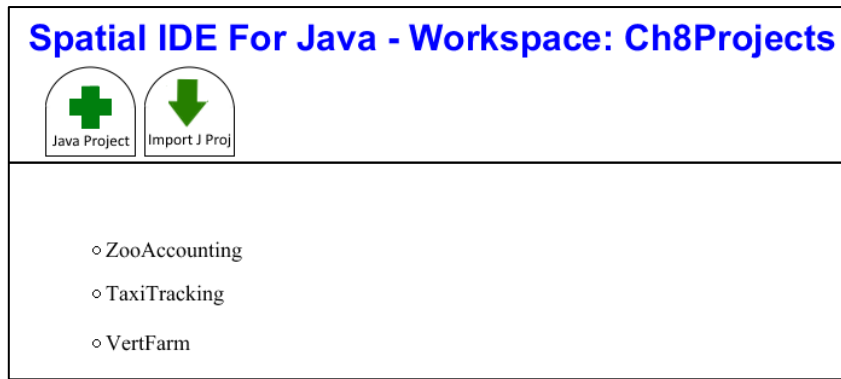


Figure 7.20: List of Java projects shown on SpIDER's initial Frame.

In order to demonstrate how Java projects are created, we will show the step-by-step process that results in the creation of a subset of the project artefacts in the application: *ZooAccounting*. The process begins with the programmer creating a Text Item to represent the name of the project. This Text Item is then picked up. With the Text Item attached to the cursor, the programmer then clicks on the *Java Project* button. This results in the Text Item becoming linked to the generated Frame. Following this link takes the programmer inside the project.

Figure 7.21 shows the first Frame inside the *ZooAccounting* project. This Frame is referred to as a Project Frame. On the Project Frame a list of top level packages and classes are listed. In this example, the programmer has arranged packages and classes in separate boxes. The buttons supplied to the programmer have changed. Now that we are inside a project, the programmer is supplied with buttons to create packages and classes.

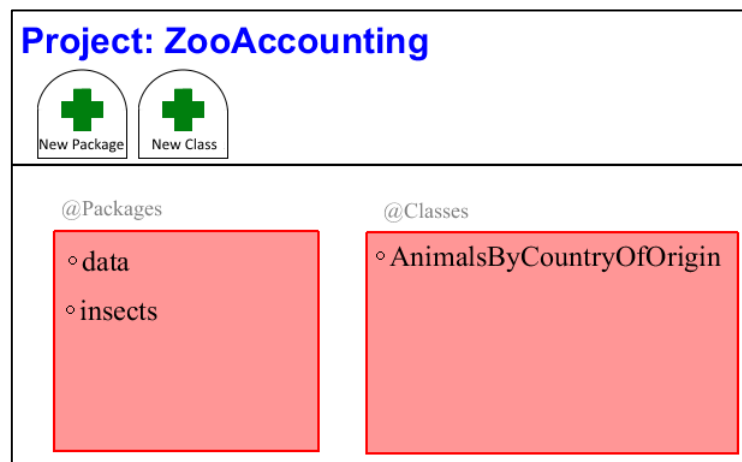


Figure 7.21: List of packages and classes in the ZooAccounting Project. An example of a Project Frame.

Using the same button activation technique as was used to create the project, the programmer has created two packages and a class. When the programmer clicks on the *data* package, they are taken inside the package to the Frame seen in Figure 7.22, we refer to this as a Package Frame. When on a Package Frame the programmer retains the controls allowing them to create packages—now sub-packages—and classes.

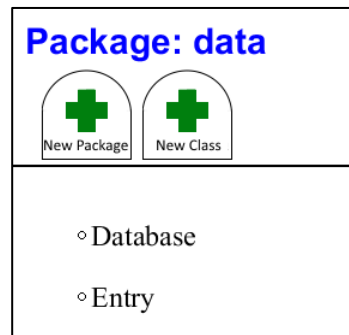


Figure 7.22: List of classes inside the *data* package. An example of a Package Frame.

If the programmer instead clicks on the *AnimalsByCountryOfOrigin* class, they are taken inside the class to the Frame seen in Figure 7.23, we refer to this as a Class Frame. When on a Class Frame, the controls change to a set of buttons useful for writing and executing code.

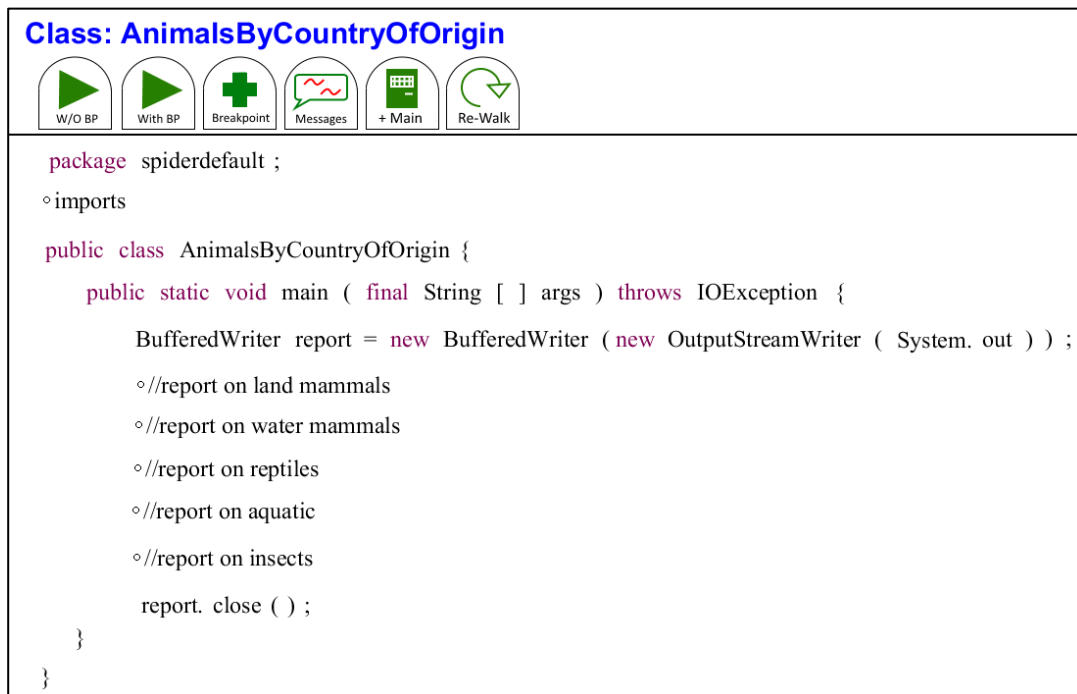


Figure 7.23: The top Frame of the *AnimalsByCountryOfOrigin* class. An example of a Class Frame.

A state machine is used to determine which set of buttons should be shown to the programmer at any given time. As the programmer moved from a Project List Frame, to a Project Frame, Package Frame and Class Frame, the state machine caused the controls seen by the programmer to update. As a programmer is developing their project, the state machine will continue to update the controls that the programmer can see.

- **When Editing Code.** The controls available, shown in Figure 7.24, give the programmer the ability to: execute the program; insert a breakpoint; print the current warnings and errors associated with the project to the Message Bay; insert a main method or request the current project be processed by SpIDER.
- **When Running Code.** The controls available, shown in Figure 7.25, give the programmer the ability to: jump to the Class Frame of the class responsible for executing the currently running project; halt the currently running project and provide 'standard input'.
- **When Debugging Code.** The controls available, shown in Figure 7.26, give the programmer the ability to: either jump to the appropriate Class Frame or the current breakpoint that the project has halted on; resume the debug session; halt the debug session; provide 'standard input'; request a report of the current state of the debug session; perform a step in the debugger.



Figure 7.24: Controls available while editing.



Figure 7.25: Controls available while running.



Figure 7.26: Controls available while debugging.

7.3 Running Java Programs

This section documents the design of SpIDER's project execution functionality. A single example program, modified at each stage, is used to explain this functionality. The code for the program that we will be using for the example can be seen in Figure 7.27. Section 7.3.1 will explain how code can be executed without breakpoints while Section 7.3.2 separately handles executing with breakpoints. While discussing the process of debugging in SpIDER, we will cover the way in which programmers can request information on the state of the application while it is suspended. As for content assist, the results from a request for assistance is constructed out of first class citizens.

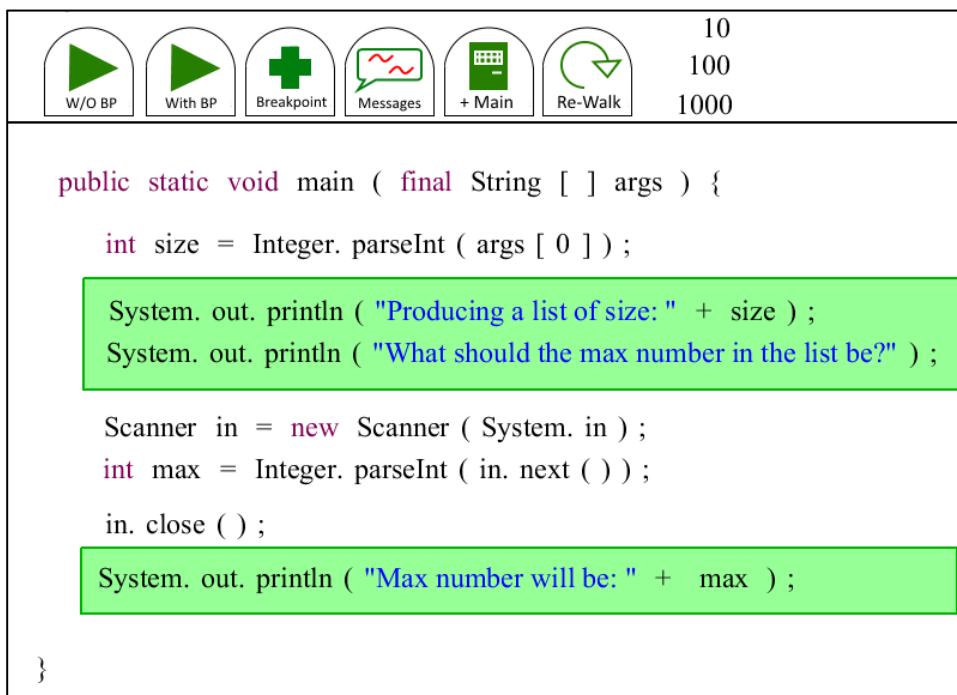


Figure 7.27: An application that uses program arguments and standard input to execute.

7.3.1 Executing without Breakpoints

The left most button seen in Figure 7.27 can be used to run the program without breakpoints. When clicked, SpIDER will locate the main method associated with the project being edited on the current Frame and execute it. In our example above, doing so will cause an *ArrayOutOfBoundsException* to be thrown. This is expected as the first statement to be executed attempts to use a program argument.

Program Arguments. In order to successfully start executing the application, the programmer must supply program arguments. Notice that, in Figure 7.27, the programmer has created three Text Items with content '10', '100' and '1000'. These have been created in the non-code area so that they are not considered code. These Text Items can be used as program arguments. In order to execute the application whilst providing some user input, the programmer picks up one of these Text Items and clicks on the run button with it attached to the cursor.

Figure 7.28 shows the application being run after the programmer has supplied '1000' as an argument. The Message Bay at the bottom of the screen is displaying the output of the program so far. The program argument has been used to construct the string for the first print statement and the second print statement has executed in order to prompt the user to provide some data on 'standard in'.

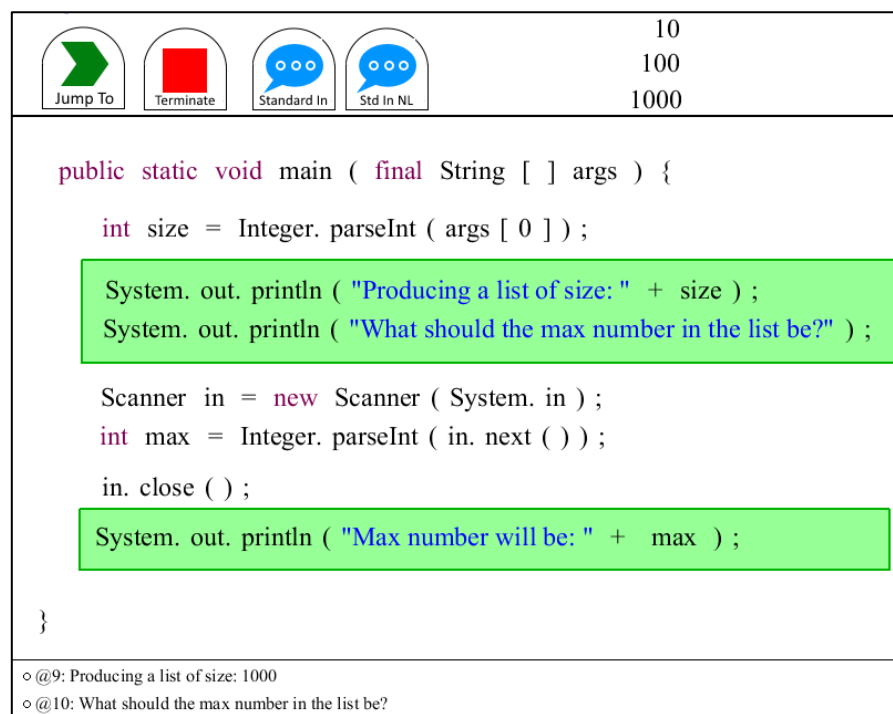


Figure 7.28: An application waiting for user input.

Standard Input. When the application started to run, the state machine caused the controls to change to those seen in Figure 7.25. The third and fourth buttons can be used to send data to standard in. Using the same technique that was used to send arguments to the program, the programmer is able to send content to standard in. The third button will echo

the exact content that the programmer provides, whereas the fourth appends a new line character to the end.

Figure 7.29 shows the state of SpIDER's Message Bay after the application has successfully executed. The final print statement has executed showing that the programmer used the Text Item with content '100' to send to standard in. A red message has appeared in the Message Bay signalling that the program has terminated. The controls have reverted back to the standard controls for editing code.

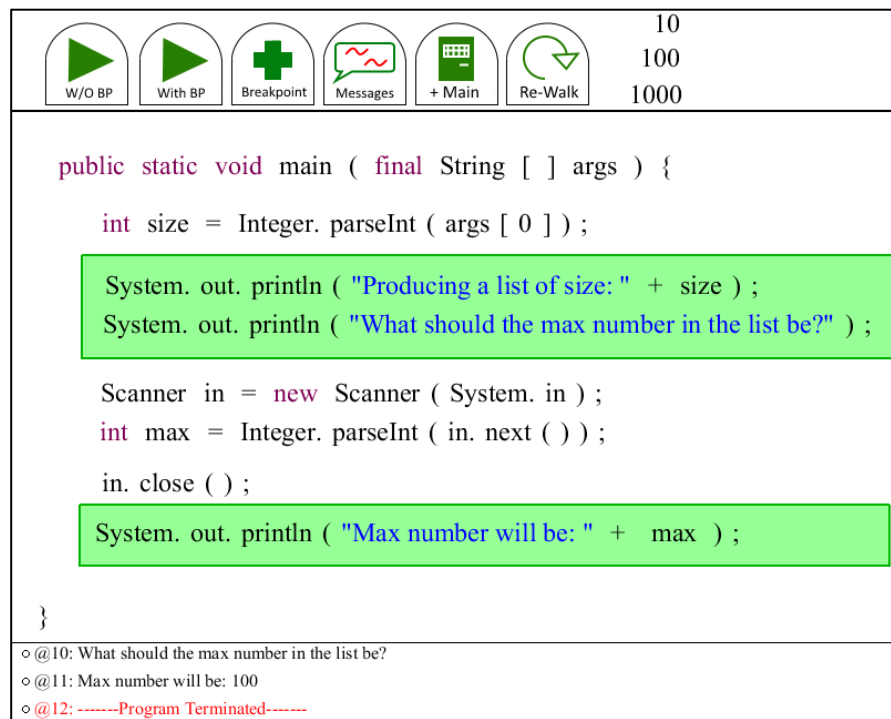


Figure 7.29: An application successfully executed.

7.3.2 Executing with Breakpoints

The second button from the left in Figure 7.30 can be used to run the program with breakpoints enabled. Arguments to the program can be provided using the technique previously explained when using the button to run the program without breakpoints. Two breakpoints have been set.

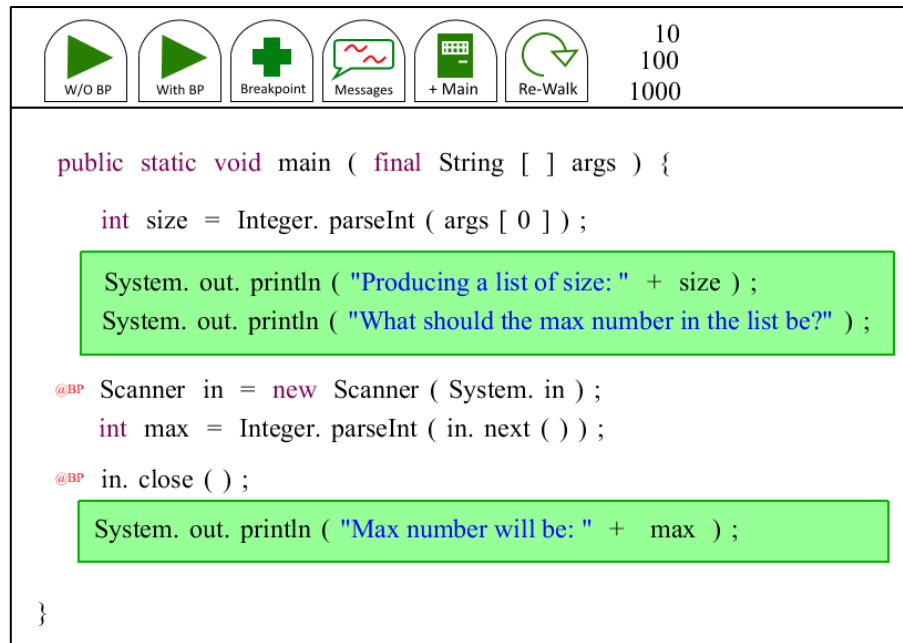


Figure 7.30: An application with two breakpoints set.

Breakpoints. Clicking the third button from the left (controls in Figure 7.24) will attach a breakpoint to the cursor. SpIDER represents a breakpoint as an annotation with content '@BP' using a red font. Placing the breakpoint annotation at the beginning of a statement will attach a breakpoint to that statement. When the program is executed with breakpoints enabled, SpIDER will suspend the application at the breakpoint and provide a different set of controls. Figure 7.31 shows SpIDER in this suspended state, waiting for the programmer to issue an instruction.

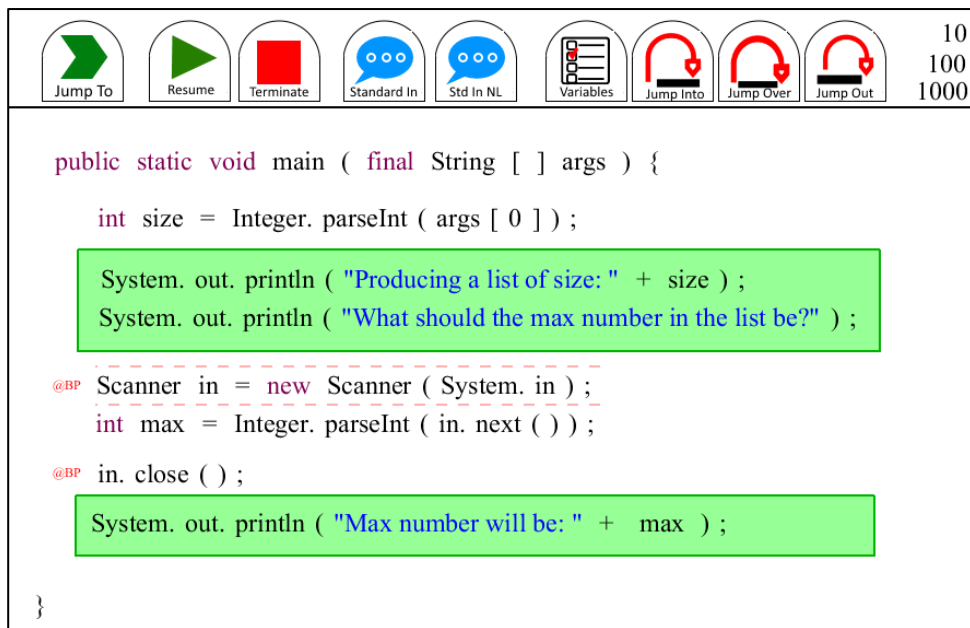


Figure 7.31: An application suspended at the breakpoint, awaiting instructions.

Debug Stepping. While suspended, the programmer is able to issue resume or step instructions. Two dashed lines show the next statement to be executed if the program was to proceed. Clicking the resume button will cause the application to continue execution until the next breakpoint. Figure 7.32 shows the state of the application after pressing the resume button. Because this application requires standard input, the second breakpoint has not yet been hit.

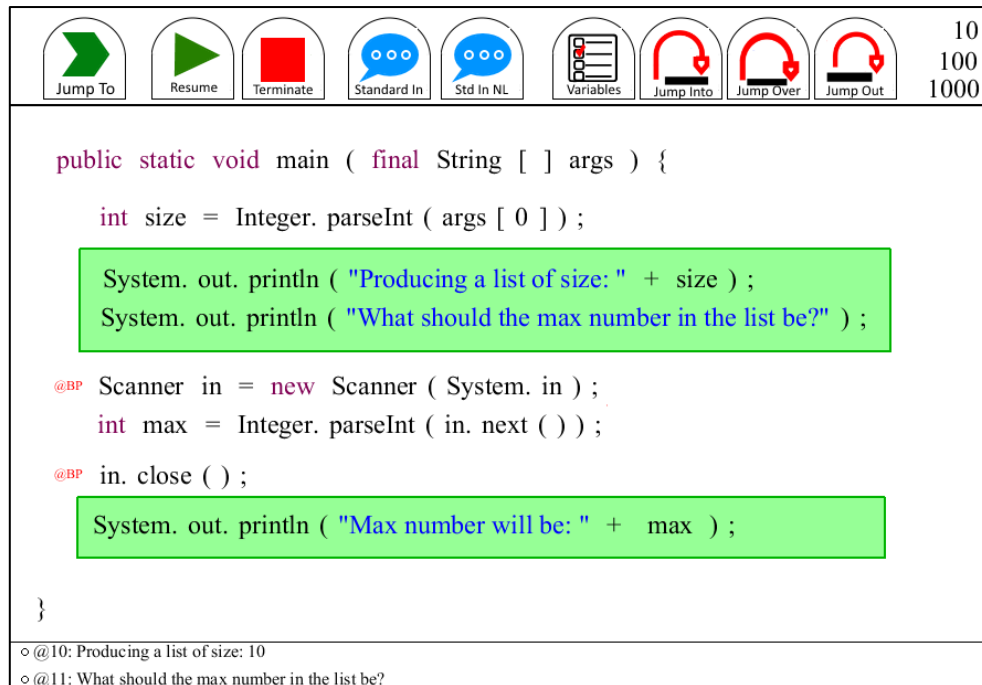


Figure 7.32: An application waiting for standard input between two breakpoints.

Once the programmer has supplied input, the second breakpoint is hit. This can be seen in Figure 7.33. The red dashed lines have been removed from the line associated with the first breakpoint and have been placed around the line associated with the second breakpoint.

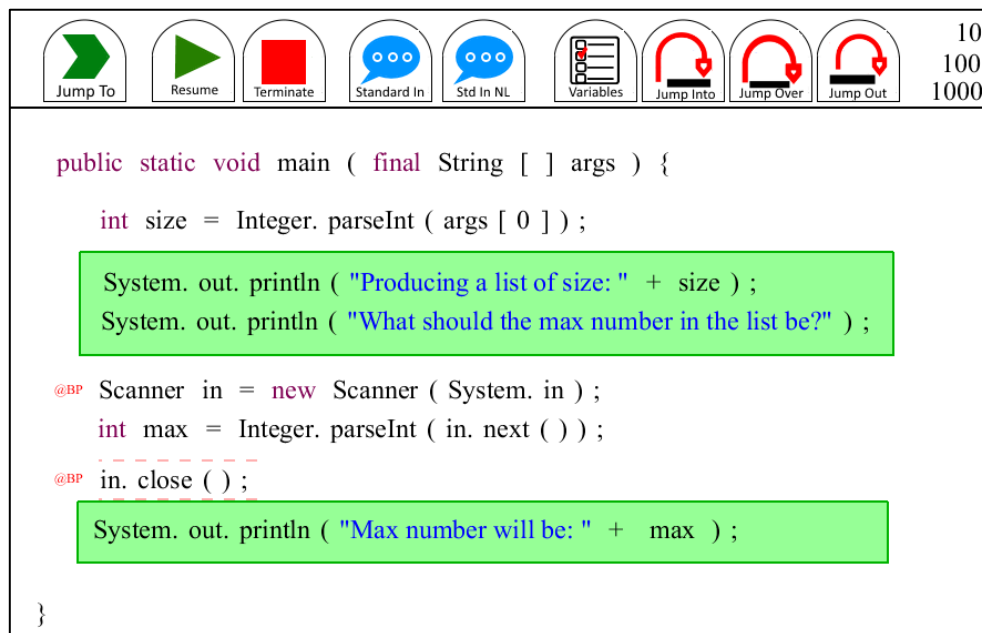


Figure 7.33: An application, once again, suspended at the breakpoint, awaiting instructions.

In Figure 7.34 the programmer has caused the debugger to step, in this case by pressing the jump over button. The dashed red lines have moved to the final statement in the program, but have not yet executed it—as evident from the content in the Message Bay.

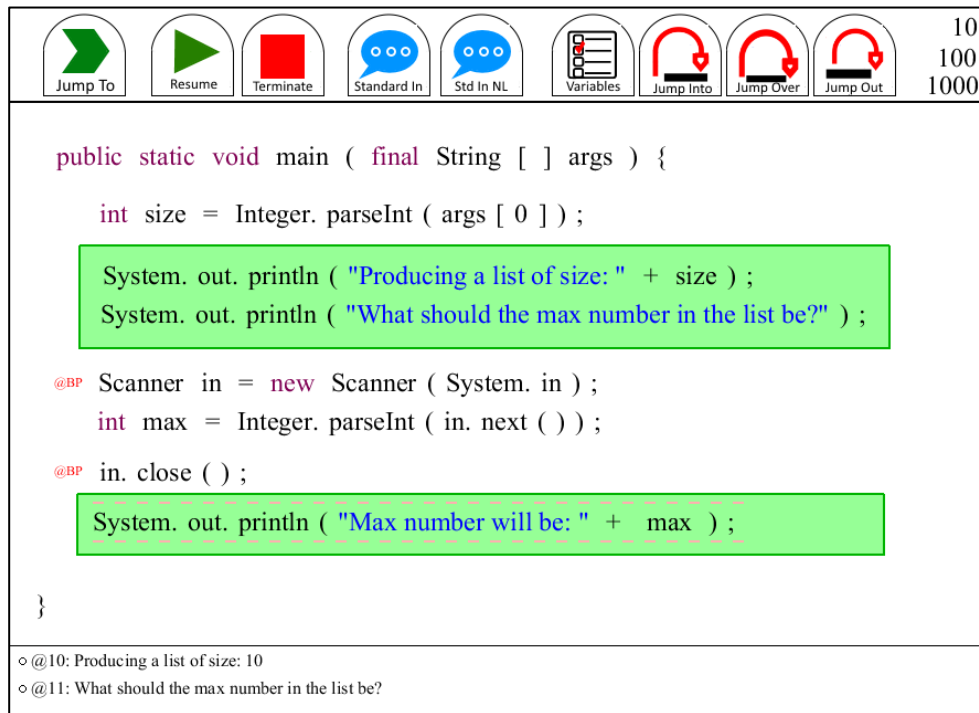


Figure 7.34: An application suspended at a statement without a breakpoint due to the programmer having instructed the debugger to perform a step operation.

Debug Variables. When stopped at a breakpoint, the programmer may wish to inspect the state of the variables. Pressing the sixth button from the left, labelled 'Variables', generates this information and presents it, initially attached to the cursor, as a set of Text Items in an annotation box. The result of this action can be seen in Figure 7.35. At this point, the annotated box containing results of the request for variables has been placed on the Frame. This confirms the values of the variables *size* and *max* to be 10 and 100 respectively.

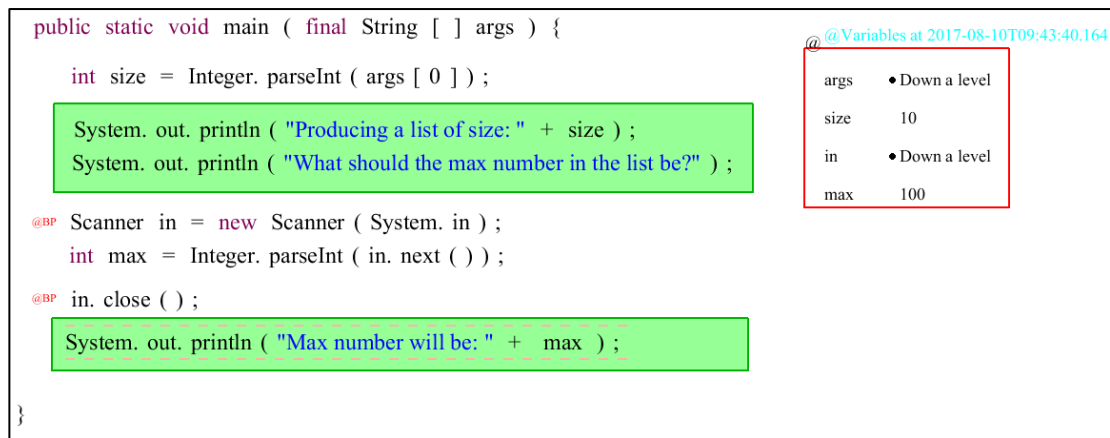


Figure 7.35: A suspended application with the results of a request for state displayed in an annotated box.

The variables *args* and *in* are references to complex types. The Text Item with content ‘Down a level’ and a black dot to its left denotes that the programmer is able to request more information. As can be seen in Figure 7.36, clicking on this Text Item reveals this information in an additional annotated box. Doing so has confirmed that *args[0]* and *size* have identical information.

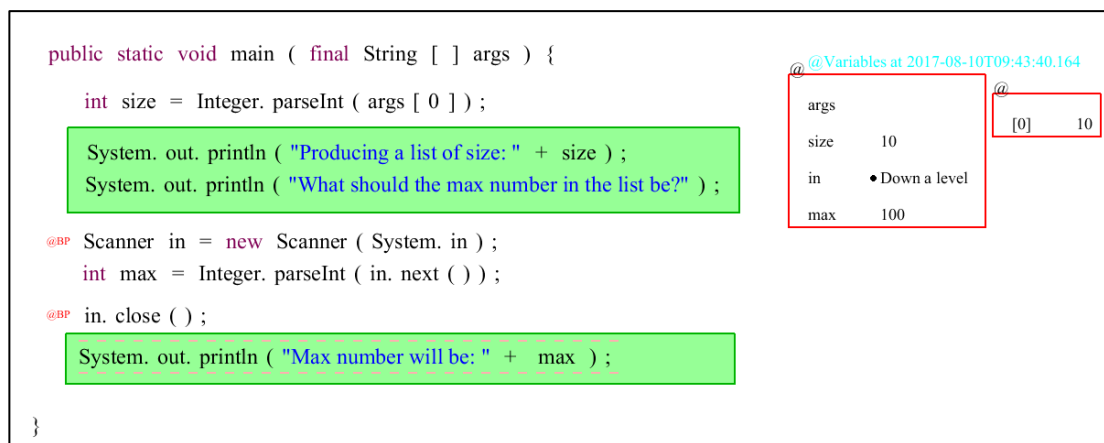


Figure 7.36: A suspended application with the expanded results of a request for state displayed in annotated boxes.

7.4 The Flow Walker Algorithm

In Section 7.1 we looked at how SpIDER allows a programmer to express code. Behind the scenes the flow walker algorithm traverses a set of Frames in order to produce Java source code files. In this section we explain the algorithm in detail.

There are two major components to the flow walker: the within Frame component and the director component. The director component coordinates the within Frame component.

The director component begins the process by instructing the within Frame component to process a top level Frame. When a link is encountered, the director component follows that link, instructs the within Frame component to process it and then substitutes the result of that process for the linked Item that was encountered. This is a recursive process.

In order to process a Frame, the within Frame component infers lines and resolves out of flow content. The director component uses the result of the within Frame component to construct a Java code file which is subsequently compiled. When instructing the within Frame component to process a Frame, the director component, rather than passing the entire Frame, provides the within Frame component with a filtered list of Items from the Frame. This filtered list excludes any non-code Items: annotations, linked Items and Item in the non-code area.

7.4.1 Within Frame Component

When talking about the within Frame component, there are four classes that we will refer to:

- **XRawItem.** An XRawItem encapsulates an Expeditee Item—which for the purpose of this thesis is a SpIDER code Item—and provides functionality to query information on that Item’s bounding box.
- **XGroupItem.** An XGroupItem is a collection of XRawItem objects. A single bounding box surrounds all XRawItem objects in the collection. At the programmer’s request, the XGroupItem object provides a sorted collection of the XRawItems that it contains. How the XRawItems are sorted is detailed below. Both **XRawItem** and **XGroupItem** are subclasses of a base class called **XItem**. This allows for heterogeneous collections of XRawItems and XGroupItems.
- **YOverlappingItemsTopEdge (top edge).** A YOverlappingItemsTopEdge acts as a signal to record the minimum y-position of a set of XItem objects. XItem objects that share the same YOverlappingItemsTopEdge are on the same line.
- **YOverlappingItemsShadow (shadow).** Multiple YOverlappingItemsShadow objects follow a single top edge. They are used to determine whether or not a newly considered XItem belongs to the same line as another XItem. Each shadow holds a reference to the top edge that they belong to. Both **YOverlappingItemsTopEdge** and **YOverlappingItemsShadow** are subtypes of **YOverlappingItemsSpan**.

Inferring Lines. Consider Figure 7.37 which shows how the statement ‘`int i = 5`’, formatted as four separate Text Items (tokens), is processed by SpIDER. The position and size of the text is exaggerated to help the exposition of the data structure formed. The figure shows how the flow walker algorithm processes these tokens to build up a data structure that allows it to deduce that they form a line even though their positions are not pixel perfect in alignment. The array shown slightly left of the centre in each step is an array of `YOverlappingItemsSpan` objects.

1. The `XRawItem` containing the *int* token is processed by inspecting its bounding box. A top edge is inserted into the array at the location corresponding to the top of the `XRawItems` bounding box. The size of the array is the same as the height of the screen in pixels. Therefore, for example, if the top-left corner of the *int* token was positioned at (150,100), then a top edge will be placed at index 100 of the array.

Multiple shadow objects are then placed in the array; beginning at one entry beneath the top edge and continuing until the entry relating to the bottom of the `XRawItems` bounding box. Continuing the previous example, if the height of the `XRawItems` bounding box was 20, then shadow objects will be placed in indexes 101 to 120 inclusive.

2. The `XRawItem` containing the *i* token is then processed by inspecting its bounding box. The algorithm is able to detect that this and a previously processed token belong to the same line. This is achieved because the bounding box on this `XRawItem` would cause shadow objects to be inserted into the array at locations that they already exist.

The bounding box in question actually completely envelops the recorded positions of the previously entered top edge and shadows. The flow walker algorithm amends the array structure by moving the top edge to the position corresponding to the top of the new tokens bounding box and adding new shadow entries until the entry corresponding to the bottom of the bounding box is filled.

The top edge records all the `XRawItems` that are assigned to it and orders them

based on their x-position.

3. The XRawItem containing the = token is then processed by inspecting its bounding box. This token does not produce any new shadows or require the top edge to be moved. The top edge is updated to include this new token in its internal list.
4. The final XRawItem contains the token 5. Its *y-position* is higher than previously seen tokens but its shadow means that it is a part of the same line. This results in the top edge being moved up in the array.

When the flow walker algorithm processes this XGroupItem, the top edge and subsequent shadows will cause a line to be inferred: *int i = 5*.

Note: the shadowing behaviour means that it does not matter which order the flow walker processes tokens in; the result will remain consistent regardless of the order tokens are processed. This is necessary as, the order that Items appear in a Frames data structure are not stable.

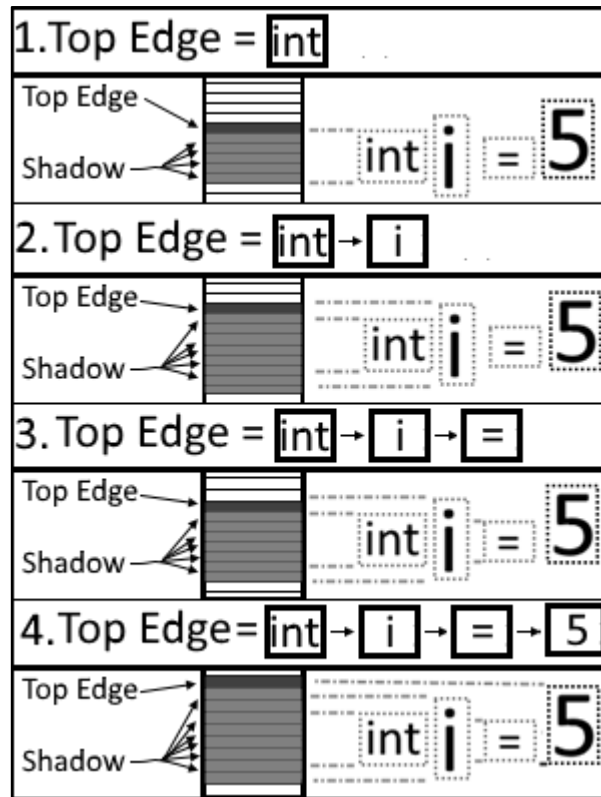


Figure 7.37: Constructing a line using shadows.

Boxing. Boxes provide additional context to the flow walker algorithm, limiting the scope of Text Items to influence how lines are inferred. Each box on a Frame is processed as a separate XGroupItem.

If the intention of a programmer was for the resulting code file to contain six print statements ordered in the following way:

```
System.out.println("1");

System.out.println("2");

System.out.println("3");

System.out.println("4");

System.out.println("5");

System.out.println("6");
```

Then they might be tempted to layout their code in three columns as can be seen in Figure 7.38.

System. out. println ("1") ;	System. out. println ("3") ;	System. out. println ("5") ;
System. out. println ("2") ;	System. out. println ("4") ;	System. out. println ("6") ;

Figure 7.38: No boxing.

However, despite the horizontal gap being bigger than the vertical gap, the process of inferring lines without boxes will cause the flow walker to detect two lines of statements. This results in a code file with the six statements in this undesirable order:

```
System.out.println("1");  
  
System.out.println("3");  
  
System.out.println("5");  
  
System.out.println("2");  
  
System.out.println("4");  
  
System.out.println("6");
```

Figure 7.39 shows a solution to this problem. *Explicitly* placing boxes around pairs of print statements causes the flow walker algorithm to process each as an XGroupItem.

System. out. println ("1") ; System. out. println ("2") ;	System. out. println ("3") ; System. out. println ("4") ;	System. out. println ("5") ; System. out. println ("6") ;
------------------------------------------------------------------	------------------------------------------------------------------	------------------------------------------------------------------

Figure 7.39: Explicit boxing to show grouping.

Whilst there are three boxes present in this figure, there are actually four XGroupItem objects: one for each box, and one for the entire Frame. The XGroupItem that represents the entire Frame contains three XItem objects, all of which happen to be, in this particular example, other XGroupItem objects. As a result of the XItem base class, it is worth reiterating that, in practise, an XGroupItem can contain a mix of XRawItems and XGroupItems. Corollary to this, the inferring line process previously discussed is agnostic as

to what XItem objects are used to form a line—both XRawItem and XGroupItem (boxed) objects cast a shadow into the YOverlappingItemsSpan array.

Figure 7.40 replicates Figure 7.39 but removes the middle box and repositions the first unboxed print statement. This demonstrates a non-intuitive aspect to the line-forming processes as explained so far. The XGroupItem that represents the entire Frame now contains, in order: an XGroupItem, several XRawItems, and a second XGroupItem.

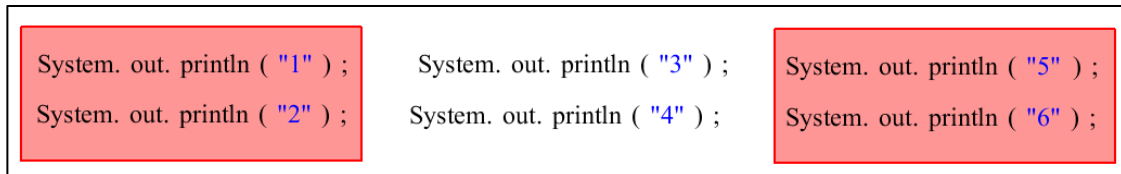


Figure 7.40: No middle box causing a problem.

However, as presented, the line forming algorithm has a flaw. As touched on earlier, when multiple XItem objects are in an XGroupItem, they are ordered by their x-position. Once again, all of the content in Figure 7.40 is contained in an XGroupItem that encapsulates the entire Frame. That is, when the un-boxed print statements are being processed, they are both separately entered into the line that is formed. The first box (XGroupItem) is the first XItem to be added to the line being constructed. As a consequence of the shadow cast by this box, the tokens belonging to the two un-boxed print statements are added to the line based only on their x-position! This causes the resulting code file to contain the following undesirable fragment of code:

```
System. out. println ( "1" );  
  
System .out. println ( "2" );  
  
System. System. out . out. println println ( ( "4" "3" ) ) ; ;  
  
System. out. println ( "5" );  
  
System. out. println ( "6" );
```

To better match the reasonable expectations of users, the flow walker performs *implicit* boxing as part of the formation of lines. Implicit boxes are not visible to the programmer. In the event a line is formed with a mixture of XGroupItem and XRawItem objects, then the line is reprocessed so that any XRawItems occurring between XGroupItems (or before/after

the first/last XGroupItem) are grouped into their own XGroupItem. Figure 7.41 reproduces Figure 7.40 but places an overlay showing where the flow walker will place the implicit box.

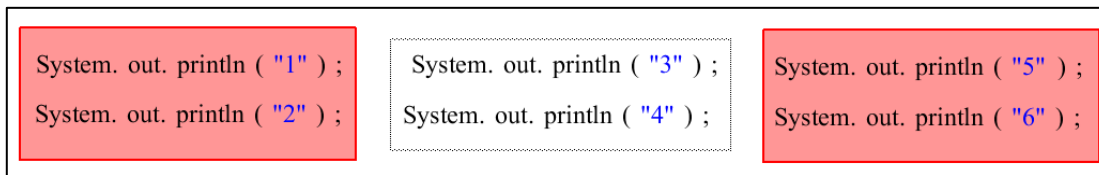


Figure 7.41: Example of Implicit boxing.

Now that the third and fourth print statements are inside an XGroupItem, the shadow cast by the XGroupItem containing all the content on the Frame does not affect their ordering. This implicit box now ensures that the sequence of code produced is as the programmer intended.

```
System.out.println("1");  
System.out.println("2");  
System.out.println("3");  
System.out.println("4");  
System.out.println("5");  
System.out.println("6");
```

Out of Flow. When an arrow is pointing into a box, that box is treated as *out of flow* and an *out of flow calculation* is performed by the within frame component. The out of flow calculation is a two-step process. First, the boxed content is processed as normal; as an XGroupItem object. Secondly, the result of that processing is treated as if it was spatially positioned at the start of the arrow instead of where it is actually positioned.

Consider Figure 7.42. Without the presence of the arrow the resulting code file would produce code that, when executed, prints "Hello World", followed by "i=5". However, because the arrow is present, executing the code would instead print "i=5" followed by "Hello World".

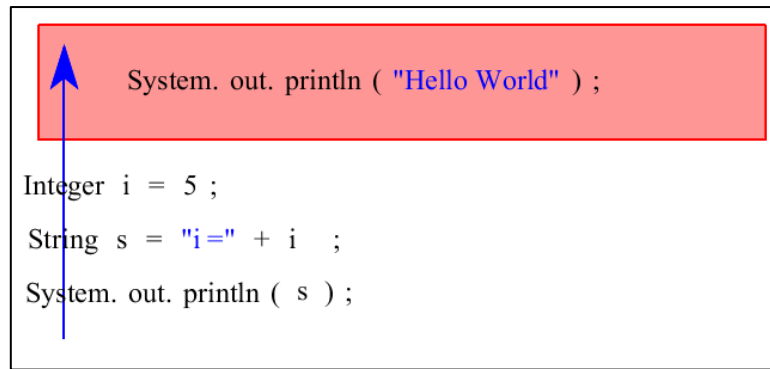


Figure 7.42: Simple Out of Flow example

Chaining. When several boxes are chained together with arrows, the algorithm must work backwards from the last box in the chain. The process, as described in Figure 7.43, begins by recursively following the arrows to find the last box in the chain. If a loop is detected, the process fails and provides an error message. Assuming the last box in the chain was located, the out of flow calculation is performed until all boxes have been resolved.

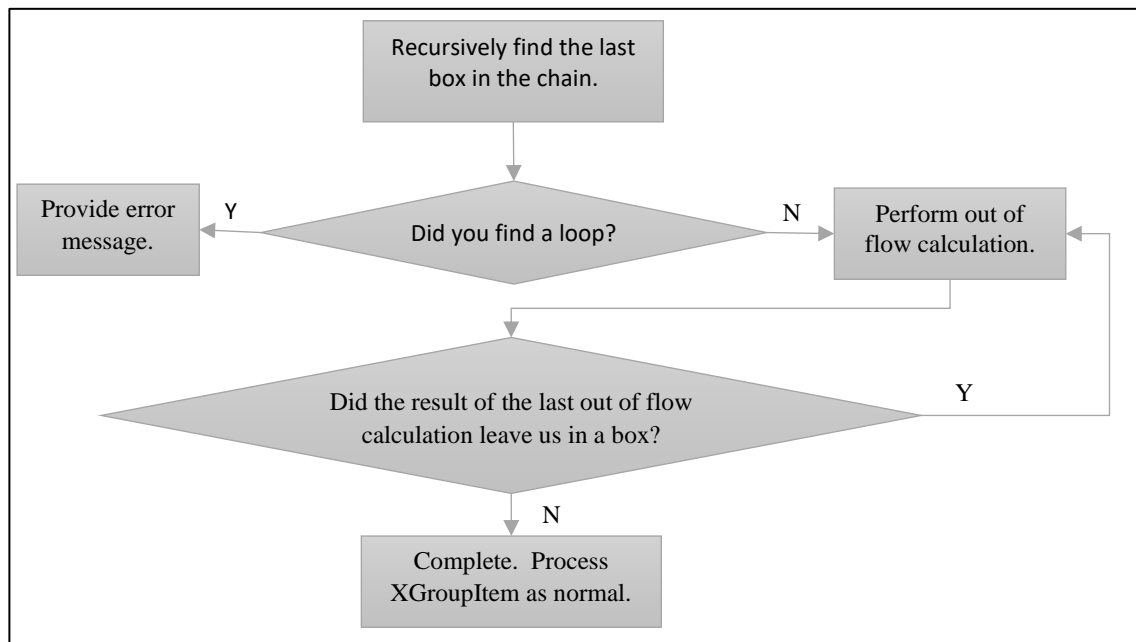


Figure 7.43: The process used to resolve chaining.

While processing a chain, the within frame component must check for loops. It achieves this by performing a depth-first walk over chain and keeping track of the boxes that it has visited. If it finds a box which it has visited before it does not travel down it. This is necessary because a loop in the chain would otherwise cause the within frame walker to continuously go in circles and never terminate.

7.4.2 Director Component

Filtering Content. When asked to produce a Java source file the director component is provided with the frame containing the beginning of the class declaration. This is the home frame of the class. The algorithm begins by collecting all items on the page and performing a series of filters to determine which items are to be considered code.

The following filters determine if an item is considered code:

- Is the item a link? If so, it is not code. It does however require following, which is explained below.
- Is the item in the non-code area? If so, it is not code.
- Is the item an annotation or inside an annotated box? If so, it is not code.
- Is the item attached to the cursor rather than the page? If so, it is not code.
- Has the user requested a debug session? If so SpIDER deals with annotations slightly differently. Breakpoint items are a special type of annotation (represented as *@BP*). If debugging, SpIDER will pick up on breakpoint items and include them as code for the purposes of determining where to break but not including them in the Java source code. All other annotations are treated as normal.

Figure 7.44 show a Frame containing a mix of code and non-code Items. Figure 7.45 shows this same Frame again, but with an overlay hiding the non-code Items that the director component filters away before providing the within Frame component with the Frame's content.

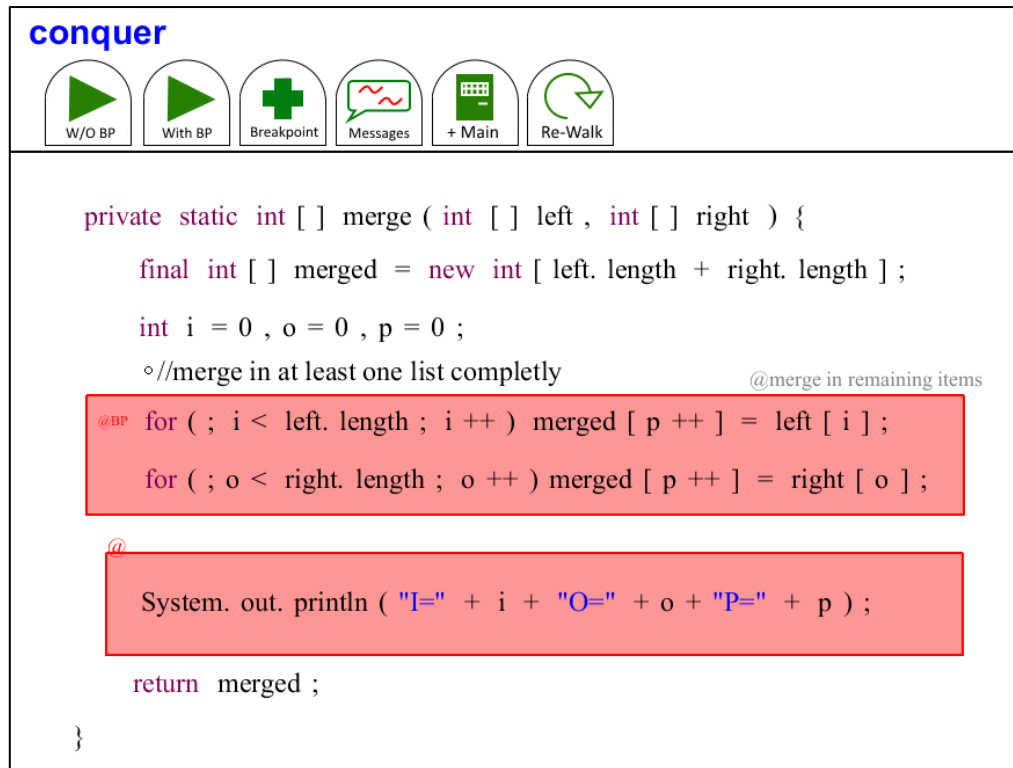


Figure 7.44: A Frame with a large mixture of code and non-code Items.

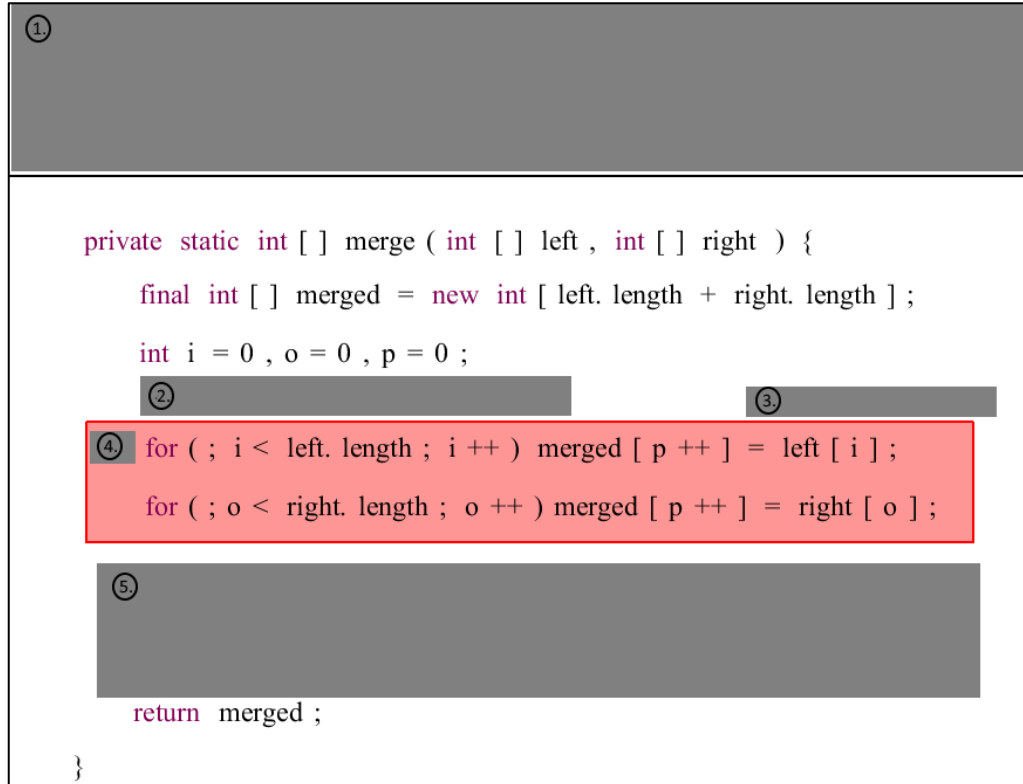


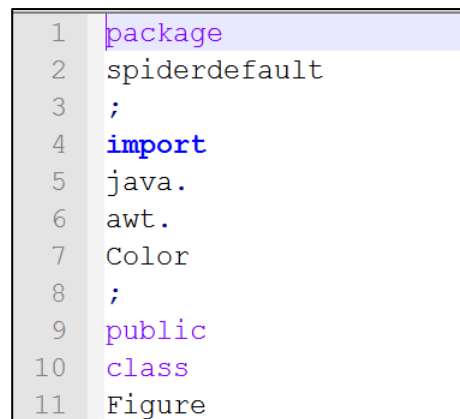
Figure 7.45: Example of frame with non-code items greyed out.

Resulting File. The director component produces three types of files. For each class processed by the flow walker two files are produced:

- A Java source code file that is used for compilation.
- A SpIDER linker file which contain information for linking each Item that appears on screen with the Java token it represents.

For each Java project that gets processed the flow walker produces a *.spiderlinks* file. This file contains XML that describes how to find the home Frame of each package and class within the project it relates to.

The Java source code files are not designed to be edited or read directly. In order to allow SpIDER to index the file as a list of tokens, each token is given its own line. The ability to index tokens is useful for interacting with the Eclipse Java Development Tools (Eclipse JDT). Figure 7.46 shows a screenshot of a Java source file produced by the flow walker algorithm.



```
1 package
2 spiderdefault
3 ;
4 import
5 java.
6 awt.
7 Color
8 ;
9 public
10 class
11 Figure
```

Figure 7.46: A look inside a Java Source file produced by SpIDER.

SpIDER linker files are arranged to mirror a specific Java source file. Figure 7.46 shows the first 11 lines of the *Figure* class and Figure 7.47 shows us the first 11 lines of the corresponding linker file. SpIDER uses these files together to map from either a specific Java token to its representation on screen or vice versa.

```

1 /*st: Tetris2 93 212*/
2 /*st: Tetris2 187 212*/
3 /*st: Tetris2 322 212*/
4 /*st: Tetris2 92 246*/
5 /*st: Tetris2 171 246*/
6 /*st: Tetris2 226 246*/
7 /*st: Tetris2 276 246*/
8 /*st: Tetris2 340 246*/
9 /*st: Tetris2 95 287*/
10 /*st: Tetris2 170 287*/
11 /*st: Tetris2 232 287*/

```

Figure 7.47: A look inside a SpIDER linkers File.

Figure 7.48 shows a screenshot of a *.spiderlinks* file for a Java project named *Tetris*. SpIDER uses this file to find entry points to each class. From parsing this file SpIDER knows that there are five classes to walk and for each class, find its home frame. For example, consider the first class listed in the file. The name of this class is *Game*. It can be found by following a linked Item on the Frame *Tetris1*. The ID of the linked Item is 21.

```

1 <project name="Tetris">
2   <pkg name="spiderdefault">
3     <clazz name="Game" linkerFrame="Tetris1" linkerID="21"/>
4     <clazz linkerID="18" linkerFrame="Tetris1" name="Tetris"/>
5     <clazz name="SquareBoardComponent" linkerFrame="Tetris1" linkerID="15"/>
6     <clazz linkerID="12" linkerFrame="Tetris1" name="SquareBoard"/>
7     <clazz name="Figure" linkerFrame="Tetris1" linkerID="8"/>
8   </pkg>
9 </project>

```

Figure 7.48: A SpIDER linkers file for a project named *Tetris*.

7.5 The Magnet System

When authoring in a relative text editor, the addition or removal of a character has an immediate effect on the surrounding characters, such as causing subsequent characters to shuffle forward when a new character is added. For the purpose of this section, let us refer to these effects as *flow effects*. Wide adoption of relative text editors has led authors to become accustomed to flow effects. (Section 4.2 discussed the differences between relative and absolute editing environments in more detail.)

We consider it good usability if a programmer using SpIDER is able to author code without having to significantly adjust the expectations that they obtained from using a relative

authoring environment. Unfortunately, the absolute positioning of content in Spatial Hypermedia applications means that flow effects do not occur as a matter of course. The absence of flow effects is in conflict with good usability as it requires expectations to be significantly adjusted.

Explicit functionality must be created to support flow effects. Expeditee does allow flow within a Text Item. Typical interaction with Expeditee results in sentence sized Text Items, thus easing the transition between a traditional text editor and Expeditee's spatial editing. However as SpIDER tokenizes programming statements so that each token is its own Text Item, this advantage is lost. In order to make SpIDER's spatial editing more similar to editing in a traditional text editor, some specific flow capabilities have been added.

SpIDER's *magnet* system maintains a data structure that records the relationships between Items. These relationships are used to provide programmers with the flow effect functionality that they are accustomed to from their use of relative text editors. Figure 7.49 shows a screenshot of SpIDER with its magnet debug display active. This debug tool assisted in the development of the magnet system and is not normally available to the programmer. Coloured lines show the relationships that each Item has with another:

- A red and green line between two Items shows that the magnet system considers them to be on the same line. The red lines show a left neighbour relationship—connecting an Item to the Item directly preceding it on the same line, whereas the green lines show a right neighbour relationship—connecting an Item to the Item directly following it on the same line. This distinction allows any Item on a line to be used to collect all the Items on the line.

The concept of a line in the magnet system is different from the concept of a line in the flow walker. Whereas the flow walker will consider two Items whose shadows conflict to be on the same line regardless of their horizontal distance from each other, the magnet system will only consider two Items to have left and right neighbour relationships if they are close enough to each other that one may have to move if the other is altered. In other words, if more than a character length is between two Text Items, they are not considered neighbours. For example, in Figure 7.49, if only a small amount of content was added after the opening bracket

of the parameters used to construct the *Scanner* object, then the *System.in* parameter need not move.

- A blue and yellow line between two Items shows that the magnet system considers them to be the end of one line and start of the next. The yellow line shows a bottom neighbour relationship—connecting an Item to the first Item on the following line, whereas the blue line shows a top neighbour relationship—connecting an Item to the last Item on the preceding line.

```
public static void main( final String[] args ) {
    int size = Integer.parseInt( args[ 0 ] );

    System.out.println( "Producing a list of size: " + size );
    System.out.println( "What should the max number in the list be?" );

    Scanner in = new Scanner( System.in );
    int max = Integer.parseInt( in.next() );
    in.close();

    System.out.println( "Max number will be: " + max );
}
```

Figure 7.49: Debug display for SpIDER's magnet system.

Every time an edit is made, the magnet system recalculates all the neighbourhood relationships on the currently open Frame. When the programmer presses a key, such as backspace or the left arrow, the magnet system is used to reposition the surrounding text. The following keys utilise the magnet system:

- **Arrow Keys.** Base Expeditee functionality allows arrow keys to be used to navigate through the characters in a Text Item in order. Because SpIDER tokenization produces a separate Text Item for each token, the magnet system extends this functionality to allow for navigation between Text Items. For example, if the cursor is currently sitting between the last and second to last character in a Text Item, then pressing the right arrow once will move the cursor to the end of the Text Item. Pressing the right arrow an additional time will move the cursor to the beginning of

the Text Item on the right; accessed by way of right neighbour reference.

- **Backspace/Delete.** When the programmer presses the backspace or delete key, the magnet system must first determine whether the cursor is the start, the end, or somewhere in the middle of a line. If the cursor is placed somewhere in the middle of the line, then a character from a Text Item will be removed, shrinking the Text Item. Right neighbour references are used collect the Text Items following the shrunken Text Item so that they can be shuffled an appropriate distance to the left.

If the cursor is placed at the end of a line and the delete key has been pressed, then the bottom neighbour reference and subsequently right neighbour references are used to collect the Items from the line beneath the altered line and move it up to become part of the altered line. A similar operation occurs if the backspace key is used at the start of a line.

When merging lines the magnet system ensures that the resulting line is not too long. For example, this operation cannot cause a line to intrude over the edge of a box. When the operation would cause lines to be inappropriately long, it instead moves up part of the line from beneath and moves the remaining lower line left a distance equal to the length of the tokens moved up.

- **Enter.** When the programmer presses the enter key, the magnet system moves all Text Items following the selected Text Item down. This does not cause two lines to merge. Instead, neighbourhood references are used to also move all following lines down as well. If the cursor is positioned right at the start of a Text Item, that Text Item is moved with the Text Items to its right, otherwise, it is excluded from the operation.
- **Insert a character.** When a character is inserted, Text Items to the right of the modified Text Item are moved an appropriate distance to the right.

SpIDER does not completely emulate the flow effects present in a traditional text editor. For example, if a vertical gap greater than the height of a line is present between two sets of

lines, then the magnet system will not connect the last Item in the first set with the first Item in the second set. Therefore, flow effects on one of the lines in the top set will have no effect on the bottom set of lines. This is a purposeful design decision, aimed at limiting the effect of the magnet system on spatial stability (see Section 2.3 for information on spatial stability).

7.6 Integration with the Eclipse Java Development Tools

The development of SpIDER has been achieved by extending Expeditee and the Eclipse JDT. This relationship is pictorially represented in Figure 7.50, identifying the primary JDT extensions points and aspects of Expeditee that SpIDER utilises.

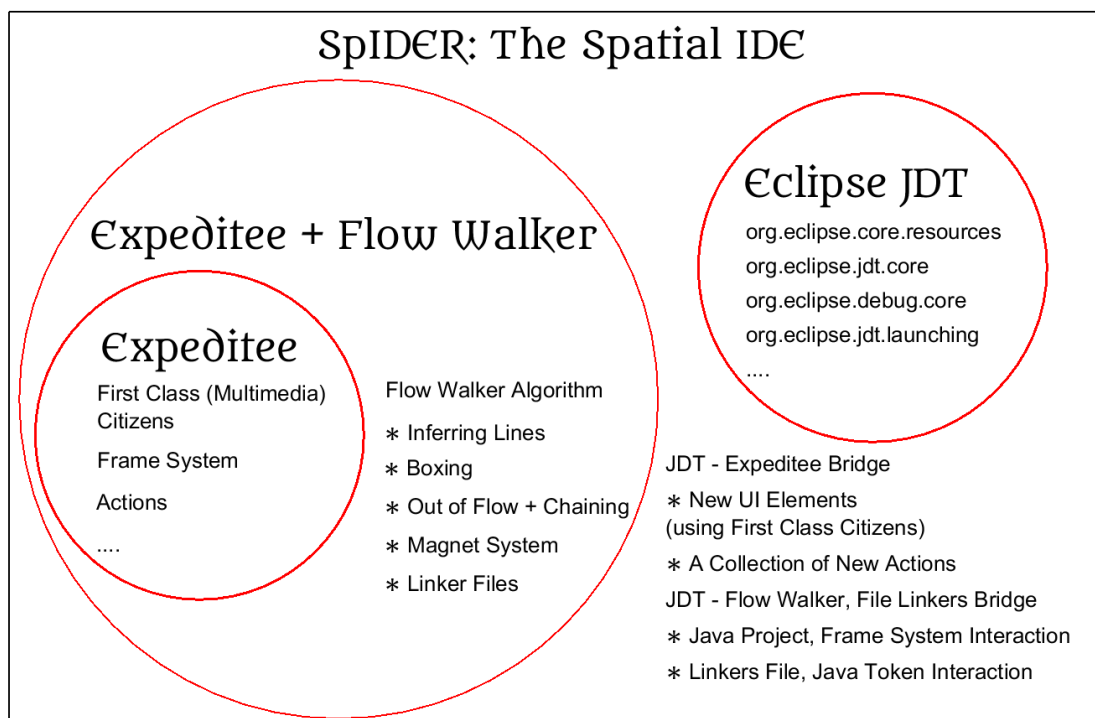


Figure 7.50: The structure of SpIDER represented in a venn diagram.

Use of Existing Expeditee Functionality. Expeditee's First Class Citizens are sufficient for building the new user interface elements required in SpIDER. For example the user interface element that displays 'content assist' results is built out of linked Text Items, boxes and annotations. This has the advantage of allowing a user to interact with the new user interface element just as other First Class Citizens. The Frame System has been utilised to allow programmers to arrange Java projects and their associated code. New Expeditee

Actions have been developed to expedite a programmer's work—such as the ability to create new packages or attach a main method template to the cursor.

Expeditee Extensions. Implementation details of the flow walker algorithm and the magnet system were detailed in Sections 7.5 and 7.4 respectively. The development of this algorithm was necessary to allow for the spatial layout of code. Previous serialisation techniques included in Expeditee were not suitable for the task.

SpIDER wrapped the Flow Walker algorithm adding additional functionality so that it could operate over a tree of Frames and interact with details provided by Eclipse JDT. This was detailed in Section 7.4.

Eclipse JDT. The Eclipse JDT provides a number of extension points. These were used in the development of SpIDER. The following are the most notable:

- **Creation and Management of Java Projects.** The packages *org.eclipse.core.resources* and *org.eclipse.ui* were used to access and manipulate the Eclipse workspace, which itself provided access to manipulate and create Java projects. Of particular importance for this task was the class *PlatformUI* with the static function *getWorkbench()*.
- **Accessing IDE Functionality.** The *org.eclipse.jdt.core* package provided the interfaces necessary for building and interrogating Java code. For example, this allowed SpIDER to request content assist information and receive alerts for problems with the Java code. In each case, having obtained this information, SpIDER was able to build and display appropriate feedback to the user.
- **Launching and Debugging Java Projects.** The packages *org.eclipse.debug.core*, *org.eclipse.jdt.launching* and *org.eclipse.jdt.debug.core* provided the access points for running and debugging code.

Java programs in SpIDER are represented by first class citizens, each located at an absolute position on a specific Frame. The Eclipse JDT however, represents the same programs as serial text files and syntax trees. SpIDER maps between these different representations as described in Section 7.4 and as shown in Sections 7.1–7.3. Operationally this occurs when: SpIDER is launched; content is edited; the programmer requests content assist; inserting

breakpoints; setting breakpoints; stepping through code; requesting variable information during a debug session and reporting warning and errors.

Obtaining SpIDER and Expeditee. If you would like to try out SpIDER, you can obtain the executable at <https://sourceforge.net/projects/spatial-ide-research-spider>. SpIDER's source code can also be found at that location. If you would like to run SpIDER directly from source code you will need to obtain the Expeditee Jar file. This can be obtained at <https://expeditee.org>. A video demonstrating SpIDER's functionality can be viewed on the YouTube channel: https://www.youtube.com/channel/UCY_7pELIfjrxUaVN_R7j63Q. The video is titled *SpIDER Showcase*.

Chapter 8

Evaluating SpIDER Spatial Layout

The flow walker algorithm is what allows programmers to spatially layout code in SpIDER. In order to achieve this, programmers are expected to draw arrows and boxes in-line with what the flow walker expects. Ideally a programmer should be able to lay code out in SpIDER's spatial style with minimal instruction. Reviewing how the flow walker interprets code as described in Chapter 7:

- It scans the page left-to-right and top-to-bottom and allows for slight variations in the Y coordinate when forming lines. It does not however decide where columns would be formed.
- It uses boxes to encapsulate content. Boxes can contain other boxes. This allows programmers to explicitly form columns, even nested columns.
- It uses arrows combined with boxes to allow for out of flow content, as explained in Chapter 7.
- Multiple arrows and boxes can be strung together to produce chains.

An initial study was designed and executed to answer the following question: how well does the output of the flow walker algorithm match what participants expect? The design of this study, along with justifications for that design, as well as listed demographics are provided prior to the study results. Following analysis and summary of this initial study a follow-up study is documented.

8.1 Initial Study

In order to establish how well the rules that govern the flow walker algorithm match what humans expect to happen, a multi-choice questionnaire was devised. Broken into six logically grouped parts, the quiz questions each contained screenshots of very simple

pseudocode paired with a multi-choice question about what the code would do. Participants would have to choose one of the options for each question before moving onto the next part. Each time a participant selected an answer it would be recorded, so that in analysis it could be detected when participants changed their minds partway through a section. The quiz was built as a website. Analysis of each part of the quiz is provided in Section 8.1.2. Whenever a question from this quiz is listed in this thesis, it will be arranged for presentation rather than how participants saw it. Figure 8.1 is a screenshot from part way through the quiz, showing how participants had the questions presented to them.

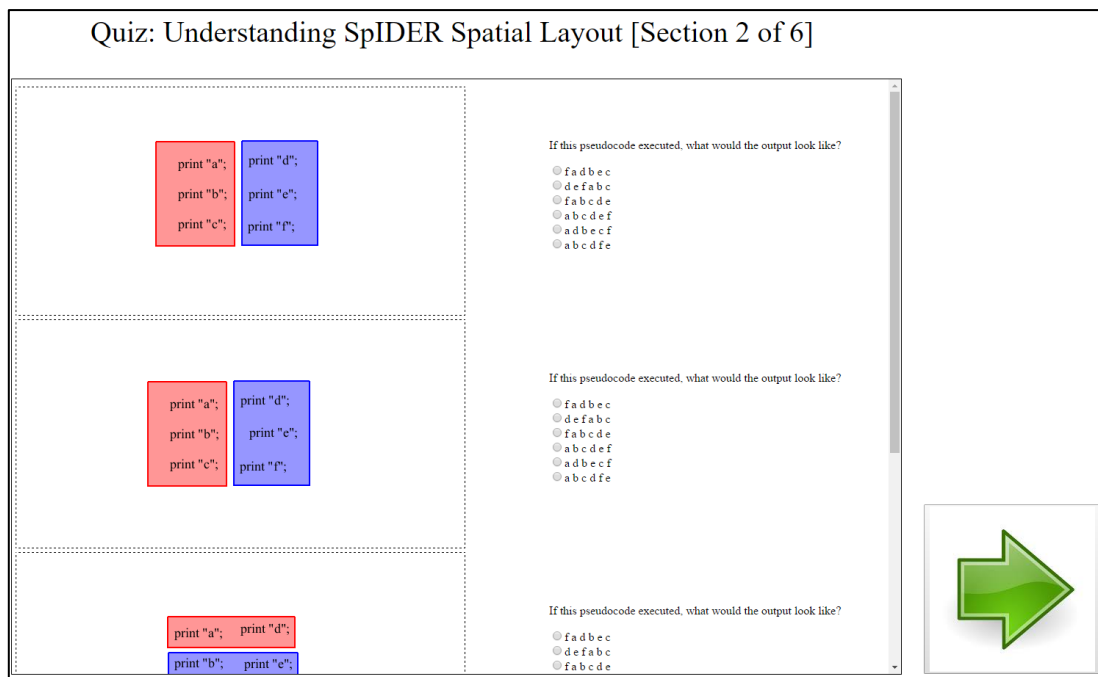


Figure 8.1: Screenshot from initial quiz.

Design Justification. As the goal was to establish if SpIDER’s spatial layout of code matched user expectations, simple pseudocode was used. This simple pseudocode did not contain suggestive syntax such as brackets, so that the ability to use existing programming knowledge did not help in deciding on the answer. The first four parts contained four questions each and tested a different aspect of the flow walker:

- Part 1: The first set of questions sought to test participants’ expectations on the simple ordering of elements, testing specifically how people would form lines out of and order the elements on the page without any additional spatial elements such as arrows and boxes.

- Part 2: The second set of questions introduced boxes, to test how participants would react to items being enclosed in rectangles.
- Part 3: The third set of questions introduced arrows and boxes, allowing for out of flow content. Of particular interest here was how participants would treat the start and end of the arrows.
- Part 4: The fourth set of questions dealt with chaining. Questions for chaining were similar to those given when testing out of flow.

A full list of questions can be found in Appendix A, ethical approval is found in Appendix C.

Rounding out the questionnaire, parts 5 and 6 each contained two questions using pseudocode closer to realistic code snippets. They also included other spatial hypermedia elements, such as diagrams.

- The fifth set of questions expressed pseudocode that modelled a ball bouncing between two walls. The code was expressed with a diagram to provide context to the code.
- The sixth set of questions expressed pseudocode that modelled some hitbox testing.

Representative examples from each part of the quiz are shown in the relevant sections that follow, where analysis of the questionnaire is given.

As we are looking to establish how close the flow walker algorithms output matches human intuition, it is desirable that participants answer each question quickly without much thought. To this end, each individual question within a category shares only the same range of functionality (for example, part two has boxes but not yet out of flow), and do not build on top of those previous. This combined with the short duration of the entire questionnaire should heavily limit the opportunity for participants to ‘learn’ over time.

Demographics. The study used 18 high school students as participants. The students were in the process of taking an IT course at high school. This ensured a minimum level of computer literacy. While demographics such as programming experience were recorded the study was concerned with how people would follow spatially laid out code and not if they would understand it, therefore, limited programming experience present among the participants was not seen as a limiting factor. Demographics collected:

- All participants were male.
- Programming experience ranged from less than a year to four years, with a mean of 2.2 years and a mode of 2 years.
- The most popular programming language was Visual Basic with 12 participants noting it as their preferred programming language. Three participants preferred C#, one preferred HTML and the remaining participants did not have a preference.

8.1.1 Study Results

Below we provide a sample of questions asked in the survey along with their results. To make the connection between the multiple choice answers and the pie-charts depicting how participants answered, we have coloured the answers to match the respective pie segment. Furthermore the answer among the provided options that the flow walker would produce is emphasised in bold and italics. This colouring and formatting was not present in the quiz presented to participants.

Part 1, Question 1.

Question: What would this pseudocode print?

```
print "a";    print "d";    print "b";
print "e";    print "c";    print "f";
```

Provided Options:

- A. *A E D C B F*
- B. *A D B E C F*
- C. A B C D E F
- D. *B A D E C F*
- E. *A D E B C F*

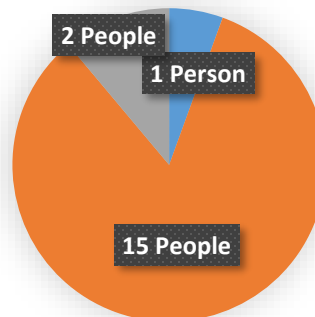
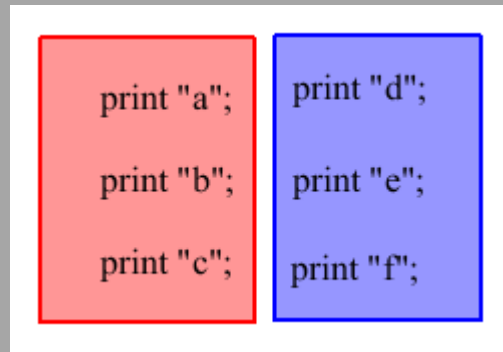


Figure 8.2: Question and results from Question 1, Part 1 of the initial quiz.

Part 2, Question 1.

Question: What would this pseudocode print?



Provided Options:

- A. **F A D B E C**
- B. **D E F A B C**
- C. **F A B C D E**
- D. **A B C D E F**
- E. **A D B E C F**
- F. **A B C D F E**

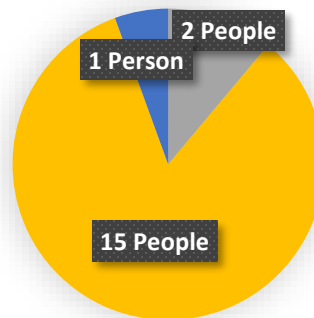


Figure 8.3: Question and results from Question 2, Part 2 of the initial quiz.

8.1.2 Analysis of Initial Study Results

Part 1: Forming Lines. Averaged over four questions, the first part of this study had 72% of respondents agree with the flow walker. For the first three questions respondents respectively agreed with the algorithm 83%, 78% and 78% of the time, however the fourth question had only 50% agreement with the algorithm. Inspection of the answers for this question (detailed in Figure 8.4) showed that 39% of people had picked the answer that assumed the print statements were treated as two separate columns, processing the left-hand column before moving onto the right-hand column.

Question: What would this pseudocode print?

```
print "a";
```

```
print "b";
```

```
print "c";
```

```
print "d";
```

```
print "e";
```

```
print "f";
```

Provided Options:

- A. A E D C B F
- B. A D B E C F
- C. A B C D E F
- D. B A D E C F
- E. A D E B C F

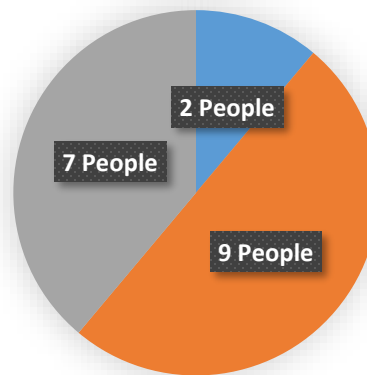


Figure 8.4: Question and results from Question 4, Part 1 of the initial quiz.

This result is interesting, as it suggests that participants are willing to spatially sort content into columns in their mind even though in traditional flat file IDEs the white space between content on a single line would be ignored. It also shows that the ability to express columns in a spatial system is more important than in a flat file system. Overall it appears that the majority of participants' answers tend to agree with what the flow walker produces, as seen in Figure 8.5.

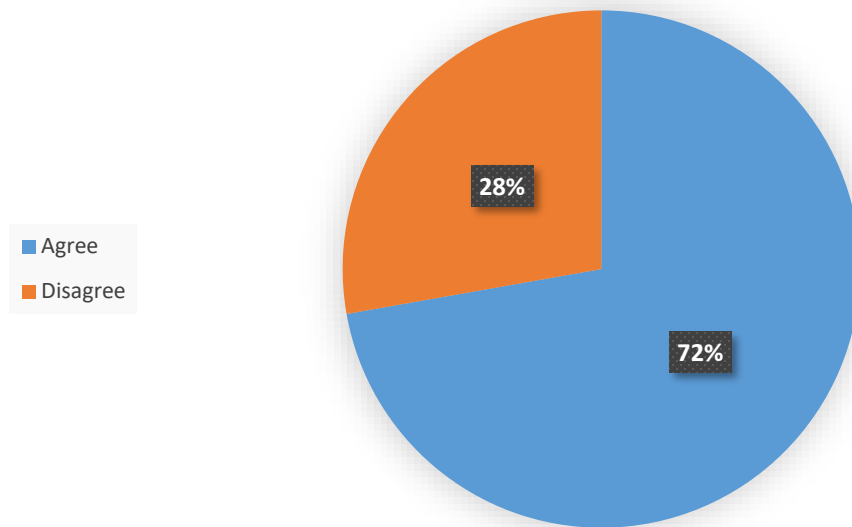


Figure 8.5: Overall agreement with algorithm, Part 1 of the initial Quiz.

Part 2: Boxing. The second set of questions has similar results to the first, with a slight increase to 78% of people agreeing with what the flow walker would produce. For the first three questions participants agreed with the algorithm 83%, 78% and 89% of the time respectively with the final question bringing the average down with 61% agreement, see Figure 8.6. Unlike with the previous section, those disagreeing with the algorithm on the final question had quite varied answers with 17% of participants being the second biggest group.

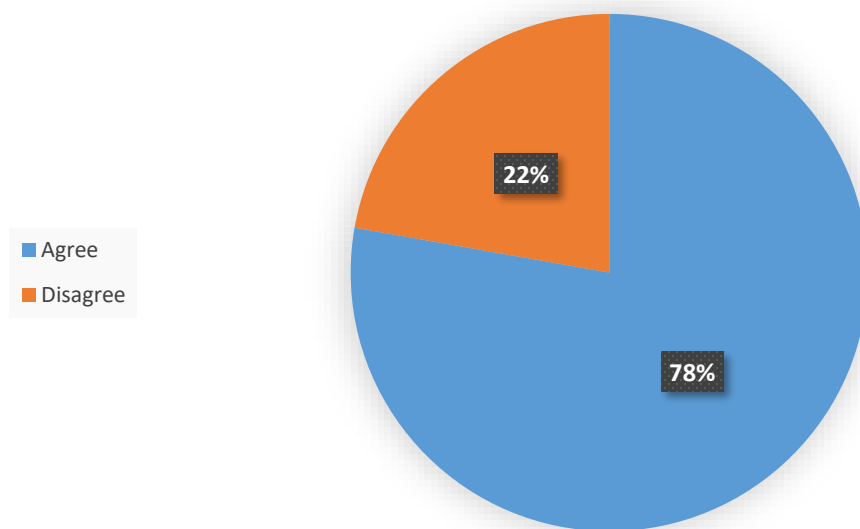


Figure 8.6: Overall agreement with algorithm, Part 2 of the initial quiz.

Analysing the final question further we see that it is the only question in this section that contains content outside of a box. This appears to have been the cause of confusion for this question. Whilst the majority of participants agreed with the flow walker algorithm we can see that the most common disagreeing answer was: *F A B C D E*. It seems a reasonable answer to conclude that participants decided that items outside of boxes are processed prior to items inside boxes.

Question: What would this pseudocode print?

```
print "a";
print "b";
print "c";
```

```
print "d";
print "e";
```

print "f";

Provided Options:

- A. *F A D B E C*
- B. *D E F A B C*
- C. *F A B C D E*
- D. *A B C D E F*
- E. *A D B E C F*
- F. *A B C D F E*

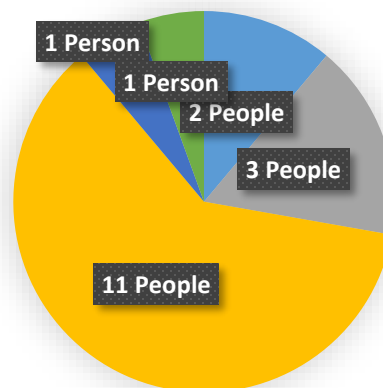


Figure 8.7: Question and results from Question 4, Part 2 of the initial quiz.

Part 3: Out of Flow. The third set of questions shows a significant drop in agreement with the flow walker, see Figure 8.8. Overall 47% of answers participants gave agreed with the algorithm. Participant answers agreed with the flow walker 50%, 67%, 17% and 55% of the time respectively. Those disagreeing with the algorithm had varied answers with two of the four questions having participants selecting five out of six available answers.

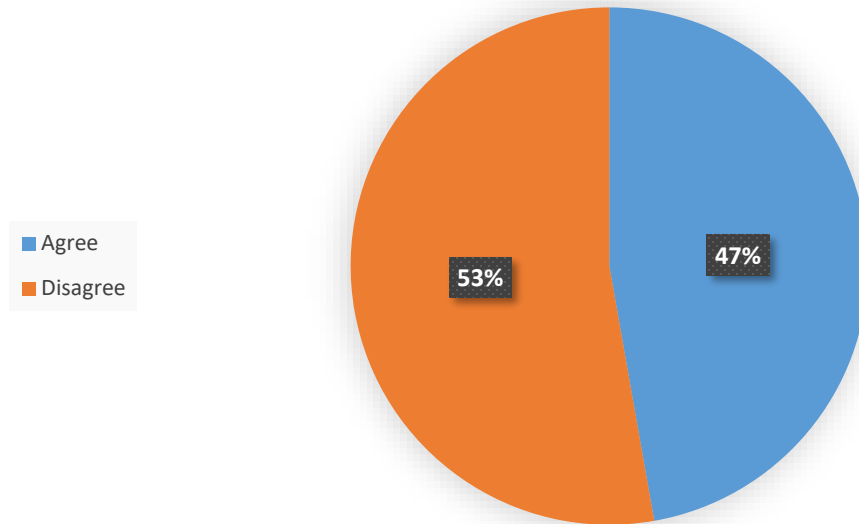
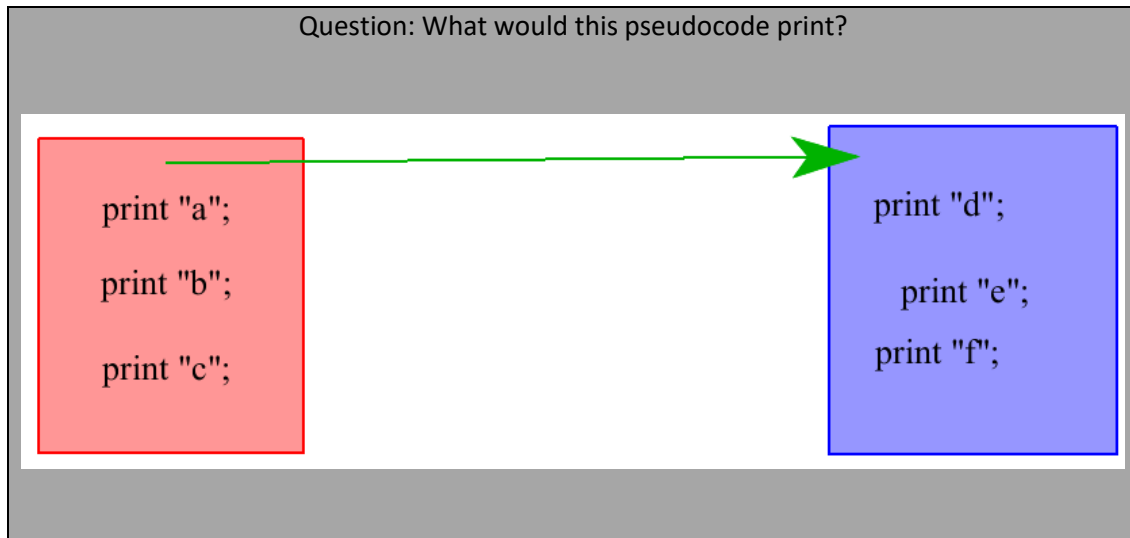


Figure 8.8: Overall agreement with algorithm, Part 3 of the initial quiz.

Question 3 of Part 3 showed particularly divided opinion. Only three of the 18 participants chose the answer that matched what the algorithm would do. The most popular opinion was answer D with eight people (44% of participants). Here we can guess that participants processed the left box before moving onto the right; moreover, we cannot even be sure that they are paying any attention to the arrow at all. The other option more popular than what the algorithm would do was answer C (A D E F B C) with four people (22% of participants), for this answer we can guess that participants saw the arrow as attached to the first print statement, processing it first and then following the arrow to the second box before returning. This question can be seen in detail in Figure 8.9.



Provided Options:

- A. **D A B C E F**
- B. **A E F B C D E F**
- C. **A D E F B C**
- D. **A B C D E F**
- E. **D E F A B C**
- F. **A B C D E F A B C**

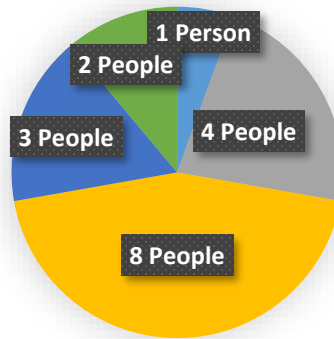


Figure 8.9: Question and results from Question 4, Part 2 of the initial quiz.

Most of the answers provided that do not match the algorithm suggest uncertainty about how to treat arrows, particularly the start. A conjecture was formed: if the starting position of the arrow is positioned more carefully in the centre of a box and is given sufficient space as to appear on its own line then people are more likely to correctly interpret how arrows are dealt with by the flow walker algorithm. We take this, along with other insights from this initial study to a follow-up study that is described in Section 8.2.

Part 4: Out of Flow Chaining. The fourth set of questions continues the pattern of declining agreement with only 33% of participant's answers agreeing with the algorithm, as seen in Figure 8.10. All questions received varied answers with at least four different options being chosen for each question and one question (question 2) getting only one person agreeing with what the algorithm would do.

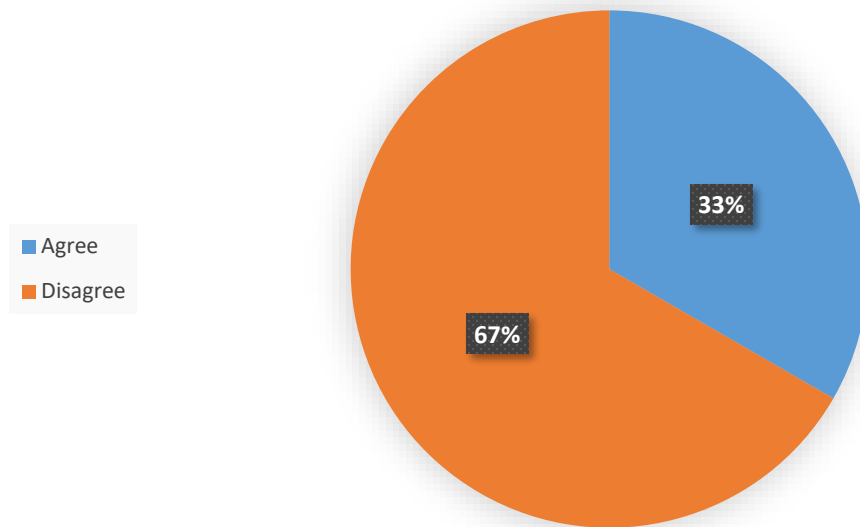
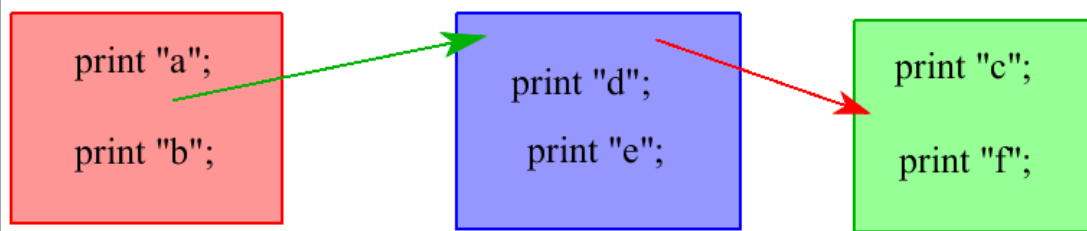


Figure 8.10: Overall agreement with Algorithm, Part 4 of the initial quiz.

The pattern of answers given for Question 2 (seen in detail in Figure 8.11) reinforced the founding of the conjecture through the analysis of Part 3. Over half of the respondents (56%, 10 participants) favoured similar behaviour to what was seen in Question 3 of Part 3, processing one complete box before moving onto the next.

Question: What would this pseudocode print?



Provided Options:

- A. **C A B D E F**
- B. **A B D E A B C F**
- C. **C D A B E F**
- D. **A D E C F B**
- E. **A B D E C F**
- F. **D E C F A B**
- G. **A C F D E B**
- H. **A B D E C F C F**

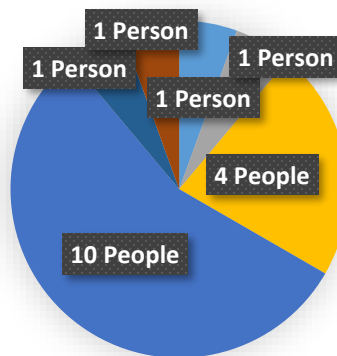
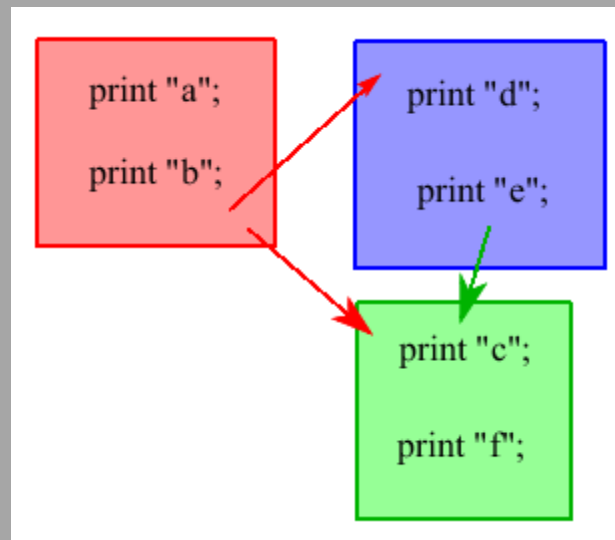


Figure 8.11: Question and results from Question 2, Part 4 of the initial quiz.

Question 4 of Part 4 is interesting because of the varied answers it received with 28% of participants agreeing with the algorithm, and the most popular answer being selected by only 39% of people, these results can be viewed in Figure 8.12. More testing was needed in order to understand what was happening with the answers to this question. A second conjecture was formed: People expect boxes to be used to form columns, consequently arranging one box on top of another causes confusion. If so, then would giving each box its own column produce answers more in line with the algorithm? This conjecture was tested in the follow-up study that is described in Section 8.2.

Question: What would this pseudocode print?



Provided Options:

- A. **C A B D E F**
- B. **A B D E A B C F**
- C. C D A B E F
- D. **A D E C F B**
- E. **A B D E C F**
- F. **D E C F A B**
- G. **A C F D E B**
- H. **A B D E C F C F**

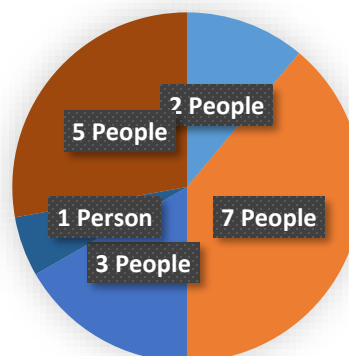


Figure 8.12: Question and results from Question 4, Part 4 of the initial quiz.

Part 5 & Part 6: More Realistic Pseudocode. Parts 5 and 6 featured more realistic pseudocode and as such we expected participants to be able to interpret the order the code appeared in more easily. Beyond more realistic pseudocode the questions also showcased how diagrams can be integrated into the code. Figure 8.13 shows Question 2 of Part 5 and Figure 8.14 shows the overall agreement with the flow walker in Part 5. Figure 8.15 shows Question 2 of Part 6 and Figure 8.16 shows the overall agreement with the flow walker in Part 6.

Question: Here is the complete snippet. At the end of the 3rd iteration through the loop, which walls have been hit?

//A Ball Bounces between two walls

leftWallX =

width =

vx =

rightWallX = leftWallX + width

x = leftWallX + 2

while true

 x = x + vx

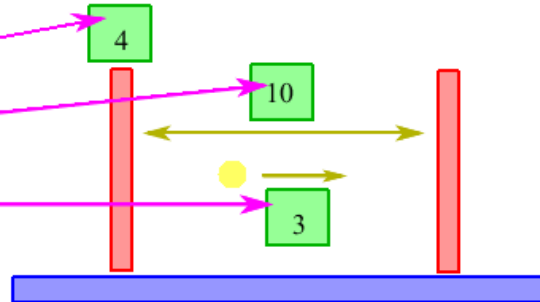
 if x > rightWallX

 x = rightWallX

 if x < leftWallX

 x = leftWallX

vx = -vx



Provided Options:

- A. Left Wall Only
- B. Right Wall Only
- C. Neither Wall
- D. Both Walls

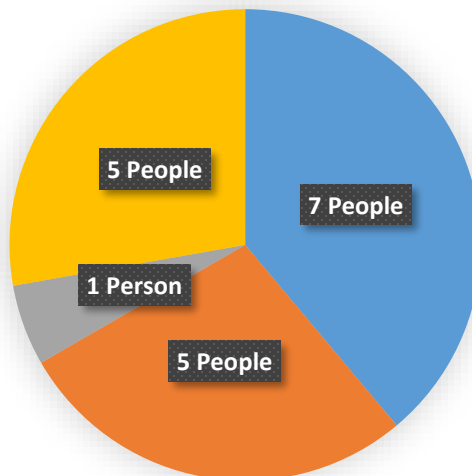


Figure 8.13: Question and results from Question 2, Part 5 of the initial quiz.

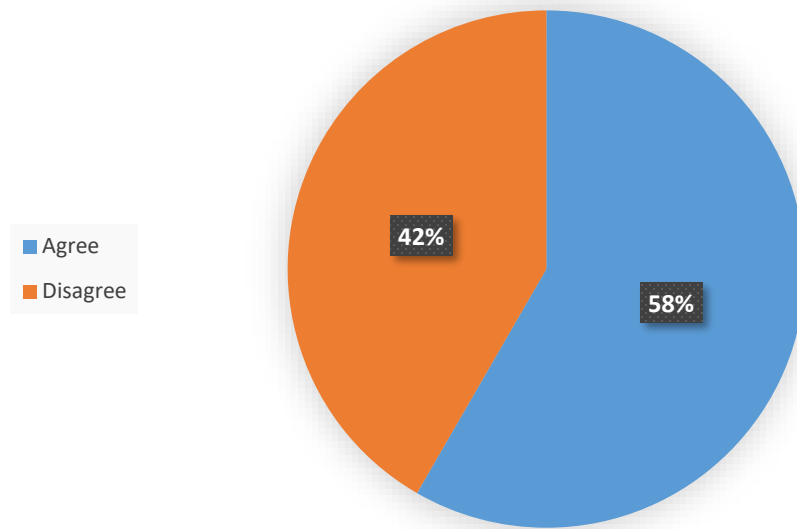


Figure 8.14: Overall agreement with Algorithm, Part 5 of the initial quiz.

Question: Here is the same code again. If hit Square(3, 7,10) is executed, what will print?

```
//Hit box testing: which part of the square are we hitting?
```

```
func hitSquare(x, y, size)
```

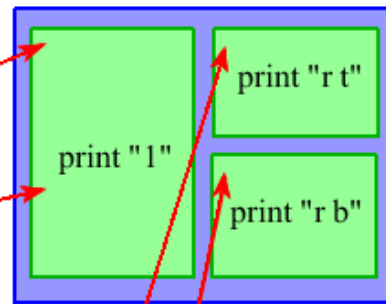
```
    half = size / 2
```

```
    if x < half AND y < half
```

```
    else if x < half AND y >= half
```

```
    else if x >= half AND y < half
```

```
    else
```



Provided Options:

- A. l
- B. //
- C. r t
- D. r b

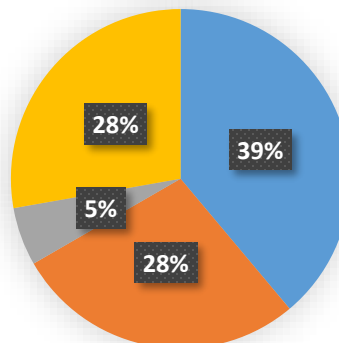


Figure 8.15: Question and results from Question 2, Part 6 of the initial quiz.

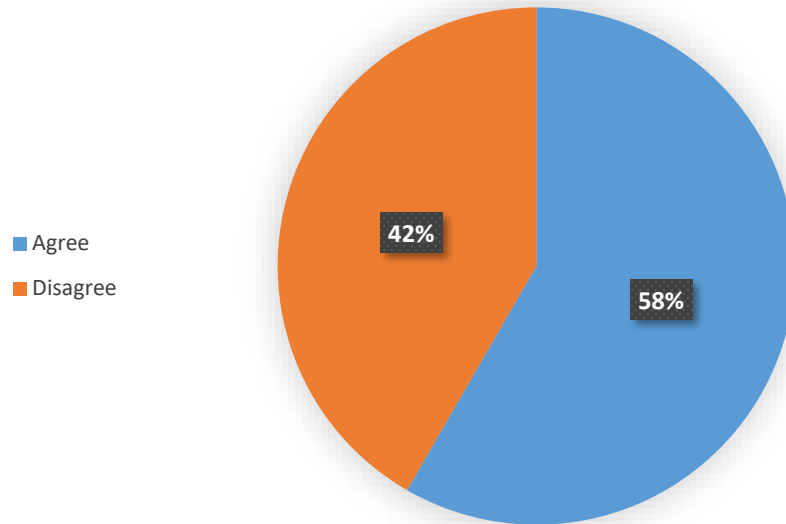


Figure 8.16: Overall agreement with algorithm, Part 6 of the initial quiz.

The overall agreement with the algorithm in each of these sections was 58%. As each Part had only two questions, potential insights are more limited. Boxing and out of flow was used but chaining was not. Three out of the four questions asked in these two Parts showed agreement exceeding the overall agreement of 47% found when examining Out of Flow. This demonstrates that changing the task from using print statement pseudocode to more complex pseudocode also changes the task from understanding the flow of the code to interpreting the code.

8.1.3 Summary of Initial Study

The initial study showed promising results. In the first two parts, that tested Forming Lines and Boxing, people agree with the flow walker algorithm roughly three-quarters of the time. It is likely that pseudocode (or Java code) denoting more complex tasks, and repeated use of SpIDER leading to familiarity, would increase this value further. However examination of the Out of Flow section shows a drastic decrease in agreement with the algorithm, often with results showing two or three popular interpretations. The section on Chaining continues this decline. A confidence level can be calculated to determine how frequently participants agree with the flow walker algorithm: $57.5\% \pm 17.39\%$ of the time. A copy of the math used to calculate this figure can be found in Appendix B.

8.2 Follow up Study

To refine the results found in Part 3 and Part 4 of the initial study two conjectures were formed and a study designed to test them:

1. *The positioning of the arrow when performing out of flow would have a large effect on a participant's intuition of how it functions.*

The flow walker algorithm does not pay attention to the exact positioning of the head of the arrow, only which box it lands in. However the origin of the arrow can be used to pinpoint more precisely where the out of flow operation occurs at. Consequently providing space around and positioning the origin of the arrow in the centre of the surrounding lines would visually communicate its function more accurately.

2. *When creating a chain, people will treat separate boxes that are roughly aligned vertically as belonging to the same column.*

This suggested to us that avoiding this type of layout, by giving each box its own column, would be remove this confusion.

8.2.1 Study Design

As we had narrower goals with this follow up study we took the opportunity to redesign how the conjectures would be tested. Students from two Waikato University Computer Science classes were chosen as test subjects, one was a second-year programming course and the other a third year HCI course. At the end of a lecture students who wanted to participate were shown a single problem similar to those in the initial study and asked to write their single letter answer on a piece of paper and hand it in as they left. One class was used as a control group and given a question strongly resembling one from the initial test and the other the same question but modified in some way as to test one of the conjectures. The results from each class were compared. Participant pools varied in size between 21 and 54 as they were determined by how many people were showing up to the lecture and how many of those participated.

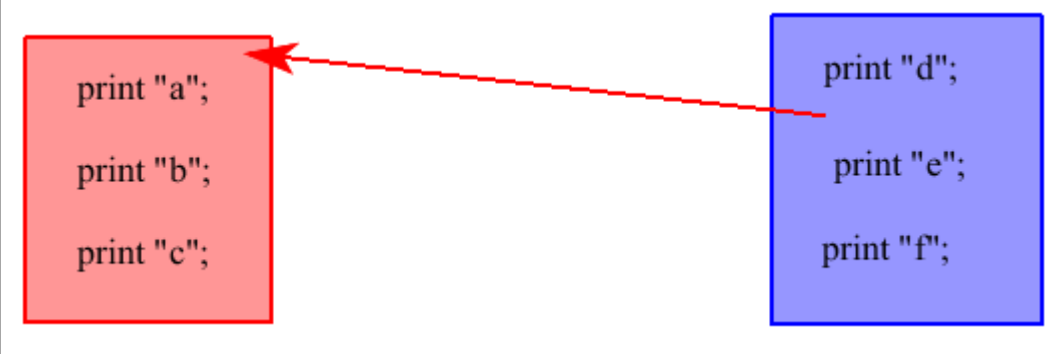
Design Justification. This format was less time consuming for the participants and allowed for rapid data collection and analysis; leading to iterative question design. Participation was more informal: no demographics were recorded and all that was asked was a single letter

answer on provided paper. Participants all completed the task within a couple minutes. The following section contains all questions and results from this follow up study.

8.2.2 Questions and Analysis

Iteration 1—Testing Conjecture 1. In order to test the conjecture concerning the positioning of the arrows the following question from the initial study (Section 3, Question 2) was given to one of the classes as a control. This question was chosen because it contained an arrow going from right to left, this removes the possibility that participants are ignoring the arrow completely.

Question: What would this pseudocode print?



Provided Options:

- A. **A B C D E F**
- B. **D A B C E F**
- C. A D E F B C
- D. **A B C D E F A B C**
- E. **A D B E C F**
- F. **D E F A B C**
- G. **A B C D A B C E F**

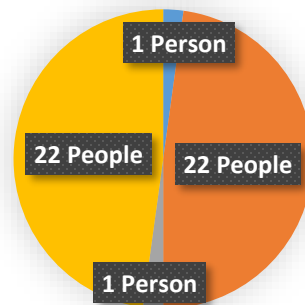


Figure 8.17: Question and results from control case in Iteration 1 of the follow-up study.

There were 47 participants for this question, one of whom answered “F or B” and is therefore not represented in the results. No participants choose options E, F or G. As can be seen the majority of opinion is equally split between two answers:

“D E F A B C” and “D A B C E F” with the latter being what the flow walker algorithm would

produce and the former ignoring the starting position of the arrow and processing the right-hand box completely before moving onto the left-hand box.

The following question was used to test our conjecture that arrow positioning was important. The origin of the arrows has been centred with respect to the surrounding lines and padding between those lines and the content has been increased.

Question: What would this pseudocode print?

```
print "a";  
print "b";  
print "c";
```

```
print "d";  
print "e";  
print "f";
```

Provided Options:

- A. **A B C D E F**
- B. **D A B C E F**
- C. A D E F B C
- D. **A B C D E F A B C**
- E. **A D B E C F**
- F. **D E F A B C**
- G. **A B C D A B C E F**

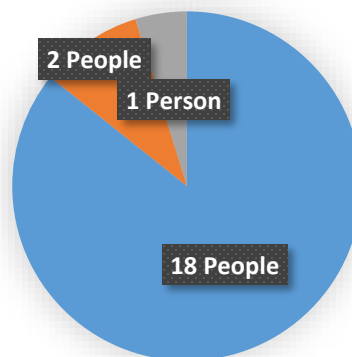
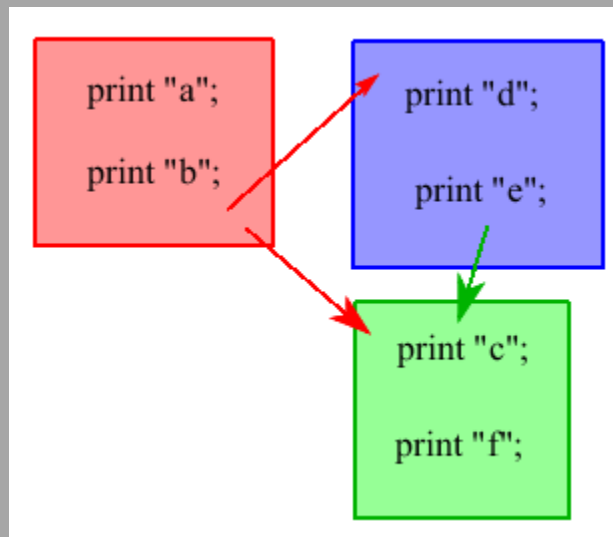


Figure 8.18: Question and results from test conjecture in iteration 1 of the follow-up study.

Shown to the other subject group—that had less than half the participants—we see a notable improvement in agreement with the algorithm. With 18 out of the 21 participants selecting answer B, agreement with the flow walker has risen from about 48% to 86%. Using a two-tailed Chi-Square statistical test we find this result to be statistically significant with a p-value of 0.0036017.

Iteration 2—Testing Conjecture 2. Conjecture 2 is concerned with the idea that people treat multiple boxes that are roughly aligned horizontally as being a part of the same column. The conjecture is that confusion is created when chaining conflicts with this perception. In order to test this the control question (see Figure 8.19) for this iteration is a duplicate from the initial study: Part 4, Question 4.

Question: What would this pseudocode print?



Provided Options:

- A. CABDEF
- B. ABDEABCF
- C. CDABEF
- D. ADECFB
- E. ABDECF
- F. DECFA B
- G. ACFDEB
- H. ABDECFCF

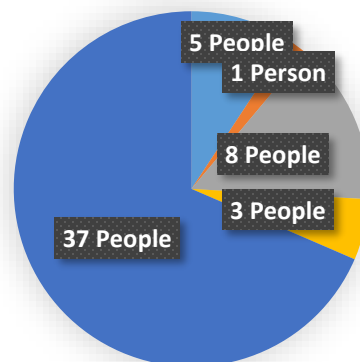
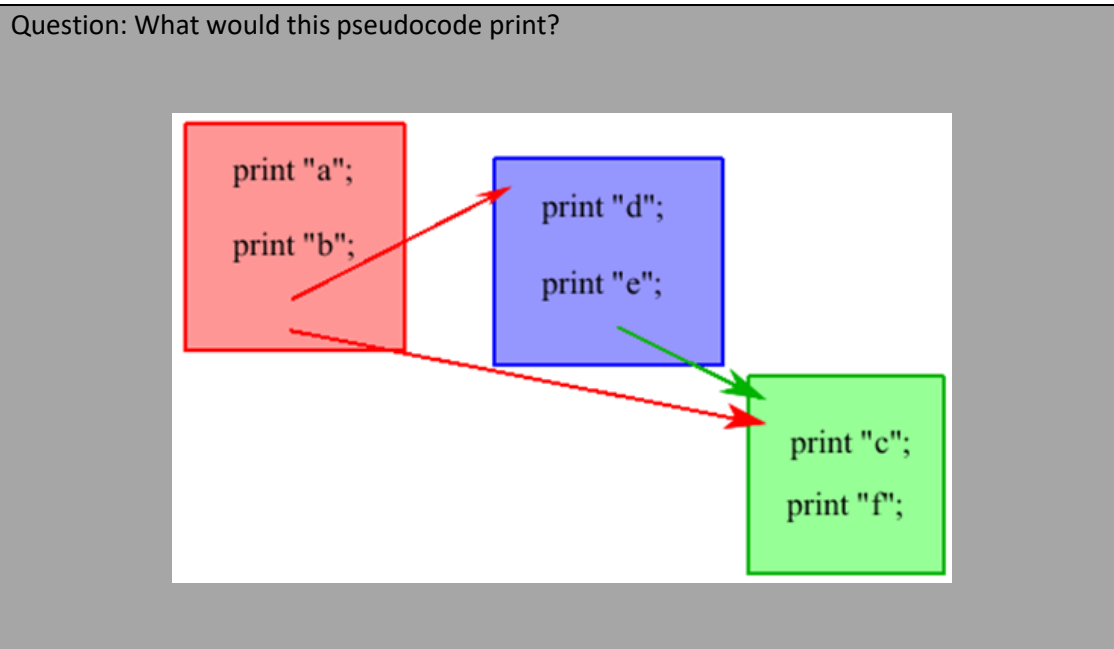


Figure 8.19: Question and results from control case in iteration 2 of the follow-up study.

There were 54 participants for this question and the result was quite surprising. In the initial study this question had 28% of respondents agree with what the flow walker would produce and in this follow up study that value went up to 69% which is much more in line

with the other results from Part 4 of the initial study, suggesting that the original results from this question were an anomaly. However it is still possible to build a similar question to test our conjecture; Figure 8.20 shows this question and the results obtained from it.



Provided Options:

- A. CABDEF
- B. ABDEABCF
- C. CDABEF
- D. ADECFB
- E. ABDECF
- F. DECFA B
- G. ACFDEB
- H. ABDECFCF

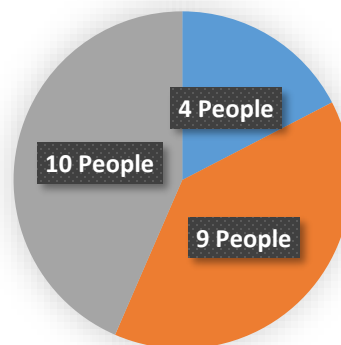
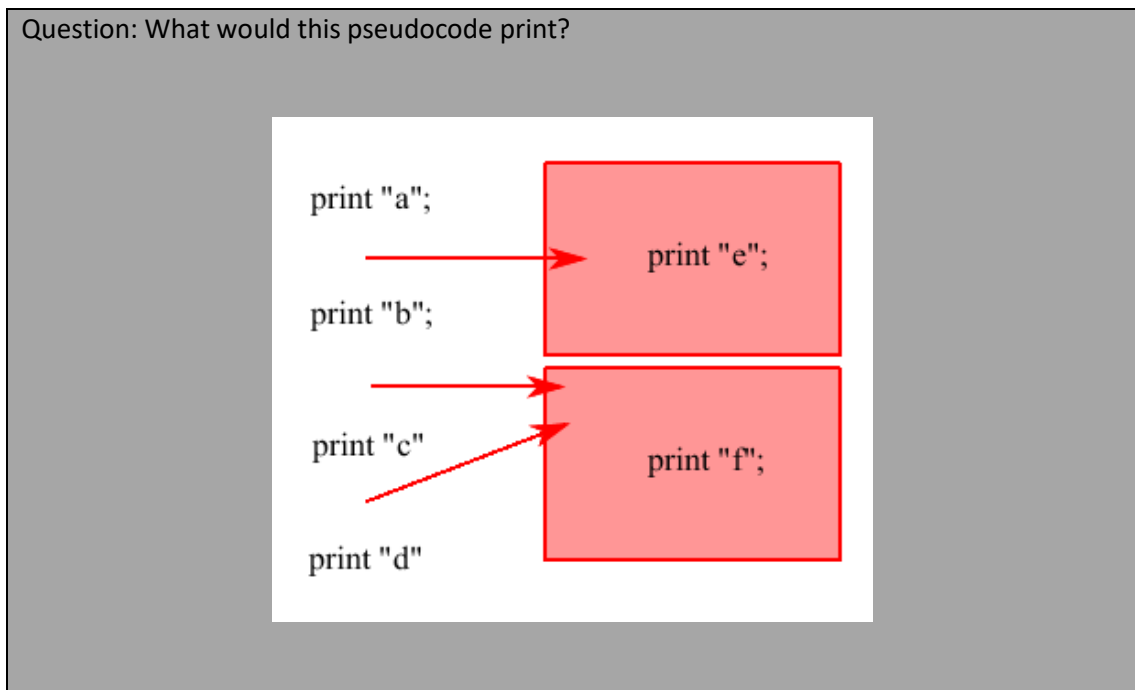


Figure 8.20: Question and results from test conjecture in Iteration 2 of the follow-up study.

In this version—which still produces the same answer as before—the boxes have been spread out in order to remove the possibility that participants will form columns out of multiple boxes. With 23 participants we received another surprising result, only 10 participants (43%) chose the option that agreed with the flow walker, still significantly better than the unmodified version of this question did in the Initial Study but also significantly worse than in the control case of this study.

Iteration 2—Refinement. The mixed results relating to Conjecture 2 received from the initial and follow up studies provided little reason to accept or dismiss the conjecture. A closer examination of the Initial Study shows that Question 4 of Part 4 is the only question that tests for understanding of reuse—having a box referenced by two (or more) arrows, causing the content of that box to be used again. In order to test if this is producing a confounding factor a new question was designed to test reuse without chaining. This question is seen in Figure 8.21.



Provided Options:

- A. **A** **B** **C** **D** **E** **F**
- B. **A** **B** **E** **C** **D** **F** **F**
- C. **A** **B** **E** **C** **D** **F**
- D. **A** **E** **B** **F** **C** **F** **D**
- E. **A** **E** **B** **C** **F** **D**
- F. **A** **E** **B** **C** **F** **F** **D**
- G. **A** **B** **E** **F** **C** **F** **D**

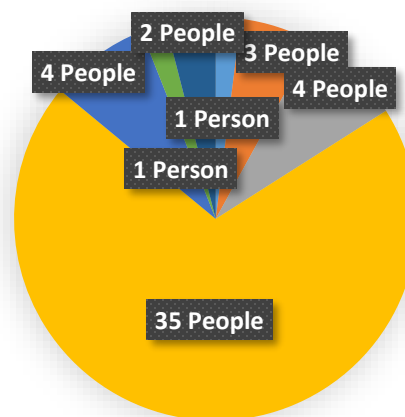


Figure 8.21: New question designed to test reuse.

This question had 50 respondents with 35 of those (70%) selecting the option that the algorithm would produce. This shows that a large majority of participants agree with the

flow walker on a basic case of reuse. While this does not definitively remove the possibility that reuse was causing confusion in the more complex example of chaining documented earlier, it does suggest that it cannot be the only aspect causing confusion.

8.3 Summary of Studies Evaluating SpIDER Spatial Layout

Overall the two studies discussed in this chapter produce positive results with respect to the understandability of SpIDER's flow walker algorithm. The initial study suggests that people's expectations roughly match (3/4 of the time) how SpIDER interprets code ordering when forming lines or using boxes. Taking the overall agreement of the first four parts of the initial quiz a confidence level can be calculated: 57.5% \pm 17.9% agreement with the algorithm, working with a 95% confidence level. It should be noted that initial reactions are being measured and simplified pseudocode is being used. Experience with the system and use of realistic code should help further. Parts 5 and 6 used more realistic pseudocode and included out of flow content; the result was higher agreement with the algorithm than seen when testing out of flow content with simpler pseudocode.

Once Out of Flow content was introduced, we saw a drop in agreement with the algorithm that only further continued when Chaining was tested. Evidence was present that suggested similar problems occurring in both sections. Two conjectures were formed and tested in a follow-up study. We found:

- Careful positioning of the arrow ends is important. A comparison between an out of flow example from the initial test and a modified version of the same example showed that, when the arrow was positioned carefully, agreement with the algorithm went from 48% to 86%. A statistical significance test shows this result to be significant with a p-value of 0.0036.
- We could not confirm that careful avoidance of confounding implied columns would increase agreement with the algorithm, or even that participants saw implied columns. The control case for the comparison test received 69% agreement with the algorithm, where the same question received only 28% agreement in the initial study.

Given the result about positioning arrows we are left having to decide how SpIDER can be adapted to encourage this behaviour. The ability to leverage Spatial Memory in a Spatial

Hypermedia system like SpIDER is assisted by the fact that content has been specifically positioned by the user (as discussed in Chapter 2). Therefore enforcing well-placed arrows via a snap-to system may be overall detrimental. Another option would be to provide people with a format function reached by a shortcut key that would reposition arrows in a local area (for example, inside the box that arrow is positioned inside when command is executed) so that they are centred between surrounding lines. As this method would only move arrows at user request it would hopefully lessen the damage to Spatial Memory. Finally a third option might be to consider well-placed arrows to be a '*Best Practise*' when using SpIDER; we expand on this idea in Section 10.2.

Chapter 9

Spatial Development Patterns

The benefit of well-produced code is manifold. Program code is written not only for the compiler to process, or for the programmer writing it at the time, it is also written for other programmers and even the original programmer at a later date, to be read, understood and maintained. In order to promote code clarity and quality, various Design Patterns [65, 66], Programming Methodologies [67, 68, 69] and Coding Conventions [70, 71] have been developed. Each of these tools in a programmer's metaphorical tool belt addresses a different facet of programming. Design Patterns establish a set of learnable and recognisable patterns for producing code, accelerating the code understanding process; Programming Methodologies provide guidelines (or strict rules) for the software development process to follow, keeping teams of programmers functioning as a well-oiled machine; and Coding Conventions provide a style guide for produced code, keeping it readable and more easily maintained.

Throughout this thesis, we have expressed a desire to allow programmers to use space to communicate information about their code. An overarching goal of this work is to explore how this might be harnessed to further promote code clarity and quality—another potential tool for the tool belt. In Chapter 7 (specifically Sections 7.1 and 7.4) we presented the flow walker, an algorithm that achieves the necessary step of converting spatially laid out code into a serialised form. Moreover, in Chapter 7, we established that there was a good level of agreement between the results of the flow walker algorithm and a person's intuition when interpreting spatially laid out code—a good sign in terms of code understandability. With the development of this algorithm, it has become possible to lay code out in more varied ways, providing new possibilities for communicating information about algorithms to

other programmers or the author's future self. More specifically, through SpIDER, programmers are able to spatially position and graphically enhance code with the intent of communicating additional information in ways not previously possible. When a programmer, or a team of programmers, using spatial layout establishes a specific configuration that they then repeatedly use, they gain a new vocabulary that exploits their Spatial Memory. We coin the term 'Spatial Development Pattern' to represent this phenomena.

We define a Spatial Development Pattern as:

Purposefully positioning software development artefacts (code, diagrams etc.) in space—either within the bounds of a Frame or virtually over the hyperspace created between multiple Frames—so as to enhance the code, providing supplementary information concerning its structure, history or programmer's motivation.

We chose the name Spatial Development Pattern to elicit thoughts of Software Design Patterns, Programming Methodologies and Coding Conventions. Using the phrase 'Spatial Programming Pattern' was considered, however, this fails to capture the fact that the flexibility and multimedia options provided by SpIDER mean that aspects of the entire development process can also be spatially captured, not just the act of writing code. For example, a requirements document can be embedded in situ with the code, or a history of debugging results may be included alongside the members it relates to.

While each of Software Design Patterns, Programming Methodologies and Coding Conventions assist by formalising a different facet of programming, they are only able to work due to consensus. Whether it be academic institutions spreading knowledge of the Factory Pattern [66], company management insisting on a specific documentation style or a group of programmers working out of a garage deciding to use Scrum [68], the usefulness of these tools is determined by their adoption. We continue this trend by presenting examples of Spatial Development Patterns driven by the capabilities of SpIDER, which we further categorise as either Expressive Patterns or Process Patterns. On this point, we note that we do not presume to describe a complete list of Spatial Development Patterns in this chapter. Instead, we take the reader on a journey, showcasing the potential of Spatial Development

Patterns and hopefully evoking the reader’s own imagination. We document a set of broad ideas that have resulted from brainstorming and use of the SpIDER prototype.

We begin in Section 9.1 with a review of programming capabilities in traditional IDEs aimed at embodying the sentiment ‘First, do no harm’. As SpIDER is unlike any previously developed IDE, we wish to establish that conventional IDE behaviour and functionality is either retained or can be closely approximated. Having established that no expressiveness has been lost, we then transition to examining the added potential that spatial layout provides. In Section 9.2 we present a series of ‘Expressive Patterns’. Expressive Patterns use spatial layout to enhance the presentation of software in ways that illustrate the programmer’s view of the structure. Section 9.3 presents a series of Process Patterns. Also utilising spatial layout, Process Patterns allow a programmer to capture the steps taken to achieve a goal, such as documenting debug history or otherwise communicate the motivation behind the sequence of steps taken.

9.1 Maintaining Existing Functionality

The transition from a traditional IDE to SpIDER adds numerous possibilities for improving the software development process. However, just as exploring these opportunities is important, so too is establishing that existing forms of expression have not been lost. We show this through a series of examples, comparing screenshots from Eclipse—our representative traditional IDE—with those from SpIDER. Each example is intended to show the equivalent/minimal transformation required to transition from a traditional IDE to a Spatial Hypermedia IDE. In other words, rather than making full use of the novel functionality provided by SpIDER (a task undertaken later in the chapter), we first establish that functionality from traditional IDEs has been retained.

Traditional IDEs such as Eclipse not only provide programmers with the ability to browse and navigate around their code, they also support programmers with integrated processes to help them produce code more efficiently. Examples of such functionality include syntax highlighting, problem notification, content assist, step by step debugging and GUI building. In designing and building SpIDER, we have used Eclipse JDT. This allows us to access the core functionality provided by Eclipse. In effect, we are able to acquire the underlying data structures—for example, the results of a content assist request stored in a data structure, as opposed to a visually displayed GUI panel—and make it accessible in SpIDER. Therefore, the

task of preserving traditional IDE functionality in SpIDER is a straightforward two-step process. First, a ‘communication layer’ is established for SpIDER to query Eclipse JDT. Secondly, a set of appropriate transformations for a variety of Eclipse GUI panels must be implemented in SpIDER. To fit into SpIDER’s worldview, these transformations need to use absolute positioning and will be more malleable for future use if they are built out of existing Expedite components. To this end, we have ported functionality such as content assist, debugging and more to SpIDER.

Instead of documenting each piece of implemented functionality, we document how specific ‘building blocks’ are translated from a traditional IDE to SpIDER. This approach has the added benefit of providing insight into how functionality not currently in the SpIDER prototype may be translated. We will begin in Section 9.1.1 by demonstrating how the relative authoring panel/area for editing code featured in traditional IDEs can be translated to absolute authoring in SpIDER—see Sections 4.1 and 4.2 for the distinction between relative and absolute authoring. Section 9.1.2 addresses hierarchical content layout, a technique frequently used by traditional IDEs to represent tree structures, and how it can be translated to SpIDER through the use of the linking system. Section 9.1.3 then shows how the effect of scrollable content can be approximated with pagination in SpIDER. Both Sections 9.1.2 and 9.1.3 both use the panels from Eclipse that surround the text area to make their point, however, in SpIDER, the concepts expressed also apply to authored code. Section 9.1.4 then moves on to looking at translations for these panels by documenting how buttons and hyperlinked labels can be represented in SpIDER. Finally, Section 9.1.5 documents how overlaid feedback, typically expressed as tooltips in traditional IDEs, can be implemented in SpIDER.

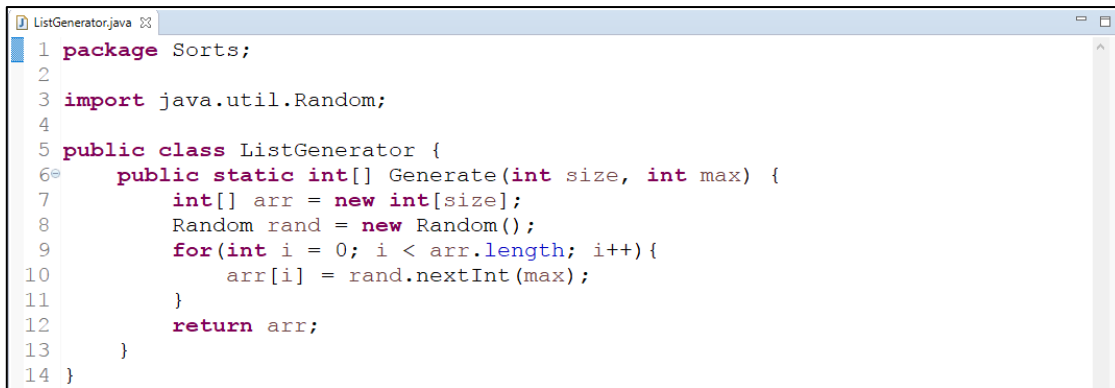
9.1.1 Authoring

In comparison to a relative authoring environment, the introduction of the flow walker in SpIDER provides more degrees of freedom to the programmer when producing code. We will demonstrate some of these new possibilities in subsequent sections of this chapter. However, in keeping with our goal of establishing that no functionality has been lost, we first must establish that one of these degrees of freedom matches the code authoring that traditional IDE environments provide.

Figures 9.1 and 9.2 respectively show a comparison of authoring in the relative IDE Eclipse with the absolute IDE SpIDER. Identical Java code is produced in each environment. When coding in a relative authoring environment, programmers make use of whitespace characters to organise their content. Indentation is used to communicate containment and frequently matches the programming language's notion of scope. Blank lines can be used to provide logical boundaries between multiple steps within the same function. As we have seen in Section 3.2, programmers frequently make use of blank lines to visually separate code within a single function. Our analysis of 14,239 Java projects showed that 35% of functions have at least one blank line, and that, given a function with at least one blank lines, multiple blank lines were present (an average of 4.5 blank lines per function, over all functions).

In SpIDER and other absolute position authoring applications, it is not necessary to explicitly represent white space through keyboard entry, as it can always be emulated. The absolute positioning of text allows the author to visually place tokens as though their position had been determined by whitespace. For example, Line 6 of the code shown in Figure 9.1 is indented one level. In the translation to SpIDER, shown in Figure 9.2, this line is simply spatially positioned further to the right. This achieves the same visual distinction. A similar translation can occur to emulate a blank line by spatially positioning text vertically.

Acknowledging the inconvenience of switching between keyboard typing and mouse positioning to achieve appropriate layout, SpIDER has been programmed to react to TAB and ENTER keys to respectively emulate indentation and line-breaks. If the cursor is hovering over a Text Item (program language token) and the TAB key is pressed, then SpIDER will use the magnet system (described in Section 7.5) to collect all Text Items belonging to the specified line and move them a set distance to the right. If the ENTER key is pressed, then SpIDER will use the magnet system to reposition the cursor as if it is starting a new line below the line containing the selected item.



```
1 package Sorts;
2
3 import java.util.Random;
4
5 public class ListGenerator {
6     public static int[] Generate(int size, int max) {
7         int[] arr = new int[size];
8         Random rand = new Random();
9         for(int i = 0; i < arr.length; i++){
10             arr[i] = rand.nextInt(max);
11         }
12         return arr;
13     }
14 }
```

Figure 9.1: Translating between relative and absolute positioning for authored code—Relative.

```
package Sorts ;
import java. util. Random ;
public class ListGenerator {
    public static int [ ] Generate ( int size , int max ) {
        int [ ] arr = new int [ size ] ;
        Random rand = new Random ( ) ;
        for ( int i = 0 ; i < arr. length ; i ++ ) {
            arr [ i ] = rand. nextInt ( max ) ;
        }
        return arr ;
    }
}
```

Figure 9.2: Translating between relative and absolute positioning for authored code—Absolute.

9.1.2 Hierarchical Content

In order to visually represent the hierarchical structure of software projects and aspects within them, traditional IDEs such as Eclipse make frequent use of tree widgets. One of the more complex and prominent panels in Eclipse is the Package Explorer. The Package Explorer provides a list of all the software projects in the current workspace. Each of these projects contains a hierarchically organised collection of labels associated with files in that project. The Package Explorer can be used to browse and navigate to and between these files.

Figure 9.3 shows a screenshot of the Eclipse Package Explorer with a single project and its content listed. This project contains a single Java package named 'javacodeanalysis' which

in turn contains five Java source files. Several additional files and folders—containing information used by Eclipse to build the projects—are also contained within the project. A simple translation of this setup into SpIDER uses its hyperlinking system and can be seen in Figures 9.4 and 9.5.

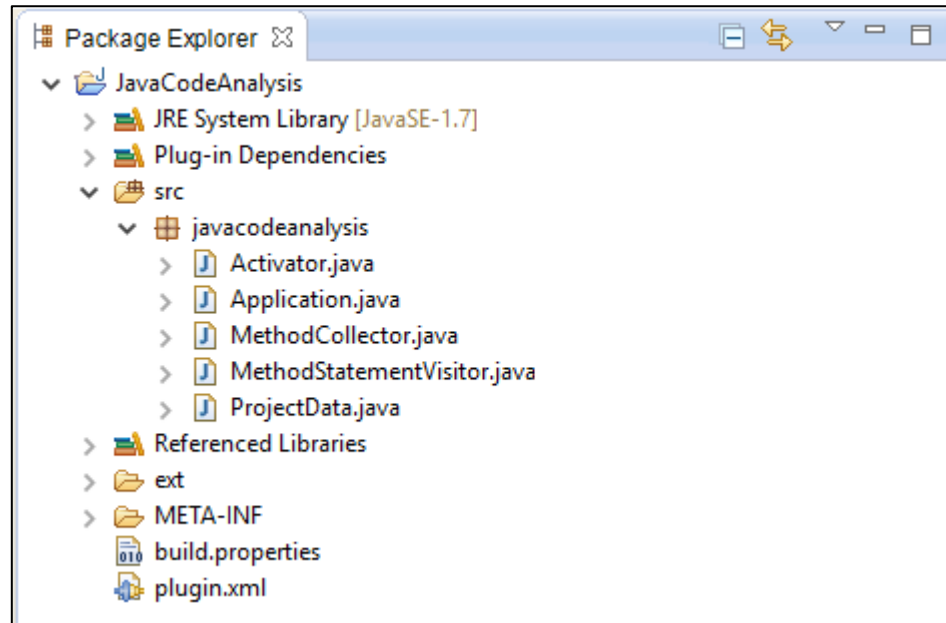


Figure 9.3: Translating the file system of a small project from a traditional IDE to SpIDER—Traditional.

Figure 9.4 shows a screenshot of the Project Frame as it relates to our example. The title of the Frame is set to emphasise its role in the hierarchy. Beneath the controls, a list of Text Items represents the top level view of the project. At the head of the list is a Text Item linking to the package 'javacodeanalysis'. Clicking on this Text Item will navigate the programmer into the package, shown in Figure 9.5. On this resulting Frame, the programmer is presented with the list of classes contained within the package. Clicking on any of the Text Items making up this list of classes will navigate the programmer inside the appropriate class so that they may resume programming.

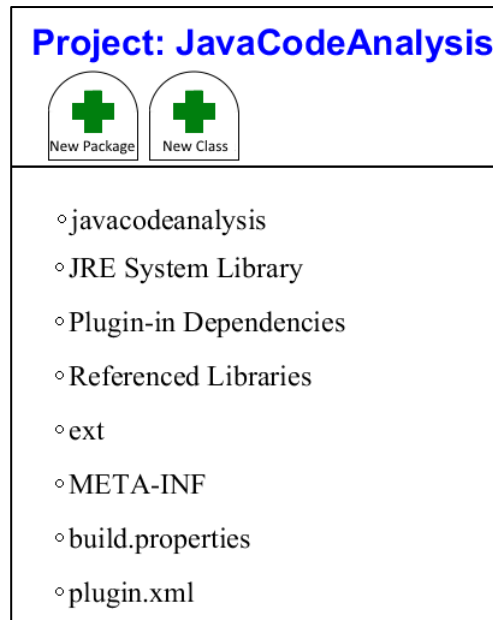


Figure 9.4: Translating the file system of a small project from a traditional IDE to SpIDER—SpIDER Project Frame.

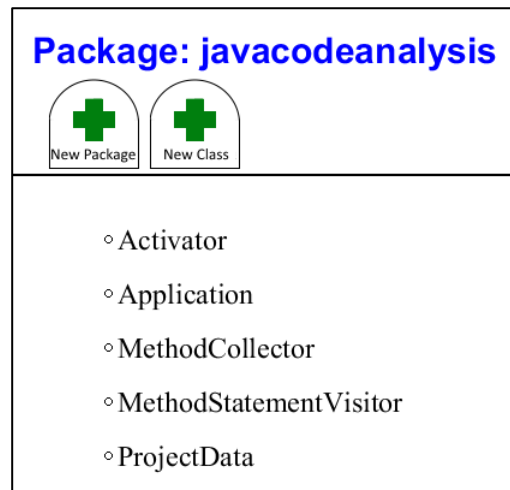


Figure 9.5: Translating the file system of a small project from a traditional IDE to SpIDER—SpIDER Package Frame.

9.1.3 Pagination

The example translation given in the previous section was modest in size and not representative of a more significant software project. In Figure 9.6 we present a screenshot of the Package Explorer with the Expeditee source code loaded into Eclipse. Eclipse handles a large number of packages by using a scrollable panel. Matching this in SpIDER by adding a scrollable panel would be in conflict with our decision to use a fixed sized spatial interface (see Section 5.3), so seeking another solution is preferable. Ultimately, if we were to

attempt to fully utilise the spatial layout capabilities of SpIDER, we might choose to arrange packages in some spatially significant way—for example, as a diagram showing the structure of the project (as seen in Section 9.2.1). However, sticking with the theme of the section, we are looking for the minimum required change to translate from a relative IDE such as Eclipse to an absolute IDE such as SpIDER. The most straightforward automatable layout approach is to use pagination.

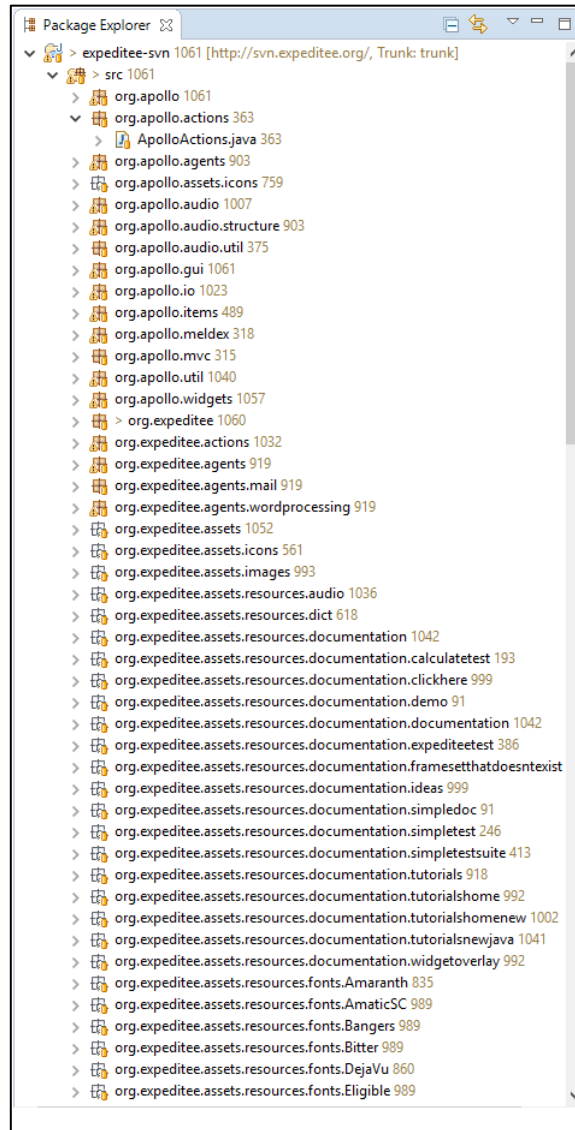


Figure 9.6: Adapting a large amount of information from a traditional IDE to SpIDER—Traditional.

The result of the translation from Eclipse to SpIDER is split over multiple Frames. The first Frame is shown in Figure 9.7 and the second in Figure 9.8. Starting with Figure 9.7, as with our previous example that used a smaller Java project, the packages are represented by a

list Text Items, each of which has a hyperlink leading to a Frame representing the inside of the appropriate package. Below the final Text Item in the list of packages is an additional Text Item with content “//Next”, linking to the Frame seen in Figure 9.8.

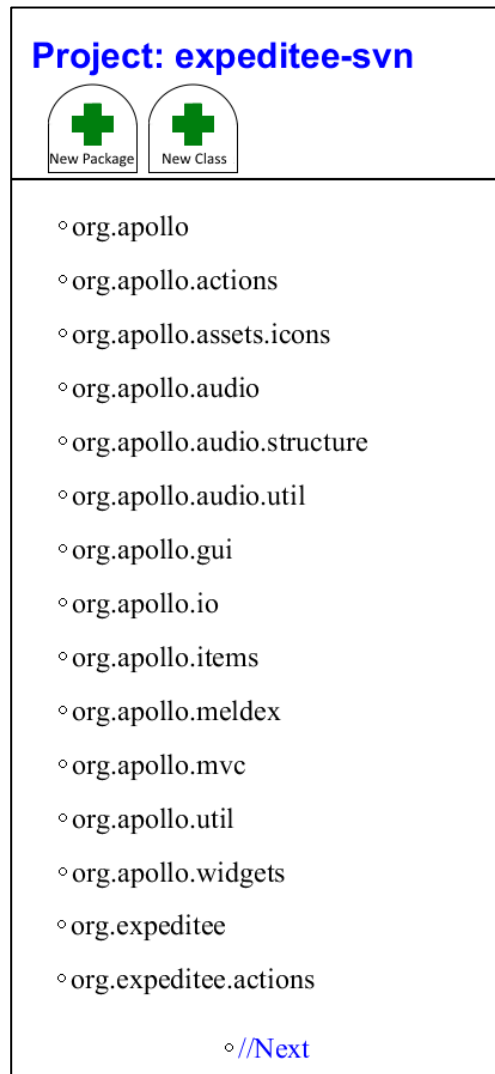


Figure 9.7: Adapting a large amount of information from a traditional IDE to SpIDER —SpIDER Project Frame.

Notice the title and controls present on the Frame in Figure 9.8. We can see that instead of being a package frame, it is still a project frame—an important distinction to be made as any packages created here are not sub-packages of an existing package. At the end of the list of packages, links exist to the next and previous Frame. This pagination process is repeated until all packages have been listed. In order to list all 102 Expeditee packages in this fashion, seven Frames are required.

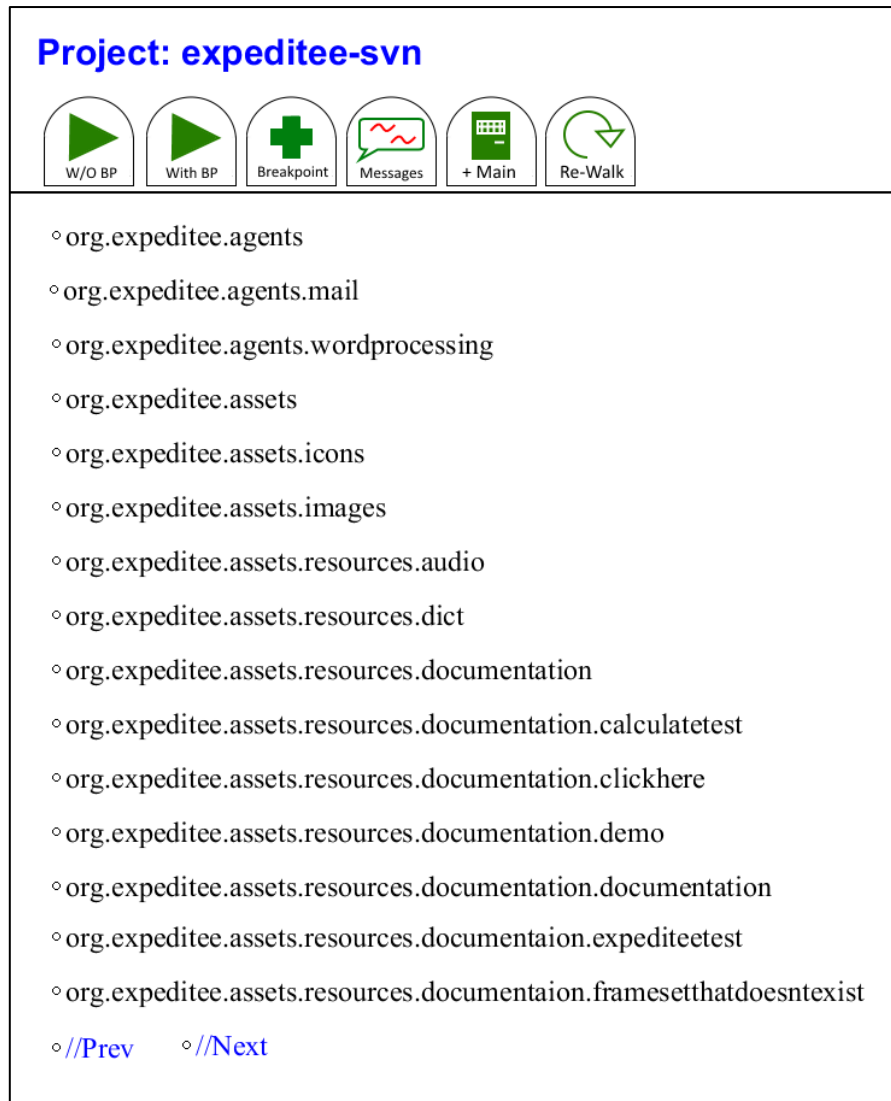


Figure 9.8: Adapting a large amount of information from a traditional IDE to SpIDER —SpIDER Project Frame cont.

9.1.4 Controls

Traditional IDEs such as Eclipse create and logically position controls for performing common actions. Frequently used buttons—such as those for running the current application—are prominently positioned. Furthermore, the state of Eclipse at any given time determines what controls are visible and where they are placed. These controls are arranged into several purpose-built panels flanking the main authoring area.

SpIDER ships with a pre-bundled set of controls that operate in an equivalent manner. These controls provide access to a selection of imported JDT functionality. Figures 9.9 and 9.10 compare some of the controls in Eclipse and SpIDER respectively. Both applications are

in the middle of a debugging session—halted at a breakpoint—and as such are displaying a specific set of controls. When activated, these buttons (and interactive labels) adjust the state of their environment. For example, both environments contain buttons for stepping through code and activating these buttons will execute some number of instructions.

These controls in SpIDER are produced by associating a Text Item or image (as they are in Figure 9.10) with a link or action—see Section 6.4.3 for information on actions. In a similar fashion to Eclipse, the controls visible at any given time are determined by the current state of SpIDER.

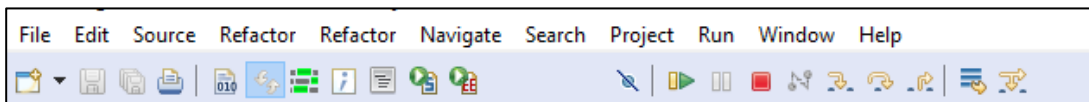


Figure 9.9: Controls in Eclipse.

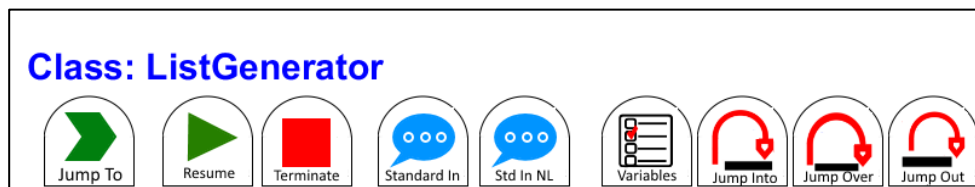


Figure 9.10: Controls in SpIDER.

9.1.5 Overlaid Feedback

Some user actions require the IDE to respond by doing more than updating the state of the environment. To this end, Eclipse (and other traditional IDEs) often use tooltips to provide feedback. One such example is Content Assist. In Eclipse, as the author is typing, a list of ‘completion suggestions’ may appear—overlaid on top of the authoring area. Tokens prior to the cursor position are used as context for filtering the completion suggestions. The programmer is then able to use these suggestions to complete the token they are typing. Figure 9.11 shows the source code of the ListGenerator class, produced in Eclipse. The programmer is currently part way through a content assist request on Line 10. Having so far typed “rand.next”, the list of completion suggestions is populated by public members from the Java library class ‘Random’ that start with “next”.

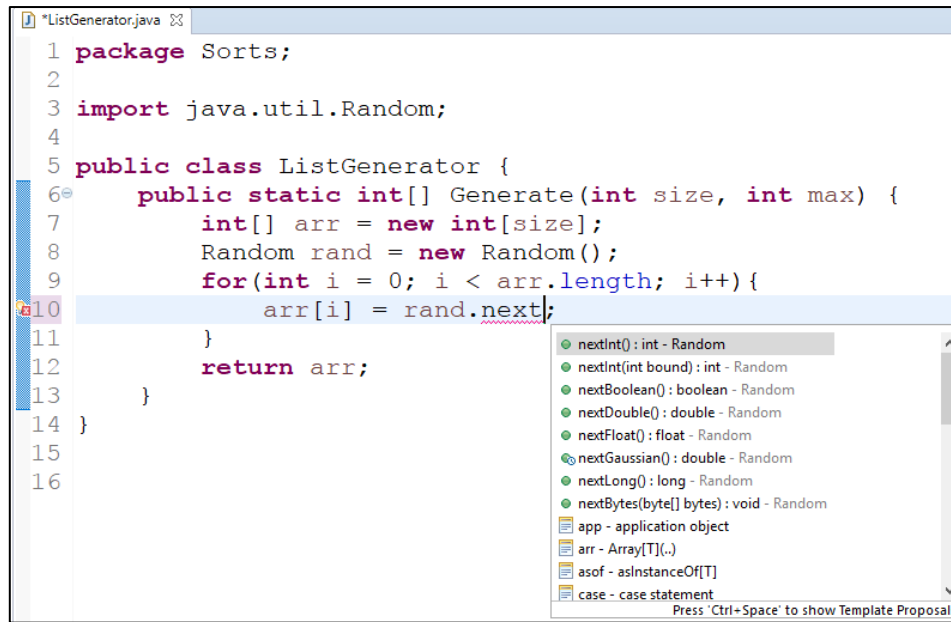


Figure 9.11: Translating Content Assist (a form of overlaid feedback) from a traditional IDE to SpIDER—Traditional.

In Figure 9.12, the same code and Content Assist request have been reproduced in SpIDER. In order to make a Content Assist request in SpIDER, the programmer has used the key combination CTRL + SPACE. Unlike the results of a content assist request produced by Eclipse, which is ephemeral, SpIDER builds the results out of Text Items and places them in an annotated box. This annotated box is then attached to the cursor as if the programmer had picked it up. This allows the programmer to spatially position the results of the content assist request and subsequently manipulate them just as any other set of Text Items. Alternatively, the programmer may decide to dismiss the content assist by pressing the middle and right mouse buttons together (the operation for delete).

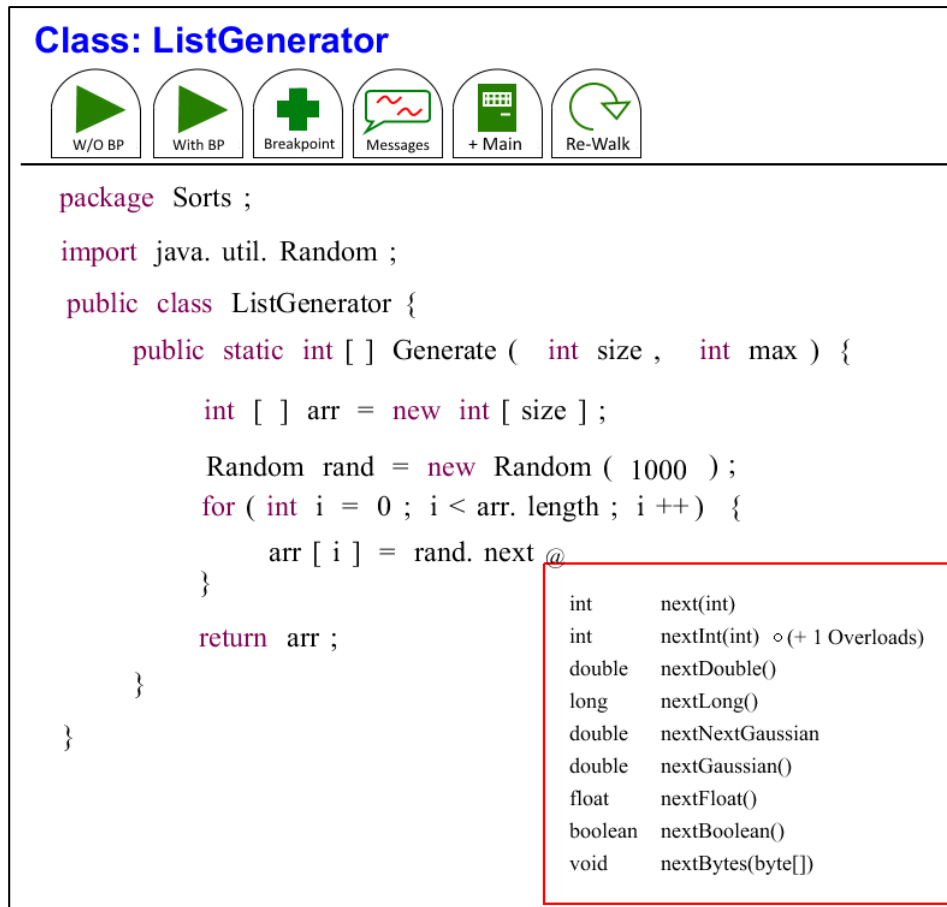


Figure 9.12: Translating Content Assist (a form of overlaid feedback) from a traditional IDE to SpIDER—SpIDER.

9.2 Expressive Patterns

A programmer who arranges content using space—either within a Frame or across the hyperspace between Frames—with the intent of communicating auxiliary information about that content is building an Expressive Pattern. In the wider context, an Expressive Pattern is a form of Spatial Metaphor. We start this section by detailing this broader connection, before going on to document multiple categories of Expressive Patterns. More specifically, in Section 9.2.1 we will show how documentation can be embedded in situ with program artefacts. Section 9.2.2 will then explore the idea of using hyperspace to escape the hierarchical structure traditionally used to organise software projects. Sections 9.2.3 and 9.2.4 will provide examples of using spatial positioning to emphasise and logically group code fragments respectively. Finally, Sections 9.2.5 and 9.2.6 will examine how the flow walkers ‘out of flow’ behaviour can be used to segregate part of a code fragment.

Spatial Metaphor. A Spatial Metaphor is a mechanism that, when employed, allows a person to gain knowledge concerning a non-spatial element through its spatiality [54]. An example from everyday life can be seen during a trip to the supermarket. The juxtaposition of a price tag and loaf of bread on a shelf can be used to imply the price of that bread. Computer user interfaces also make use of Spatial Metaphors. For instance, in Microsoft Word, the close proximity and grouping of controls for creating bullet points and numbered lists show that they are functionally related—both are used to create different forms of lists.

Sometimes in HCI, a more specific name is used to communicate the use of a Spatial Metaphor, such as in [72] where Greenburg and Roseman use the phrase Room Metaphor in place of Spatial Metaphor to clarify that their work on collaboration attempts to treat a user interface as a specific space—a room. Our use of the term Expressive Pattern can also be viewed in this way.

User interfaces presented earlier in the thesis, for the purpose of explaining aspects of Spatial Memory and Spatial Layout (Chapters 2, 4 and 5), can also be framed in terms of their Spatial Metaphor. Moreover, we can distinguish between two different types of Spatial Metaphors used in such user interfaces as follows.

- **Tailored Spatial Metaphor.** When developing a user interface, the designer may wish to promote a specific type of Spatial Metaphor. Implied in this case is that the goal is to produce a high-quality Spatial Metaphor. Because a specific Spatial Metaphor is being developed, interactions with it can be tailored to lead the user towards the end goal.

- **Unconstrained Spatial Metaphor.** Alternatively, the user interface designer may wish to provide users with the ability to create their own Spatial Metaphors. The goal, in this case, is not to produce a specific high-quality Spatial Metaphor, but rather to provide the users with a framework and appropriate tools with which to create their own. This can be achieved by developing and subsequently providing users with the tools needed to build Spatial Metaphors. It is worth noting that this case is only suitable for authoring applications, such as programming in an IDE. A corollary of this is that the end user is unlikely to be interested in creating a high-quality Spatial Metaphor, instead, they will be trying to create an acceptable and suitable Spatial Metaphor in as little time as possible, without distracting from their primary task.

CommandMaps [15], Space Filling Thumbnails [17], Code Thumbnails [12], Code Canvas [51] and Code Bubbles [52, 53] are all examples of applications using Tailored Spatial Metaphors. All of these applications are either not authoring applications—CommandMaps, Space Filling Thumbnails and Code Thumbnails—or specifically prescribe what authored content can be contained within a spatially positionable container. For example, a container in Code Canvas can contain any content that would otherwise be presented in a Microsoft Visual Studio tab: this includes, code files or images but not a combination of these or code snippets.

Of the previously reviewed applications, VIKI [36, 37, 42], VKB [38, 39, 40] and Expeditee [41, 50, 46, 47] all belong in the Unconstrained Spatial Metaphor category. They have all provided tools for users to create Spatial Metaphors. For example, the Collections, Composites and Objects in VKB support a variety of media and do not impose restrictions on the content placed within them.

Boxer [73], developed by diSessa and Abelson, is another example of an application utilising unconstrained Spatial Metaphors. Seeking to develop an accessible programming language they produce Boxer, an application that we would classify as Spatial Hypermedia, and that intertwines visual and spatial properties with a programming language to allow programmers to express containment, scope and cross-cutting access as well as visually

inspect the results of functions. The spatially positionable container in Boxer is the Box, which is very similar to Objects from VIKI and VKB.

While developed independently, in retrospect, we can classify an Expressive Pattern as an Unconstrained Spatial Metaphor. An Expressive Pattern in SpIDER is an instance of a Spatial Metaphor, designed to communicate additional information about the produced code and made possible by SpIDER's flow walker. The tools provided by SpIDER for users to produce Expressive Patterns differ from those provided by VIKI, VKB and Boxer by focusing on representing the flow of content rather than its position and containment in the information space. Also, differentiating SpIDER from Expeditee is the fact that Expeditee, as a general authoring environment, is not required to serialise produced content. SpIDER is required to do this so that authored program code can be compiled, ran and debugged.

9.2.1 In Situ Documentation

It is not difficult to conceptualise scenarios where more integration between documentation and code would be useful. Programmers often produce documents—as a result of planning—that describe the structure of their software projects. To use a Spatial Metaphor, functionality to lessen the distance between these planning documents and code in a traditional IDE is limited. A typical solution may be to position a comment containing a link to the planning documents in a relevant place in the code. Applications such as Code Canvas [51] allow images to be included, in their own container, side-by-side with relevant containers of code. This functionality could be used as a step in the right direction, lessening that distance by including the documentation as an image. SpIDER's multimedia support also makes this possible, but it can go further, integrating documentation with code.

Figure 9.13 one way in which Expeditee's source code could be arranged in SpIDER. The Frame seen in this figure is the top-level Frame for the Expeditee code base in SpIDER. Spatial layout, colour and images are used to communicate information about the structure of the project, similar to a UML Component Diagram:

- There are two primary packages in Expeditee: *org.expeditee*, containing the base Expeditee source files and *org.apollo*, containing the code for an extension to Expeditee that adds hypermedia tools for authoring and editing music. These packages are each represented by a large red box. Within these red boxes are Text Items, each linking to either a class or sub-package.

At a glance, it can be seen that the red box surrounding the Apollo code is smaller than that surrounding the Expeditee code. The proportions of the boxes have been deliberately chosen by the programmer to indicate where the majority of the code lies.

- Images are used to provide redundancy and reinforce which red box is associated with each of the previously mentioned packages. Rather than force a programmer to read the label associated with the red box, the pictures can be used to achieve the same association. The image in the left box is the icon for Expeditee whereas the image in the right box is the icon for Apollo. These images, due to their prominence and proximity to related content are suitable candidates for landmarks.
- As previously stated, within the red box there are several Text Items which lead to further content, most of which are surrounded by additional boxes.
 - Text Items without an additional surrounding box represent classes. When clicked, the programmer is navigated inside the class and is able to view, author or edit code.
 - Text Items with an additional surrounding box represent sub-packages. When clicked, the programmer is navigated into that package.
- Colour is used to group packages together by related function.
 - Packages with a blue surrounding rectangle contain functionality concerned with authoring and running user scripts.
 - Packages with a green surrounding rectangle contain functionality concerning with IO, both local file system IO and network IO.

- Packages with a yellow surrounding rectangle contain functionality that effects visual aspects of the GUI.
- Packages with a pink surrounding rectangle contain resources that Expeditee and Apollo require—such as icons.
- Packages with a white surrounding rectangle and black outline contain packages concerned with user settings and auxiliary functionality.
- Packages with a blue surrounding rectangle and red outline are only featured in the Apollo section of the project layout and contain functionality concerned with providing audio authoring.

This solution has allowed the programmer to fully integrate code artefacts—in this case packages and classes—with some high-level aspects of documentation. This has allowed these program artefacts to provide a dual purpose, functionally supporting the authoring of the *expeditee-svn* software project while at the same time providing future programmers with information on its structure.

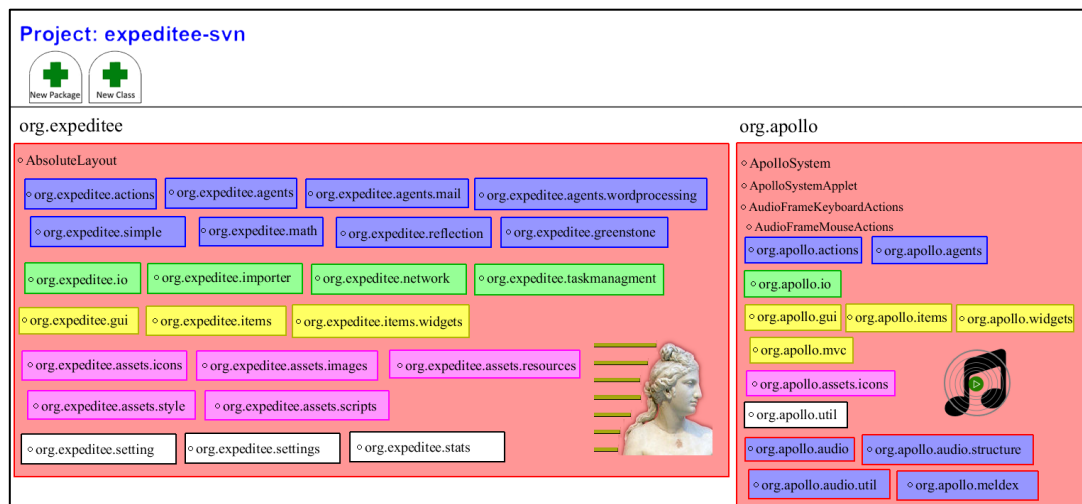


Figure 9.13: Spatially arranging the package structure of *Expeditee* in *SpIDER* so as to document additional information.

9.2.2 Escaping Hierarchical Structure

The Frame and Linking system in *SpIDER* provides programmers with the ability to segment code across multiple Frames. To review, as can be seen in Figure 9.14, a single function may be broken into component parts, spread across multiple Frames, each part represented by a Text Item linking to one of these Frames. This creates a hierarchical structure. However,

links can also be used to express cross-cutting relationships—escaping hierarchical structures.

```

public boolean isValidMove
( GamePiece piece , Point destination ) {
    final GamePiece. Colour team = piece. getTeam ( ) ;

    ◦//destination cannot contain piece where piece.getTeam() == team, return false on fail ↑

    ◦//delta of move must be valid for piece type, return false on fail ↑

    if ( ! ( piece instanceof GamePiece. Knight ) )
        ◦//path much be clear, return false on fail ↑

    GameBoard copy = board. clone ( ) ;
    copy. update ( piece , piece. moveTo ( destination ) ) ;

    ◦//team' king on copy cannot be in check, return false on fail ↑

    return true ;
}

```

Figure 9.14: A function split into component parts in SpIDER, expressed with hierarchical structure.

Consider a scenario where a programmer has produced the above code to validate a specific chess move. They now wish to produce a test function to help ensure their code functions as expected. To achieve this, the programmer first creates a new top-level package to hold test classes and a *GameStateTest* class for holding tests concerning the current game state. This is followed by writing the code for generating a wide range of possible moves and testing them. If the programmer now wished to navigate between the *isValidMove* function and the *isValidMoveTest* function they would currently have to navigate up the hierarchy to the top level, causing navigation over at least three Frames, followed by navigation down a

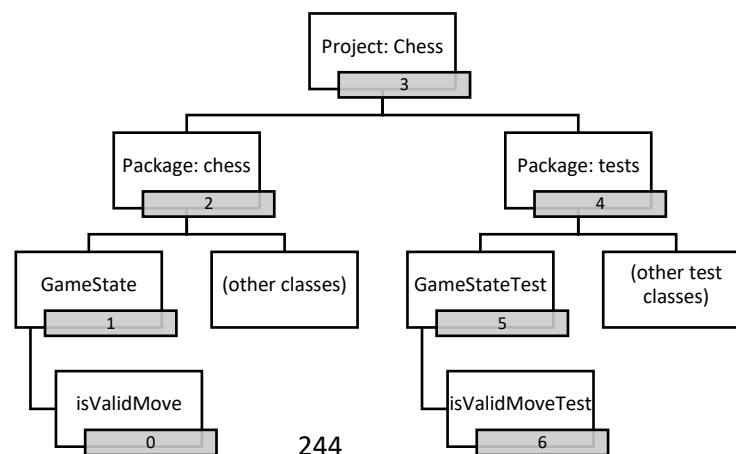


Figure 9.15: Hierarchical representation of navigation actions between two functions of different classes.

different branch, causing navigation over at least another three Frames. This navigation is represented pictorially in Figure 9.15. If it becomes apparent that *isValidMove* is a particularly problematic function, then the programmer may find themselves making this trip multiple times.

In order to circumvent this process, the programmer is able to create some controls with cross-cutting links. Figure 9.16 shows the Frame with the *isValidMove* function. Figure 9.17 shows the Frame with the *isValidMoveTest* function. Prominently positioned on each of these Frames, at the top, is a link to the other. These links allow the programmer to quickly navigate between these two Frames, making it unnecessary to perform the previously mentioned, comparatively expensive, navigation operation. It is worth noting that the programmer is free to use cross-cutting links to communicate association and accelerate navigation between any two Frames as they see fit.

```

◦@Go To Test Case
public boolean isValidMove
( GamePiece piece , Point destination ) {
    final GamePiece. Colour team = piece. getTeam ( ) ;

    ◦//destination cannot contain piece where piece.getTeam() == team, return false on fail
    ◦//delta of move must be valid for piece type, return false on fail
    if ( ! ( piece instanceof GamePiece. Knight ) )
        ◦//path much be clear, return false on fail
    GameBoard copy = board. clone ( ) ;
    copy. update ( piece , piece. moveTo ( destination ) ) ;
    ◦//team' king on copy cannot be in check, return false on fail
    return true ;
}

```

Figure 9.16: The *isValidMove* function with a cross cutting link to *isValidMoveTest*.

```

◦@Go To Implementation
public boolean isValidMoveTest ( ) {
    Random rand = new Random ( ) ;

    ◦//generate 10 tests that should pass
    ◦//run should pass tests, return false if any fail
    ◦//generate 10 tests that should fail
    ◦//run should fail tests, return false if any pass

    return true ;
}

```

Figure 9.17: The *isValidMoveTest* function with a cross-cutting link to *IsValidMove*.

9.2.3 Emphasis

Figure 9.18 shows a fragment of a personal contacts application that has been produced in SpIDER. The particular function we have chosen to focus on searches through a data structure of contacts, looking for those whose hometown is the specified location. The upper half of the function performs some argument checking and pre-processing—a necessary but mundane step—the lower half of the function performs of search and produces output.

```
public void printContactByArea ( String area ) {  
    if ( area != null && isValidAreaName ( area ) ) {           //Our datastructures expect  
                                                                //lowercase parameters  
        String normalisedAreaName = area.toLowerCase ( ) ;  
        List < String > towns = findTownsInArea ( normalisedAreaName ) ;  
        if ( contacts == null ) initialiseContacts ( ) ;  
        for ( Contact c : contacts ) {  
            if ( towns.contains ( c.hometown ) )  
                System.out.println ( "Found " + c.name + " with home town in " + area ) ;  
        }  
    }  
}
```

Figure 9.18: A function to search a list of contacts and subsequently print those living in a specified area—no emphasis present.

In maintaining this function, a programmer may decide to emphasise the important aspects of the code: the search, output and documentation. Figure 9.19 shows the same content seen in Figure 9.18 but with emphasising applied to the code deemed important. The for-loop and its subordinate code that is used to search through the *contacts* data structure is now contained within in a large red box; that is in turn further enclosed by a yellow box to exaggerate the loops prominence on the Frame. Documentation, in the upper-right corner—that has been left as a note for other programmers—has also been placed in a box. In contrast, the code responsible for performing the required checks and pre-processing has been left unchanged. An additional step may be to move the code responsible for this checking onto another Frame, substituting a single link, further de-emphasizing the mundane portions of the code. This is a process we refer to as Lightweight Abstraction and is expanded upon in Section 9.2.5.

```

public void printContactByArea ( String area ) {
    if ( area != null && isValidAreaName ( area ) ) {
        String normalisedAreaName = area.toLowerCase ( ) ;
        List < String > towns = findTownsInArea ( normalisedAreaName ) ;
        if ( contacts == null ) initialiseContacts ( ) ;
        for ( Contact c : contacts ) {
            if ( towns.contains ( c.homeTown ) )
                System.out.println ( "Found " + c.name + " with home town in " + area ) ;
        }
    }
}

```

//Our datastructures expect
//lowercase parameters

Figure 9.19: A function to search a list of contacts and subsequently print those living in a specified area—emphasis used.

9.2.4 Logical Grouping

When a programmer purposefully groups one code fragment with another, they are communicating that a logical connection between them exists. This is done to improve code readability and reduce the time spent understanding the code.

An example of logical grouping can be seen when boxes are used purely as a visual aid, to categorise code fragments that a programmer would naturally place on the same Frame. Consider the list of fields, shown in Figure 9.20, taken from the same chess program featured in Section 9.2.2. Boxes are used to visually distinguish three groups from each other.

- **Top Group.** This group contains three fields, two of which form a subcategory. The subcategory is represented using a green box within the surrounding blue box. The field *canvas* is used to execute the commands to draw the chess board. The fields *width* and *height* are used to specify the size of the window, and therefore the size of the squares on the chess board.
- **Middle Group.** This group contains two fields, each relating to the current state of the chess program. The first field in this group is a reference to a class the programmer has created to keep track of the current state of the board. The second is another class the programmer has created to alert the player of specific changes to the state of the game—such as the win condition being satisfied.

- **Bottom Group.** This group contains only a single field, a reference to an inner class *GamePiece.Creator*. When instantiated, this object acts as a factory for producing game pieces.

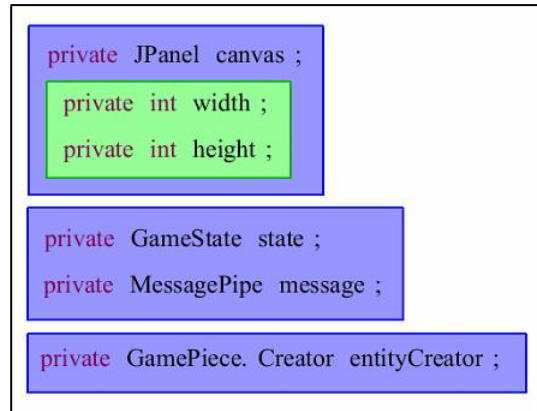


Figure 9.20: A list of categorised fields from the chess program implemented in *SplDER*.

Another example of logical grouping can be seen in Figure 7.3 on page 155. In this case, two code fragments—functions—that, should the author not be trying to make a point, would normally remain separate, are being deliberately placed side-by-side on a single Frame. Boxes are used to semantically separate them from each other so that the flow walker may function correctly. Both of these functions execute similar code on different, but related, variables. Whereas the example in Figure 9.20 uses grouping to distinguish subgroups in a list of code fragments that naturally go together, the example in Figure 7.3 uses grouping to pair two normally separate code fragments together.

9.2.5 Alternative Forms of Abstraction

Consider a scenario where the length and complexity of a function is growing. Readability begins to deteriorate. Opportunities to label and reuse fragments of code may be being missed. At this point the programmer may decide to refactor the code, making use of abstraction to break up the code, thereby regaining readability. In this scenario, the programmer is faced with a decision: what form of abstraction is most suitable for the given situation?

Abstraction in General. We consider the creation of an abstraction to be any transformation that allows for a fragment of code—regardless of size—to be seen to have greater independence than prior to the transformation being undertaken. An abstraction

can be used to improve readability or provide opportunities for reuse. We can define an abstraction in terms of three properties: a label, the code's context and blocking, the latter being a term we coin, and expand upon shortly. We make use of two running examples from traditional programming to illustrate these properties: the creation of a new function and use of blank lines for separation.

- **Label.** The presence or absence of a label (such as an identifier) determines how and if the abstraction can be referenced. For example, when extracting code into a new function, a name must be specified. This name has formal meaning in the underlying programming language and allows for code reuse.

When separating the steps of a function with blank lines, documentation may optionally be used to explain what each step does. Such comments also perform the task of labelling. They assist with code understanding, however they do not allow for code reuse.

- **Context.** When an abstraction is created, the context surrounding the fragment of code being abstracted may change. For example, when extracting code into a new function, variables in the block of code that now forms the body of the new function may no longer be in scope; this requires a set of parameters to be specified. These parameters have formal meaning in the programming language.

Using blank lines to separate steps in a function does not involve formal changes to context. However, documentation may be used to create artificial context to improve readability. For example, a programmer may wish to list the names of the variables used in the code that follows.

- **Blocking.** We define blocking to refer to the degree that the abstracted excerpt of code is removed where it is used (its origin). In general, the blocking of an abstraction is best viewed in relation to the blocking of another abstraction. That is, when comparing two abstractions, one will be more removed from its origin than the other. For the purpose of our explanation we will begin by documenting the two extremes of blocking.

- **In Situ.** This is when an abstraction does not syntactically move the code being abstracted from its origin. For example, using blank lines to separate steps in a function produces a visual distinction but does not cause any syntactic changes.
- **Removed.** This is when an abstraction produces a distinct visual separation between the abstracted code and its origin. For example, extracting a fragment of code to become a new function produces both a visual and syntactic change to the code: content from the abstraction is replaced with a call to the newly created function and the abstracted fragment of code is quite possibly no longer visible at the same time as the function it was extracted from.

Table 9.1 summarises the structure of a general abstraction. Specific abstractions will use one of the two forms of Blocking listed. The form of blocking used determines the requirements for labelling and context. For example, a specific abstraction that uses Removed Blocking must have a label and must create a new context. Throughout this section, as we document further examples of abstractions, we will specify which form of Blocking they use.

BLOCKING	LABEL	CONTEXT
In Situ	Optional. No formal meaning.	Optional. No formal meaning.
Removed	Required.	Creates new context.

Table 9.1: The two general forms of Blocking used to in abstractions and their rules for labelling and context.

Abstraction in Traditional IDEs. In a traditional (relative) programming environment the programmer's options for abstraction are more limited compared to those available in SpIDER. One of the options is to use a form of abstraction we will refer to as White Space Partitioning. This uses minimal editing to visually bring out logical boundaries between fragments of code through white space or documentation. An option that requires more editing is to use an abstraction we refer to as Extracting as Function. This is accomplished by converting a logical block of code into its own function, potentially with no scope in common. If the programming language used supports closures, then an intermediate

transformation would be to define an anonymous function, requiring less change to context than a fully extracted function. This abstraction is called a Lambda Abstraction.

Table 9.2 lists these three forms of abstraction. Each entry specifies the form of Blocking used and how it is constructed in terms of label and context. We list these in order from least removed to most removed.

- White Space Partitioning (row 1) uses In Situ Blocking. The presence or absence of a label or change in context is, as with In Situ Blocking in general, optional.
- Extracting as Function (row 3) uses Removed Blocking. Whereas the general definition of Removed Blocking tells us that a label is required and that new context is created, the entry for Extracting Code provides more detail. Not only must the label be present, but it must be unique as programming languages require functions to have unique names. The new context that is created is determined by a list of parameters and the destination of the new function.

ABSTRACTION	BLOCKING	LABEL	CONTEXT
White Space Partitioning	In Situ	Optional. No formal meaning.	Optional. No formal meaning.
Lambda Abstraction	In Situ	Typically no label but can be stored in variable for reuse.	Parameters. Scope depends on where it is declared.
Extracting as Function	Removed	Required. Must be unique.	Parameters. Scope depends on destination of new function.

Table 9.2: The structure of three specific abstractions in traditional programming—White Space Partitioning, Lambda Abstraction and Extracting as Function.

Lambda Abstractions use In Situ Blocking. A Lambda Expression is not created with a visible label. The programmer can choose to apply a label by assigning it to a variable—necessary if it is to be used multiple times. As a function, a Lambda Expression has a new context in the form of parameters. There is an important distinction between the context created by a Lambda Abstraction and Extraction as Function abstraction. Whereas the scope in a new function—created by the Extracting as Function abstraction—can be completely detached

from the scope it originated from, the scope in a Lambda Expression is tied to the location it was declared.

The inclusion of Lambda Abstractions provides a compromise between White Space Partitioning and Extracting Code. It now begins to become beneficial to start thinking of specific abstractions as sitting somewhere on a continuum. We will use Blocking as our primary axis. Towards the left of the continuum, abstractions with In Situ Blocking are placed. Towards the right of the continuum, abstractions with Removed Blocking are placed. When comparing two abstractions with the same blocking, those with fewer restrictions concerning their label and context are placed to the left of those with more. For example, both White Space Partitioning and Lambda Abstractions use In Situ Blocking, however White Space Partitioning does not require new context to be created, whereas a Lambda Abstraction does. This places White Space Partitioning further left than Lambda Abstraction on our continuum.

Abstraction in SpIDER. Just as the inclusion of Lambda Abstractions did, SpIDER's flow walker provides the programmer with additional options when creating an abstraction—more points along the continuum. The primary method by which this is achieved is by refining our definition of Removed Blocking.

Figure 9.21 presents a diagrammatic refinement of the structure of an abstraction—presented without refinement in bullet point form earlier in the section when describing abstraction in general. Two new forms of Blocking have been introduced, Out of Flow and Off Frame. Previously, a graphical representation would have shown two forms of blocking: In Situ and Removed. Refinement of the later has allowed it to be considered a category. It is within this category that the new forms of blocking we are introducing are placed—along with the existing Removed Blocking. We use the phrase Lightweight Abstraction to refer to specific abstractions that use Out of Flow or Off Frame blocking.

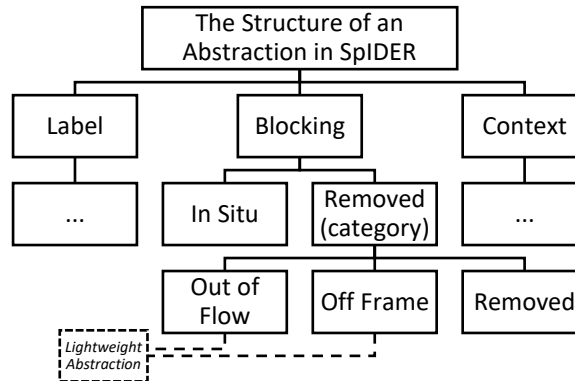


Figure 9.21: The general structure of an abstraction in SpIDER.

Examples of code being placed off frame or out of flow can be seen throughout the thesis, most notably in Sections 7.1.6 and 7.1.3 respectively. Out of Flow Blocking uses flow walker functionality to remove an abstracted fragment of code from the general flow. Off Frame Blocking uses the Frame and Linking System to position the abstracted block of code on its own Frame. Both visually displace code more than White Space Partitioning does, with Off Frame Blocking producing more separation than Out of Flow Blocking.

Table 9.3 shows the forms of blocking present in SpIDER. When defining a specific abstraction in SpIDER, we use this table over Table 9.1 to determine the resulting contracts. Labelling and Context on Out of Flow Blocking is identical to Inline Blocking, the difference is the degree to which the abstracted block of code is removed. While abstractions using Off Frame Blocking do not create syntactically meaningful context as those using Removed Blocking do; a label is required. This label comes in the form of the content associated with the link leading to the Frame with the abstracted code. The content of this label has no effect on the serialised structure of the code produced in SpIDER and can therefore be anything; in practice however, a descriptive phrase of the code behind the link would be more useful than a nonsensical phrase such as an ellipsis (“...”). Additional labels such as annotations, comments or hypermedia may also be used to support the required label.

BLOCKING	LABEL	CONTEXT
In Situ	Optional. No formal meaning.	Optional. No formal meaning.
Out of Flow	Optional. No formal meaning.	Optional. No formal meaning.
Off Frame	Required. No formal meaning.	Optional. No formal meaning.
Removed	Required.	Creates new context.

Table 9.3: The four general forms of Blocking used to in SpIDER abstractions and their rules for labelling and context.

Additional options for labelling and providing context are also available. When creating abstractions in traditional IDEs, programmers are limited to using text-only documentation for markup. SpIDER’s hypermedia support allows programmers to label and provide context with images, annotations, sound files, diagrams etcetera. For example, in the case of abstractions using Off Frame Blocking, the link leading to the Frame with the abstraction on it can be used to provide context.

Table 9.4 summarily documents the existing three specific forms of abstraction in traditional IDEs in addition to three Lightweight Abstractions made possible by SpIDER’s flow walker algorithm—as was the case in Table 9.2, entries are ordered as they would appear in the continuum we have been constructing. The three Lightweight Abstractions are: Anonymous Indirection, Flow Chart (newly introduced to demonstrate how new forms of abstraction can be given consideration) and Extracting as Frame. An example of Anonymous Indirection can be seen below. A Flow Chart abstraction is suitable for distinguishing the steps of an algorithm when each step is of comparable size, chaining (Section 7.1.4) is used to represent each step. Extracting as Frame moves abstracted content onto a new Frame, substituting a Text Item linking to the newly created Frame. This is visually similar to the results achieved by Extracting as Function. As with Extracting as Function, a label is required. However, in keeping with the requirements of its Off Frame Blocking, Extracting as Frame does not require this label to be unique.

ABSTRACTION	BLOCKING	LABEL	CONTEXT
White Space Partitioning	In Situ	Optional. No formal meaning.	Optional. No formal meaning.
Lambda Abstraction	In Situ	Typically no formal meaning. Must be stored in variable for reuse.	Creates parameters. Scope depends on where it is declared.
Anonymous Indirection	Out of Flow	Typically not, never with formal meaning. Accessed via arrow. Can be accessed multiple times per Frame.	Optional. No formal meaning.
Flow Chart	Out of Flow	Typically not, never with formal meaning. Accessed via arrow. Can be accessed multiple times per Frame from any step in the chain.	Optional. No formal meaning.
Extracting as Frame	Off Frame	Required. No formal meaning.	Optional. No Formal Meaning.
Extracting as Function	Removed	Required. Must be unique.	Creates parameters. Scope depends on destination of new function.

Table 9.4: The structure of six specific abstractions in SpIDER, three of which are Lightweight Abstractions.

Figure 9.22 shows an example of an Anonymous Indirection abstraction. The code in the figure shows a recursive *sort* function—part of a merge sort application implemented in SpIDER—that is split into three parts. Anonymous Indirection is used to visually separate the three parts from each other. The first part is the base case for the recursive function. The second part creates and manipulates arrays to further the sorting process. The third part executes the recursive call. In keeping with the specifications of an Anonymous Indirection abstraction, no formally meaningful label is provided; an arrow is used to access the abstracted code. In order to assist with readability, an informal label is provided in the form of an annotation. No new context is created—formally meaningful or otherwise.

```

private static int [ ] sort ( final int [ ] input ) {
    final int middle = input.length / 2;
    final int [ ] left = new int [ middle ] ;
    final int [ ] right = new int [ input.length - middle ] ;
    for ( int i = 0 ; i < middle ; i ++ ) left [ i ] = input [ i ] ;
    for ( int i = middle ; i < input.length ; i ++ ) right [ i - middle ] = input [ i ] ;
    return merge ( sort ( left ) , sort ( right ) ) ;
}

```

Figure 9.22: Anonymous Indirection used to express multiple steps in a function.

Consider the continuum we have been building and the specific abstractions placed along it (Table 9.4). As a thought experiment we can argue that Anonymous Indirection is the most suitable abstraction to use to delineate the steps in the *sort* function from Figure 9.22.

Firstly, Extracting as Function and Lambda Abstractions can be eliminated from consideration because the code we are looking to abstract contains return statements. Furthermore, the length of the code fragment to abstract in each case is sufficiently short that moving them onto their own Frame seems like an over-engineered solution. Similar reasoning can be used to eliminate a Flow Chart Abstraction. This leaves us with White Space Partitioning, Anonymous Indirection and any other specific abstractions we have not documented. For the purpose of our thought experiment we can eliminate those not documented. It therefore comes down to the degree of visual removal we desire. As both fragments of code we wish to extracted are flanked by either the middle step of the function or boilerplate code, it seems logical that more removal is better; therefore, an Anonymous Indirection abstraction seems like the most logical choice.

9.2.6 Alternative Handling of Containment

Consider the task of creating an action listener in Java. There are multiple approaches to producing code that satisfies this task. Figure 9.23 shows one of these approaches—creating an anonymous class instance. In producing this code, no spatial behaviour beyond that which is described in Section 9.1.1 has been utilised; providing a baseline from which to demonstrate an alternative way to express containment in code. This fragment of code is

from a calculator program implemented in SpIDER. Specifically, this code specifies a set of actions to execute when the user activates the ‘Add’ button.

```
btnAdd. setOnAction (
    new EventHandler <(ActionEvent) > ( ) {
        public void handle ( final(ActionEvent) e ) {
            System.out.println ( "How much would you like to add?" );
            final Scanner in = new Scanner ( System.in );
            final int toAdd = Integer.parseInt ( in.next ( ) );
            in.close ( );
            AdderFX.currentValue += toAdd ;
            System.out.println ( "Adding done; why not ask for the current value?" );
        }
    } );
```

Figure 9.23: Creating an anonymous action listener—without using deconstruction.

We introduce the term ‘Deconstruction’ to describe an Expressive Pattern where the programmer utilises chaining and spatial positioning to improve the readability of code. Readability is improved by extending the expression of containment from using bracket pairs and indentation to using arrows and boxes. Code is split into parts, where each part is syntactically contained within a surrounding part. Chaining is used to direct the flow of the code between parts. Arrows are positioned between opening and closing brackets. Brackets are visually positioned directly next to each other. The positioning of brackets as well as the presence of boxes and arrows visually communicates which part encloses any given part.

Figure 9.24 shows the same code as seen in Figure 9.23. In this case however, Deconstruction has been used to spatially lay out the anonymous class and its association with the ‘btnAdd’ variable. In this example, the code has been split into four component parts. The deconstructed layout helps promote the important elements of the code by minimising the visual impact of less significant parts of the code. In particular:

1. The positioning of the **instigating statement** makes it easier to identify ‘btnAdd’ as the control we are adding an event handler too. Three aspects of its positioning contribute to this: its seclusion, there is no nearby boilerplate; its logical positioning,

it is positioned in the top left corner of the Frame; and the careful positioning of brackets that allow the statement to be seen in full.

2. The **outer layers** of the anonymous class—the constructor call and ‘handle’ method signature—are each positioned in their own boxes and are positioned to the right of the screen. This separates them from the **executed actions**. These boxes each contain only a single line of code. This combined with their positioning—especially when compared to the **executed actions**—deemphasise them.
3. The final component part contains the core statements to be executed when ‘btnAdd’ is activated—the **executed actions**. A large red box, spanning the width of the screen is used to contain this code. By applying deconstruction, several sets of brackets, and the resulting indentation, has been visually withdrawn; removing the necessity of keeping track on them. This allows a programmer to more easily analyse and manipulate this part of the code; beneficial as this is the part of the code likely to require maintenance in the future.

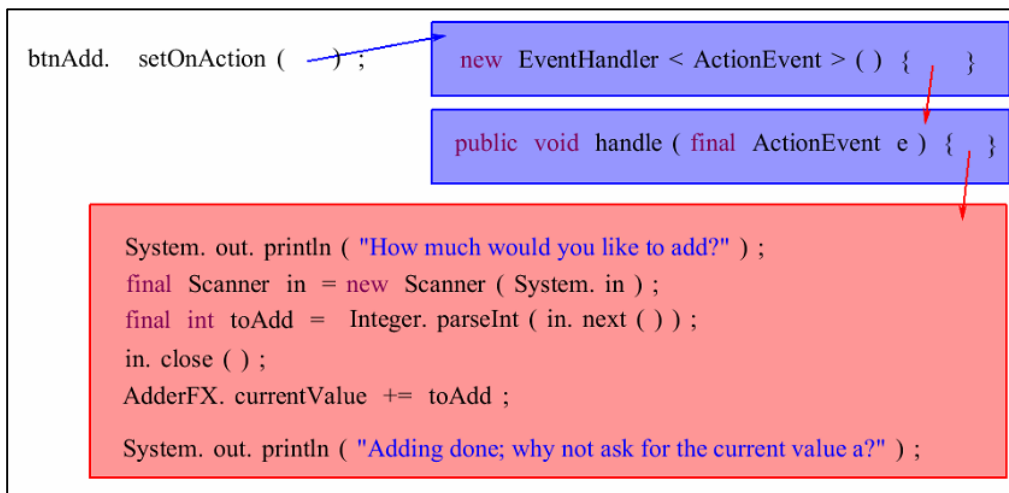


Figure 9.24: Creating an anonymous action listener—using deconstruction to improve readability.

The visual changes that Deconstruction makes to improve readability address aspects of code identified by Buse and Weimer [29] as problematic. Buse and Weimer developed an application to automatically assess the readability of code. In developing their application they conducted a study to find what aspects of code can be interpreted to decide upon a level of readability. Some of their strongest findings show that brackets, line length and deep indentation hinder readability whereas blank lines help. These are all aspects that deconstruction addresses. We conjecture that blank lines help the readability of code by

visually separating code fragments, an effect also achieved by the blocking and out of flow behaviour resulting from Deconstruction.

9.3 Process Patterns

A programmer who uses the Spatial Hypermedia functionality provided by SpIDER to represent progress or change over time, with the intent of improving the efficiency of either themselves or other programmers, is using a ‘Process Pattern’. In this section, we will present four scenarios that utilise Process Patterns. Each scenario will begin by providing some motivation. Following this, the application of the Process Pattern will be split into multiple steps and documented. Each step may be viewed in terms of an Expressive Pattern or—more generally—a Spatial Metaphor; overall, however, it is the set of steps taken as a whole that improves programming productivity.

Section 9.3.1 is a scenario concerning a single programmer. In this scenario, the programmer reduces the cognitive load involved in calling and understanding constructors. Section 9.3.2 then establishes a scenario where a programmer wishes to document their progress towards finishing a task. They know they will not be able to complete it in a single sitting and therefore wish to be able to easily resume work another day. In Section 9.3.3 a programmer designs a function to allow their program to run correctly regardless of their internet connection status. Finally, in Section 9.3.4 a group of programmers create a collaborative space to help keep on task.

9.3.1 Construction by means of Notes

Donald works for a construction company that is looking to place a tender on a deal to build a bridge between a residential and commercial area, situated on opposite sides of a river—thus alleviating a long commute. As it is a government contract, it is a potential windfall for the company. Standard to government procedures the company awarded the contract is liable for maintenance for the first 10 years—this has been done to ensure that quality materials are used for construction.

The company Donald works for knows that there will be a lot of tenders for this job, they put a call out to employees for ideas on how to make their tender competitive. Donald puts forward the idea of designing the bridge to support the types of vehicles currently using the longer commute, rather than designing to match what the bylaws of the area stipulate as

possible. The company likes his idea and instructs him to create a simulation program that will safely service the traffic likely to use the bridge whilst minimising upfront and maintenance costs. Donald is provided with traffic data covering commute traffic between the residential and commercial areas.

With some experience in programming, Donald decides to build a small utility application to help him produce this simulation. His end goal is to be able to easily alter variables so that he can try different models. For example, his application should be able to tell him whether it is a good idea to provide a designated lane for buses.

The first step he takes is to create a hierarchy of classes representing the types of vehicles that will be using the bridge. Figure 9.25 shows the abstract *Vehicle* class that Donald will extend when creating Factory classes for the different types of vehicles. The constructor to *Vehicle* takes four variables describing the shape of a vehicle followed by an array of integers listing the lanes that this vehicle will be able to use in the simulation.

```
package vehicle ;
public abstract class Vehicle {
    ◦//fields: Facets of Vehicles
    public Vehicle ( int weight , double length , double width , double height , int [ ] lanes ) {
        this. weight    =    weight ;
        this. length    =    length ;
        this. width     =    width  ;
        this. height    =    height ;
        this. lanes     =    lanes  ;
    }
}
```

Figure 9.25: The base class *Vehicle* used in construction company scenario.

Having created the abstract *Vehicle* class, Donald creates concrete classes extending *Vehicle* to represent the different forms of vehicles that will use the bridge: *Car*, *Truck*, *TruckTrailer*, *Bike*, *MotorBike*, *Trike*, *Bus* etc. Each of these classes adds new parameters to their constructor. For example, as can be seen in Figure 9.26, when constructing a car the number of doors must be specified.

```

package vehicle ;
public class Car extends Vehicle {
    private int numDoors ;
    public Car ( int weight , double length , double width , double height , int [ ] lanes , int numDoors ) {
        super ( weight , length , width , height , lanes ) ;
        this.numDoors = numDoors ;
    }
}

```

Figure 9.26: An example of a concrete type of vehicle that would use the bridge.

Donald has reached the point where he can run some simulations on some fixed data and decides that doing so will be a useful test for the code he has written so far. In writing these tests Donald is able to use SpIDER's Spatial Hypermedia functionality to mark up his code, assisting in its construction and shortening the code understanding process. Figure 9.27 captures a moment in time where Donald is part way through creating a test case. This particular test case limits the types of acceptable vehicles to buses, cars and motorcycles and specifies that the outer-most lanes of the bridge be used only by buses. Each piece of Spatial Hypermedia mark-up present contributes to visually explaining Donald's code.

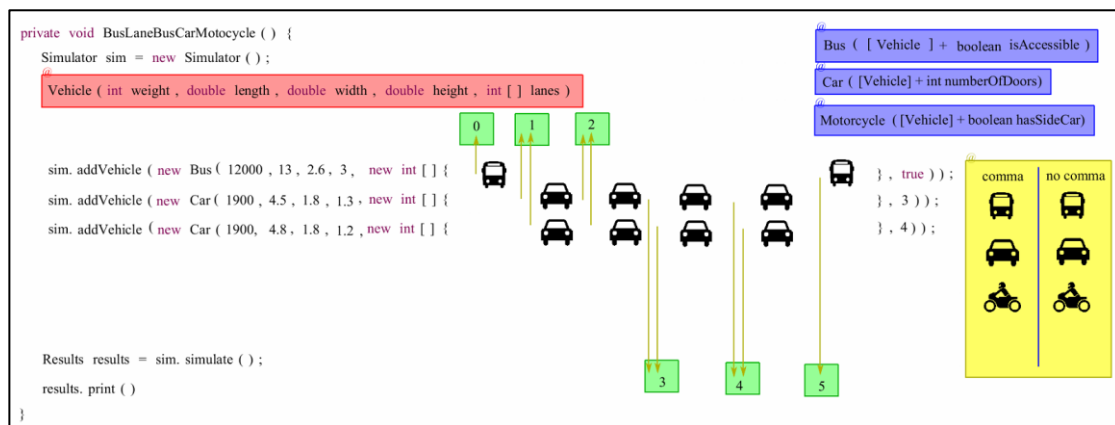


Figure 9.27: A test case for Donald's bridge simulator where Spatial Hypermedia features assist with construction and understanding of code.

The red annotated box contains a copy of the parameter list for the constructor in the abstract *Vehicle* class. Each blue annotated box documents the parameter list for the constructor of a specific type of vehicle in terms of the parameter list to *Vehicle* and any additions needed. Combined, the red and blue boxes act as an aide, reminding Donald—or any other programmer looking at the code—how each parameter is used. For example,

when analysing the second vehicle to be constructed, by using the second blue box, Donald is able to identify the last parameter, as specifying that the car being constructed has three doors.

As previously mentioned, the parameter *lanes* lists which lanes the constructed vehicle is allowed to use. Lanes are indexed from zero. In this example, the bridge is six lanes wide, three going in each direction. Therefore, in order to specify that buses only use the outer most lanes when travelling each direction, the *lanes* parameter should be declared as 'new int[] { 0, 5 }'. Donald realises that, instead of simply declaring these arrays of integers in this non-informative way, he has the opportunity to visually reinforce how traffic will use the bridge in this test. He achieves this by inserting a series of visually separated boxes and arrows. Each arrow points to a box, containing a number (0 to 5), which numbers a lane. Between sets of arrows, images are positioned. These images serve a dual purpose. They visually communicate the type of vehicle being constructed and the lanes they use. Behind the scenes, these images are linked to a Frame containing a comma; thereby causing the flow walker to produce a syntactically valid serialisation. The yellow annotated box is used as a toolbox for Donald to easily acquire an appropriate image. Donald constructs one vehicle at a time:

Producing Bus. When constructing the bus and writing the *lanes* parameter: Donald produces two arrows, one pointing to the box with '0' in it and the other pointing to the box with '5' in it. The first bus image he inserts is linked, leading to a Frame with a comma. The second bus image is not linked to a Frame. He then specifies that the bus is wheelchair accessible, closes off the declaration. SpIDER's flow walker will serialise the array declaration by following the linked Image Item, producing 'new int[] { 0, 5 }' as desired. Donald stores the two variants of the bus image in his toolbox.

Producing Three and Four Door Cars. Donald wishes to limit cars to using the inside lanes only. Therefore, the *lanes* parameter should be 'new int[] { 1, 2, 3, 4 }'. Following the same process he used to specify the lanes that buses could use, he draws a series of arrows and positions a series of images that, when serialised by the flow walker, will produce this result.

When writing the constructor for the three door car, Donald has yet to create his car images. As he did when writing the constructor for the bus, Donald creates two variants, one that links to a Frame with a comma and one that does not. Donald stores these images in his toolbox and is able to reuse them when creating the four door car.

Producing Motorcycle. Not pictured in Figure 9.27 is the step Donald takes to create a motorcycle. This requires that he create motorcycle images to be used in his diagrammatic layout of the *lanes* parameter.

Donald has followed a process to visually communicate how the lanes are set up in this test case. At this point, he has all the tools he will need to continue to populate the test case with more vehicles. While following this process he has created multiple artefacts: the constructor parameter lists in the red and blue annotated boxes, the toolbox in the yellow annotated box and the two image variants for each vehicle. When moving on, to create a new test, he will be able to take copies of some these with him, removing the need to recreate them each test. For example, should Donald wish to create a test that specifies the outside lanes are push bike only lanes instead of bus lanes, he will have to create a blue annotated box for bikes, but will be able to reuse the annotated boxes for *Vehicle*, *Car* and *Motorcycle*.

9.3.2 Documenting History

Grace has been working as the sole developer at an accounting firm with a particularly odd set of clientele for the previous 10 years. Frequently a client will make a request that cannot be accomplished with the commercial accounting software her associates use. When this occurs, it falls to Grace to create a piece of one-off software that an accountant can use to satisfy the client's request. She has recently been working on a piece of software to produce reports for a local zoo. A liaison from the zoo has stipulated that their statement of financial position should note each animal's country of origin. Supposedly the zoo hopes to use this information for tax purposes.

Grace is due to head overseas for a holiday at the end of the working day. As such, she had hoped to have completed her current project for the zoo before days' end. Unfortunately, a

particularly annoying bug, in her code to catalogue the zoo's insect exhibits, has been causing her problems for several days now. Grace has made progress with the bug. She has a reproducible instance of the bug and knows how to get the debugger to a point shortly before the error occurs. The accounting firm has hired Alan from a temporary work agency to cover for Grace while she is away. Unfortunately this means that Alan is going to have to take over from Grace at a problematic time. Unable to delay her flight, Grace decides to document her debug history—at least this way Alan will know what she has already checked and will be able to more easily pick up from where she left off.

In order to document her debug history she resolves to revisit each previous debugging step from the start of her process and make notes as she goes. Figure 9.28 shows the entry point into her program as it stands before she began debugging. The application begins by specifying a file to print its results to. It then reports on each category of animal—a non-trivial task requiring the search of a database. Each of these categories requires a different type of enclosure for storing animals, roughly represents a different area in the zoo, and contains multiple exhibits. For example, the land mammal category contains a safari exhibit with elephants and giraffes; it also contains a jungle exhibit with various types of monkeys. The application finishes by gracefully closing the connection to the output file.

```
package spiderdefault ;
◦ imports

public class AnimalsByCountryOfOrigin {                                @IOException from report.close() not handled.
    public static void main ( final String [ ] args ) throws IOException {
        BufferedWriter report = new BufferedWriter ( new FileWriter ( "report.csv" ) ) ;
        ◦ //report on land mammals
        ◦ //report on water mammals
        ◦ //report on reptiles
        ◦ //report on aquatic
        ◦ //report on insects
        report. close ( ) ;
    }
}
```

Figure 9.28: The entry point into Grace's project for the zoo, prior to any debugging occurring.

When running her application Grace receives the following output (truncated to relevant parts):

```
...
BEGIN INSECT CATALOG
BEGIN BUTTERFLY CATALOG
Species Tag: MonarchButterfly, Country of Origin: Various, Last Count: 146
Species Tag: BluesButterfly, Country of Origin: Japan, Last Count: 23
Species Tag: MetalMarksButterfly, Country of Origin: South Africa, Last Count: 28
END BUTTERFLY CATALOG
BEGIN MOTH CATALOG
Species Tag: LymantriaDisparMoth, Country of Origin: Sweden, Last Count: 0
Species Tag: IndianMealMoth, Country of Origin: India, Last Count: 0
Species Tag: LunaMoth, Country of Origin: North America, Last Count: 0
Species Tag: GardenTigerMoth, Country of Origin: Various, Last Count: 0
END MOTH CATALOG
...
```

The bug is that the report is incorrectly claiming that the zoo does not have any moths in their moth exhibit. The first step Grace took when debugging was to use SpIDER annotation marks to effectively ‘comment out’ code that had to be executed prior to the insect code. With the bug still occurring once this is done, Grace can be confident that the cause of the bug is not earlier in the program. Further, it accelerates debugging by allowing Grace to add breakpoints to pieces of shared code that all categories would normally use. The second step is to alter the program to output to standard-out rather than a file, making feedback more instant. Grace makes note of these steps in an annotated box that is left on the Frame. Figure 9.29 shows an updated version of the main method to Grace’s program; reflecting the progress she has made debugging.

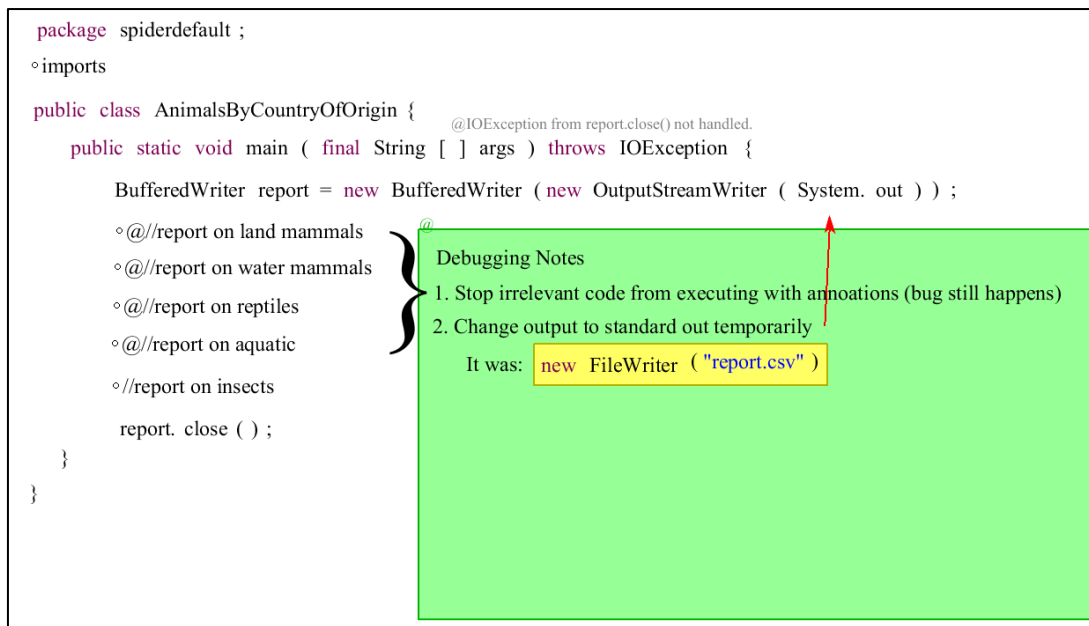


Figure 9.29: Grace's first steps in debugging sitting alongside her notes.

Following the link with content 'report on insects' reveals that the function *addToReport* is called for each exhibit. When the moths are to be processed, for example, the line of code *Moths.addToReport()* is executed. Investigating this function is the next step for Grace.

Figure 9.30 shows the result of Grace's use of the debugger to further pinpoint the location of the bug. Grace has set a breakpoint to stop the program on the first line of code in the green box. This is achieved by adding the annotation `@BP` prior to starting the debugger. Once the program has stopped at this point she requests that the debugger give her a list of variables. Grace receives the variables from the IDE as a set of first class citizens attached to her cursor and places them down, positioned in the left-bottom corner of the screen. They show that the only variable in scope at this point is the *BufferedWriter* object. She then proceeds to step over the current line, executing the code that does the database lookup and subsequent organisation into *Entry* objects. Having done so, she then re-requests the list of variables so that she may inspect the results of the call to *Database.getEntries(...)*. This second set of variables is subsequently positioned to the right of the previous request for variables. Grace expands the results from her request for variables and finds that, at this point, the values for 'expectedCount' are already zero. This further narrows down the location of the bug. Grace can now be confident that the bug is somewhere inside the *Database.getEntries(...)* function.



Figure 9.30: The results of Grace's use of the debugger to pinpoint the location of the bug.

Grace has now finished running through the steps she has taken to debug her code. She takes a copy of the results—as shown in Figure 9.30—obtained from running the debugger on the *Moths.addToReport(...)* function and places it on a non-code Frame. She then updates the notes for Alan to include the step she took running the debugger—this included adding a link to the non-code Frame she had just created. She also adds some detail to the non-code Frame explaining the lines of code the debugger was run over along with some suggestions on what to check next. An updated version of the entry point to Grace's program, with the above changes, can be seen in Figure 9.31.

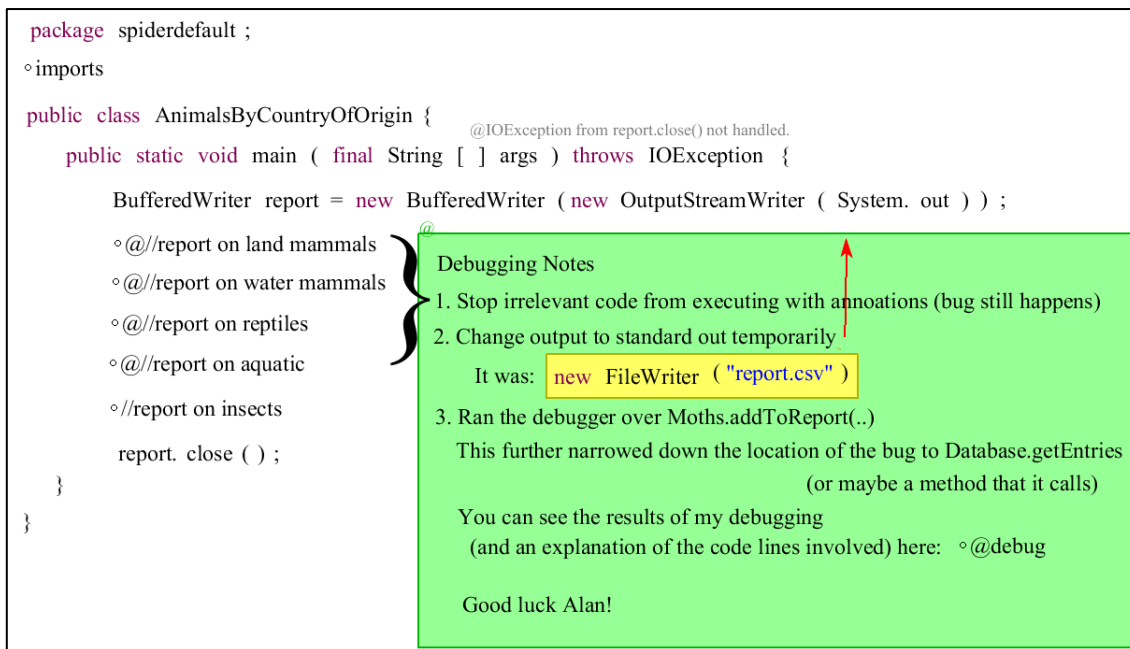


Figure 9.31: The final copy of notes that Grace leaves for Alan, including the link to the results from her run of the debugger.

9.3.3 Altering Flow

Ada works as a freelance programmer. She is currently working on a project for a taxi firm to help monitor their drivers. The company is based in a different city to where she lives. They have requested that she build a piece of software to sit between a large display screen in their depot and a live feed of information concerning their drivers. The software Ada has written interprets data from the company's live feed and displays a formatted version of it on the large display screen so that the dispatchers of the taxi company can keep track of the locations of drivers.

In order to allow Ada to work from home, the taxi company has provided her with an internet address from which to read the live data. The taxi company requires fortnightly in-person meetings to gauge progress. To allow for this, Ada is flown to the company's headquarters, arrives to present her progress and then flies back home on the same day. Ada likes to use her time in the air to double check the functionality that she will be presenting. Unfortunately, the airline that the taxi company uses to transport her does not provide an in-flight internet connection, making it impossible for Ada to use the live data feed while she is flying. In order to address this, Ada has produced text files whose content mimics the protocol used by the live feed. Ada has designed her own subclass of

BufferedReader: *TimedBufferedReader* which allows her to simulate the delay between entries inside the text files she has created.

Figure 9.32 shows the *Connection* class that Ada has built to connect to the taxi company's live data feed. In order to get a connection to the data, Ada calls the function *getConnection*. She has designed it so that she can easily switch to using the data from one of her text files while she is mid-flight. Chaining within this function determines what code is executed. Ada is able to adjust the chaining to produce three different results—one result creates a socket connection to the live data stream whereas the others provides access to one of two text files. As shown in Figure 9.32, the chaining causes a socket connection to be used.

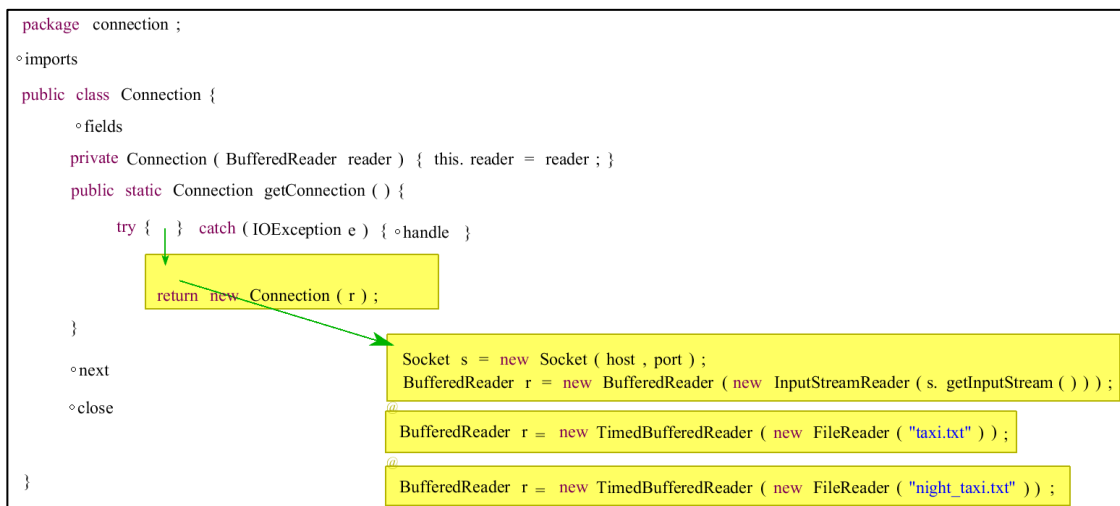


Figure 9.32: Ada's *Connection* class. Currently connecting to the taxi company's live data feed.

In Figure 9.33 Ada has switched her code to 'Flight Mode'. She has achieved this by switching the annotation from one box to another and repositioning the arrow. Both boxes contain code to produce a *BufferedReader*, thus allowing the final statement in the originating yellow box to construct a *Connection* object, regardless of which box is active.


```

package connection ;
◦imports
public class Connection {
    ◦fields
    private Connection ( BufferedReader reader ) { this.reader = reader ; }
    public static Connection getConnection ( ) {
        try { } catch ( IOException e ) { ◦handle }
        return new Connection ( r ) ;
    }
    ◦next
    ◦close
}

```

@ Socket s = new Socket (host , port) ;
 BufferedReader r = new BufferedReader (new InputStreamReader (s.getInputStream ())) ;
 @ BufferedReader r = new TimedBufferedReader (new FileReader ("taxi.txt")) ;
 @ BufferedReader r = new TimedBufferedReader (new FileReader ("night_taxi.txt")) ;

Figure 9.33: Ada's Connection class. Currently connecting to the taxi.txt file Ada has created.

9.3.4 Communication Techniques

Tim, Edsger and Charles are a small team of programmers working for a software development company currently tasked with the design and production of software to assist with crop tracking and rotation in a recently built vertical farm. As a team, they have short daily stand-up meetings. Additionally, every month they meet with Ken, a liaison from the vertical farm, to discuss their progress and get feedback. It was during their most recent daily meeting that Charles communicated his concern that some tasks had not been completed because they had been forgotten.

Standard practice at the company is for a secretary to sit in on meetings, take notes and then provide a copy of the notes to each member of the team later in the day.

Unfortunately, as Charles points out, this system that has contributed to the situation they are in at the moment—having forgotten to work on some tasks. The current system has multiple issues: as a non-programmer, the secretary has, on multiple occasions, made errors relating to technical matters in the notes, contributing to the team's reluctance to rely on them; the delay in receiving the notes, combined with the PDF format—and the hindrance this causes to editability—makes utilising them in a timely and useful manner more difficult. The disconnection between the file produced by the secretary and other artefacts relating to the code makes it more tedious to reference.

Tim, Edsger and Charles put their heads together and came up with a solution. Tim puts aside a set of non-code Frames, one of which will be used to record minutes during their

next meeting with Ken. Their solution addresses all of the shortfalls in the previous system: Tim nominates himself to be in charge of recording relevant information during the meeting. As a programmer himself, this minimises the chance of a technical error being recorded; the notes are instantly available for the team to use following the meeting; the fact that the minutes are being recorded in SpIDER provides the programmers with the full suite of Spatial Hypermedia authoring and editing functionality; the minutes are recorded alongside the other software artefacts, making referencing easier to accomplish. Following the meeting, Tim is able to export the Frame containing the minutes to a PDF, which can then be provided to the secretary for record keeping purposes. Other Frames in the reserved set of non-code Frames are used as a bulletin board to facilitate communication between the team members, allowing notes to be left, discussion to be documented and references to relevant information to be kept.

Figure 9.34 shows the Frame that Tim sets up to store the minutes from the team's meeting with Ken. Experience from previous meetings with Ken gives Tim an idea of how the meeting will proceed. He is able to use this to reserve space for three agenda topics that are likely to come up. The meeting typically starts with a demonstration of the program and its functionality. Having received feedback on their progress, the team then discuss the overall timeline for the remainder of the project with Ken. If Ken is happy with the implementation work done in the last month then this is unlikely to take much time. If Ken requires some changes to be made or wishes to adjust the project requirements, they must discuss and have Ken approve a new timeline and completion date. With the timeline for completion established, the meeting usually ends with Tim, Charles and Edsger proposing a set of tasks to be completed by next month that are then once again subject to approval from Ken.

Unfortunately, sometime between 8am and 9am, Edsger calls in sick, so was unable to attend the July meeting. This, the start time of the meeting and a list of those in attendance are listed in the red box. As this content has been placed on a non-code Frame, SpIDER does not require that boxes be marked as annotations.

July 2017

Agenda Topics

Functionality demonstration for Ken

Timeline going forward

Requested deliverables for next month

Meeting start: 03Jul2017[09:02]
Attendees: Ken (Client), Tim (Primary Notetaker), Charles
Absent: Edsger (Sickness)

Figure 9.34: Initial setup of non-code Frame for meeting with Ken.

As predicted, Ken began the meeting by requesting that new functionality be shown. Charles loaded up the application: *VertFarm.exe* and began showing Ken the changes to the end user experience since the last monthly meeting. Ken immediately noted that the load file dialog still had the incorrect file type filters listed—an issue that he had pointed out in the last meeting. Tim and Charles explained the situation and the note taking solution they had come up with to avoid this happening again. Tim made a note under the current topic to ensure this issue would be fixed as soon as possible.

Charles then brought up a newly implemented GUI panel designed to allow end users to view specifics on various crops. As it turns out, Ken had miss-communicated what he wanted: some of the values should have been editable. In the June meeting, Ken had provided the programmers with a sketch of what he imagined the panel would look like, but no-one had thought to question if any of the values where to be stored in textboxes rather than labels. While this is being discussed, Tim scans the sketch provided by Ken on to the computer and inserts it as an image into SpIDER. Text Items are then used to specify which containers are to be textboxes instead of labels.

Having sorted out the confusion, the meeting moved on to a discussion as to how this setback would affect the overall timeline of the project. Charles and Tim discuss the changes amongst themselves and decide to ask for an additional week. This will provide them with the opportunity to perform fixes and hopefully give them some slack to make up

for the fact that Edsger is sick. Ken agrees without much hesitation. The discussion then quickly moves onto deliverables for the next month. Based on the overall timeline for the project, Ken puts forward the idea that it may be time to begin working on the reminder sub-system for their vertical farm application. This system should be able to communicate with physical devices such as sirens to alert workers that it is time to begin a crop rotation. Charles and Tim agree and the meeting is concluded. Figure 9.35 shows the updated meeting notes Frame at the conclusion of the meeting as well as the sketch provided by Tim with textual mark-up applied.

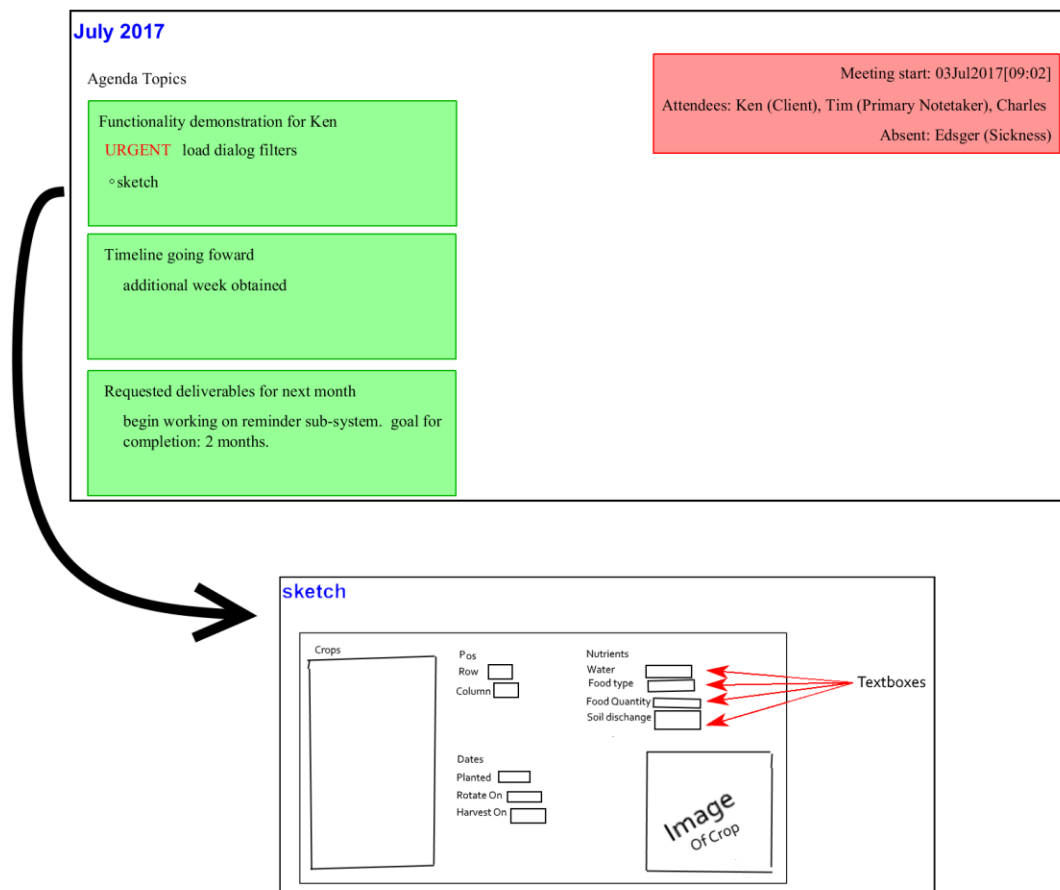


Figure 9.35: State of meeting minutes at the end of meeting with Ken.

The following morning Edsger is unable to get out of bed. Using his laptop, he obtains the latest version of Frames for the *VertFarm* project and discovers the notes from the meeting. Coincidentally, as he was stuck in bed all day, he had spent a significant amount of time thinking about how he would approach the reminder sub-system when it came around to working on it. Almost a year ago, when he was working with a different team, he had

worked on a project with a similar requirement. He navigated to the FrameSet containing this project and examined the code. Satisfied that he had found a way to be helpful, he amended the notes page with a link to the relevant part of this project and included a note to Tim and Charles.

The note left by Edsger, along with changes made by Tim and Charles following the previous days meeting with Ken, can be seen in Figure 9.36. Tim and Charles have added linked Text Items referencing code Frames belonging to the *VertFarm* project. For example, the linked Text Item with content “GUI Code” leads to a Frame containing the code responsible for creating the GUI that Ken has identified as requiring adjustments. Colour coding has been applied to keep track of the current state of the notes. A link with a red background indicates that it is new and still needs to be followed up by one of Tim, Charles or Edsger; a yellow background indicates that the team believes the task to be completed but has yet to be approved by Ken.

July 2017

Agenda Topics

Functionality demonstration for Ken

URGENT load dialog filters

◦Completed @ 03Jul2017[11:17]

◦sketch => ◦GUI code

Timeline going foward

additional week obtained

Requested deliverables for next month

begin working on reminder sub-system. goal for completion: 2 months.

◦Class definition

Meeting start: 03Jul2017[09:02]

Attendees: Ken (Client), Tim (Primary Notetaker), Charles

Absent: Edsger (Sickness)

Hey guys, sorry I missed the meeting yesterday. I am hoping I will be better before the week is over. I realised yesterday that I had previously worked on a project that had to interact with a physical alarm. I think it might be useful. Check it out: ◦SafetyDoor12

Cheers, Edsger 04Jul2017[11:07]

Figure 9.36: Meeting minutes with note left by Edsger the following day.

9.4 Summary: Applying Spatial Development Patterns

Chapter 6 examined Expeditee—the platform SpIDER was built on. Chapter 7 discussed the specifics of SpIDER’s design and how that design was implemented. Chapter 8 then evaluated the flow walker algorithm that SpIDER implemented to allow programmers to spatially position code. Together these chapters represent the process undertaken to develop SpIDER. Chapter 9 has taken the next step by showing how all these previously

discussed aspects of SpIDER fit together, providing the opportunity to improve code quality. Using examples, we show the reader that the Spatial Hypermedia functionality present in SpIDER, combined with the flow walker algorithm, provides programmers with a development environment capable of an extremely wide variety of expression.

Section 9.1 established that existing forms of expression from traditional IDEs are either present or could be approximated in SpIDER. This was achieved by dividing the section into parts, each dealing with a separate building block provides authors with an avenue for interaction and, through an example, showing how SpIDER approaches the concept embodied in the building block.

Section 9.2 examined Expressive Patterns. A programmer that uses space or Hypermedia functionality to visually communicate structure or relationships amongst code and other program artefacts is using an Expressive Pattern. Examples covered a range of development activities, from the integration of documentation to the exaggeration of specific facets of code. Each of these examples used visual communication such as colour or nesting to improve the quality of the code. Some provided functionality to the application, such as the integration of documentation as seen in Section 9.2.1.

Section 9.3 examined Process Patterns. A programmer using space or Hypermedia functionality to visually communicate changes over time is using a Process Pattern. As with Expressive Patterns, the examples used to showcase Process Patterns cover a range of activities from the development process. Section 9.3.1 showed how Spatial Hypermedia functionality can be interwoven with code, producing a Frame that was both functional and explanatory. Section 9.3.2 used the non-transient nature of content in SpIDER to assist with documenting the history of a debug session. Section 9.3.3 showed how arrows and boxes could be used to alter the flow of code to suit multiple scenarios. Finally, Section 9.3.4 showed that traditionally separate parts of the software development cycle can be brought into SpIDER, further integrating the entire process.

In this chapter, we have only touched on the strength of SpIDER's spatiality and expressiveness. The wide freedom of code expression provided by the Spatial Hypermedia environment and the flow walker algorithm make it impossible to produce a compressive list of ideas for expressing code and code processes. This is the reasoning that lead us to approach this chapter as a series of examples. It is our hope that, as someone reads

through this chapter, that they are able to use these examples as a catalyst for forming their own ideas of how spatial layout can be used to improve the quality of code. A cautious reader however, may be concerned that the freedom of SpIDER's spatiality may be used in less than ideal ways. These concerns are discussed in the next chapter.

Chapter 10

Recommendations for the use of Spatial Layout

In Chapter 9 we presented a series of examples aimed at conveying the wide range of expression that SpIDER provides for programming. We ended Chapter 9 on a cautionary note, stating that the freedom provided by SpIDER’s spatiality can be abused and mistakenly used in such a way as to hinder readability. Expanding on this thought: in much the same way a programmer may layout code poorly in a traditional IDE, so too can a programmer make comparable mistakes in SpIDER. However, in this case, the Spatial Hypermedia functionality combined with the flexibility of the flow walker, amplifies this issue by providing the programmer with many more opportunities for expression—some of which can be perverse! We now address these concerns.

Three avenues for less than ideal spatiality are presented. For each, we explain the issue and document a set of best practices to minimise the issue. The best practices stated have been formed by the author as a result of working with the SpIDER prototype during its development, and lessons learned from the usability experiment detailed in Chapter 8. Section 10.1 examines the forgiving approach of the flow walker and the possibility that this may lead to ambiguous layout. Section 10.2 then addresses the issue concerning the positioning of arrows and the effect this can have on people’s perception of the flow walker. Section 10.3 identifies a potential issue concerning the abstraction of content onto a linked Frame.

10.1 Ambiguous Code Layout

Consider Figure 10.1 and ask yourself: will the flow walker assemble code in which *x* or *y* is to be printed before the other? The answer depends on whether the bounding boxes of the two central print statements overlap on the *y*-axis. If they do overlap then a line will be constructed (as explained in Section 7.4.1), causing the print statement that outputs *x* to precede the other. Alternatively, if they do not overlap on the *y*-axis, then the print statement that outputs *y* will be seen as occurring on the line above the print statement printing *x*, causing the print statement that outputs *y* to occur first. As laid out by the programmer in Figure 10.1, it is visually ambiguous as to if the statements overlap on the *y*-axis. For the record, the statements do overlap, causing the ‘*x*’ to print before the ‘*y*’.

```
package spiderdefault ;
public class AmbiguousLayout {
    public static void main ( final String [ ] args ) {
        System. out. println ( 1 ) ;
        System. out. println ( 2 ) ;
        System. out. println ( "x" ) ;
        System. out. println ( "y" ) ;
        System. out. println ( 3 ) ;
        System. out. println ( 4 ) ;
    }
}
```

*Figure 10.1: An example of visually ambiguous code layout. Will the flow walker cause *x* or *y* to be printed first?*

Best Practices. Since the possibility of ambiguity in the flow walker was first considered, a conscious effort, in our own programming with SpIDER, has been made to avoid ambiguous layout. We have found that good practice is to carefully position and use space generously to separate individual statements. A good ‘rule of thumb’ is: if a statement is being positioned a distance along the *x*-axis from surrounding statements, then some distance on the *y*-axis should also be given to separate it from those statements. Explicit boxing can be used to further reduce the possibility of ambiguity; this has the additional advantage of making it easier to reposition a set of tokens or statements.

Figures 10.2 and 10.3 show variations of Figure 10.1 with the ambiguity removed. Each solution is suitable for a different scenario. Figure 10.2 causes *y* to print before *x* by spatially repositioning the print statements and containing them in an explicit box. Figure 10.3 causes *x* to print before *y* by repositioning the print statements so that it is clear that a line is being formed.

As an aside, it was not a specific code example produced in SpIDER that caused us to consider the possibility of ambiguity. Instead, it was the result of an exploratory question posed during a discussion of the flow walker. Upon this question being posed, previously produced code examples in SpIDER were inspected and found to be lacking any code that might be considered ambiguous. This leads us to believe that programmers are likely to instinctively avoid ambiguous layouts.

```
package spiderdefault ;
public class AmbiguousLayout {
    public static void main ( final String [ ] args ) {
        System. out. println ( 1 ) ;
        System. out. println ( 2 ) ;
        System. out. println ( "y" ) ;
        System. out. println ( "x" ) ;
        System. out. println ( 3 ) ;
        System. out. println ( 4 ) ;
    }
}
```

Figure 10.2: Ambiguity removed: y prints before x.

```
package spiderdefault ;
public class AmbiguousLayout {
    public static void main ( final String [ ] args ) {
        System. out. println ( 1 ) ;
        System. out. println ( 2 ) ;
        System. out. println ( "x" ) ; System. out. println ( "y" ) ;
        System. out. println ( 3 ) ;
        System. out. println ( 4 ) ;
    }
}
```

Figure 10.3: Ambiguity removed: x prints before y.

10.2 Positioning of Arrows

The initial evaluation of the flow walker algorithm, documented in Section 8.1, lead us to believe that careful positioning of arrows increases programmer-flow walker agreement, with haphazard arrow positioning having the opposite effect. The follow up study, documented in Section 8.2, where this conjecture was tested, indicated that this was the case.

An example of less than ideal arrow positioning can be seen in Figure 10.4. The flow walker will produce code causing “A D E F B C” to print. This relies on a programmer both understanding and noticing that the origin of the blue arrow causes the green box to be

included between the *System.out.println("A")* and *System.out.println("B")* statements. Furthermore, the programmer must also understand that the position of the arrowhead does not affect the ordering of the lines in the green box. SPIDER allows a programmer to produce code like that seen in Figure 10.4.

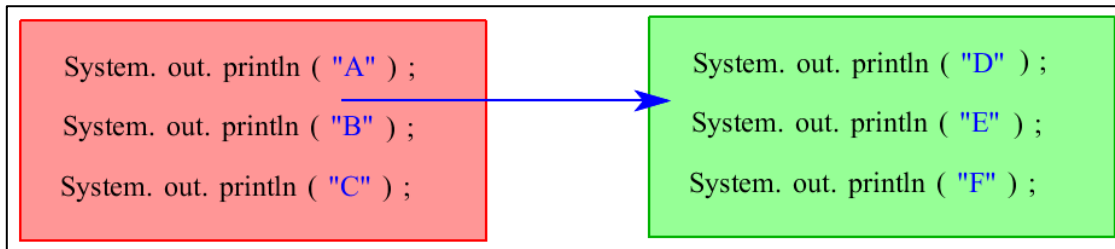


Figure 10.4: An example of out of flow Java code with bad arrow positioning.

Mitigating Factors. At the end of Section 8.3 an idea for mitigating this issue was mentioned. This idea was to implement a snap to spot system for arrow positioning. Such a system would function in a similar way to a snap-to-grid system with the difference of providing a set of pre-approved positions for arrow placement. This would force programmers to position their arrows in such a way as to minimise confusion. However, this solution has downsides. It may harm the programmer's production of Spatial Memory and limit the flexibility provided by the flow walker. For example, a programmer wishing to ensure the stem of an arrow does not intersect with a token for may wish to position the arrow head outside of the pre-approved snap-spots. A refinement on this idea would be to only cause arrows to snap when the programmer has depressed a modifier key on their keyboard. Such a refinement may lessen the negative impact on Spatial Memory whilst retaining the flexibility provided by the flow walker by making snapping optional.

Best Practises. We have developed a set of general guidelines that we consider to be best practise. These are generally applicable, minor variations for communicative or preferential reasons are acceptable. In Section 11.1.5 we apply these guidelines to produce a prototype for the previously mentioned snap to spot system.

1. The head of an arrow entering a box from the left/right should be positioned slightly inside the top-left/top-right corner of the box as is reasonable. If the corner is not suitable, the position of the arrowhead may be moved along the y-axis.

2. The head of an arrow entering a box from the top/bottom should be positioned slightly inside the top-left/bottom-left border of the box. If the corner is not suitable, the position of the arrowhead may be moved along the x-axis.
3. The origin of the arrow should be positioned centrally on an imagined text line. The programmer may want to space out the lines in the originating box to make space for this line. Care should be taken when a programmer wishes to place the arrow between one set of tokens and another (for example between two lines), the surrounding tokens should be aligned with each other.

Figure 10.5 shows an improved version of Figure 10.4. Following the first guideline, the arrow head has been positioned in the top-left corner of the destination box. Following the third guideline, the origin of the arrow is positioned centrally between the surrounding statements, which have had an increase in spacing between them.

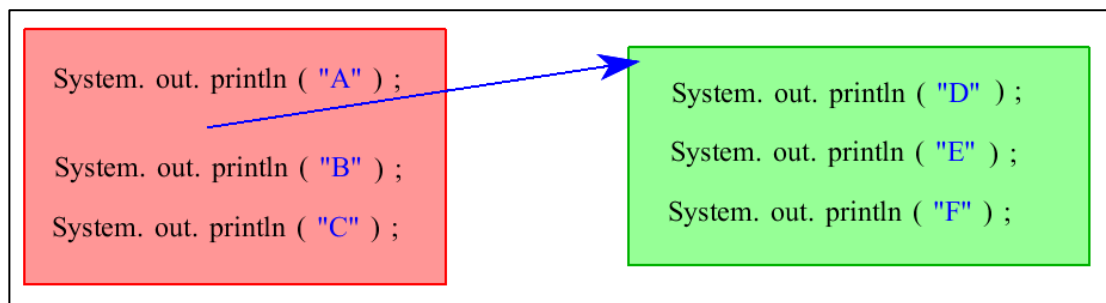


Figure 10.5: An example of Java code with improved arrow positioning.

10.3 Abstraction by way of Frame

A comparison can be drawn between SpIDER's Frame and Linking system and the characteristics of the harmful 'goto' programming statement. Whereas a 'goto' statement is used to perform an uncontrolled jump in the logic of the program, a linked Text Item is used to perform a jump and return in the flow (and subsequent serialisation) of code. However, both have the potential to—when abused—obfuscate the structure of the program.

It should be noted that the jump and return behaviour of the Frame and Linking system suggests that the analogy between it and the 'goto' statement might be better expressed as an analogy between it and the 'gosub' statement. As the serialisation process returns to the Frame it branched from after having processed Frames reached by following a link, programmers are unable to cause the serialisation to loop back on itself—a frequent pattern

of use for ‘goto’ statements prior to structural programming. However, it is the shared property of allowing for obfuscation of structure that we base our analogy on.

Through the lens of history we are able to view the controversy surrounding ‘goto’ statements and rise of structured programming as beginning with Dijkstra’s famous letter to the editor [74]. In this letter, Dijkstra argues that the use of ‘goto’ statements makes it difficult to prove software correctness. The ensuing debate saw a call for civilised discussion on the subject, resulting in numerous writers attempting to establish the strength, weaknesses and appropriate use-cases for ‘goto’ statements [75, 76]. Amidst this discussion, the discipline of structured programming was being developed [77, 78, 79]. Focusing on clearly presenting the structure of code, structured programming produced more readable, and hopefully easier to prove, code.

In this section we seek to document our thoughts on how to best counter the potential misuse of SpIDER’s Frame and Linking system. We examine two examples that demonstrate how the existing system, if not used carefully, can obscure structure and lead to reduced code readability. We then establish a set of best practises designed to avoid such cases.

Separated Structure. Figures 10.6 and 10.7 show the two Frames containing a function called *getConnection*. Instead of arranging the content with a hierarchical setup, the programmer has opted to paginate the content. This has caused the structure of the function to be distributed over multiple pages. There are three levels of structure—represented by indentation—in the function: the declaration, the error handling and the inner content. Whilst the latter is completely represented on a single Frame, it is contained within the other levels of structure, each of which is split between two Frames. The result is that, in order to ensure that a programmer understands the structure of the function, they must consider both Frames at once.

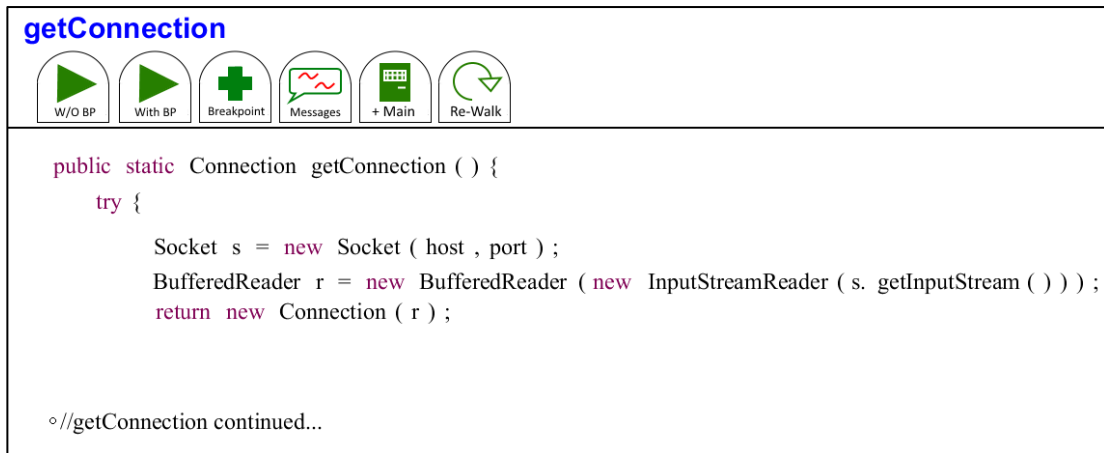


Figure 10.6: Page 1 of the `getConnection` function.

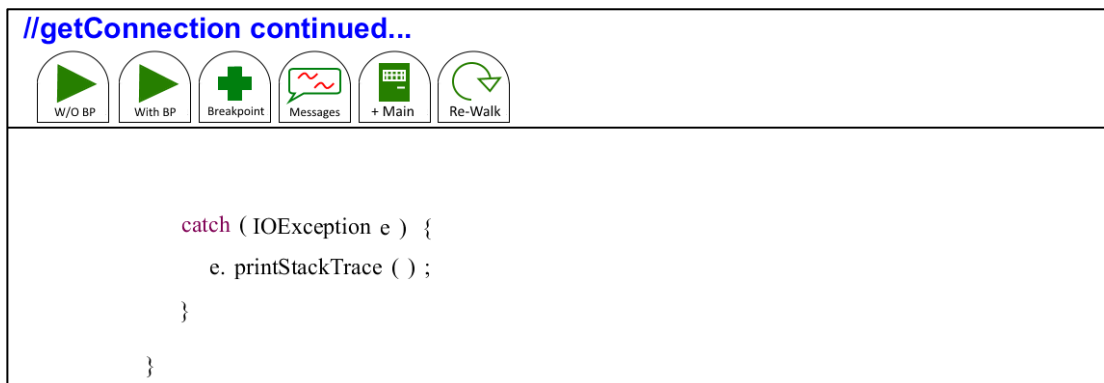


Figure 10.7: Page 2 of the `getConnection` function.

Multi-parent Reliance. Figure 10.8 is a diagram representing a portion of the Frame hierarchy for a project named Tetris. Each box represents a Frame, with the content within each box being the name of that Frame. The Frame *tetris18* contains links to *tetris19* and *tetris22*; *tetris19* contains links to *tetris20* and *tetris21*; *tetris22* contains a single link to *tetris23*. For the purpose of serialisation, each Frame has only a single parent.

Now consider Figure 10.9. A new Frame, *tetris34*, has been added to the representation with a link connecting it to the Frame *tetris19*. When the Frame and Linking system serialises the code, the content of *tetris19* will now contribute to code authored on both *tetris18* and *tetris34*. Furthermore, the code on *tetris20* and *tetris21* now indirectly contribute to the code on both of these Frames as well.

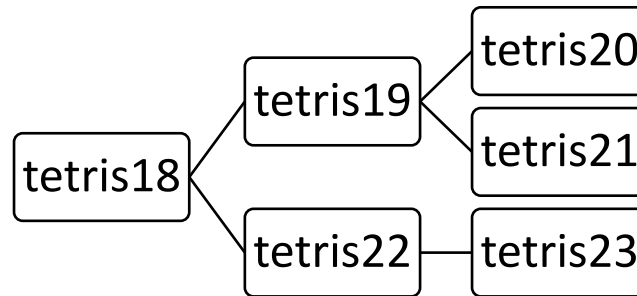


Figure 10.8: Frame hierarchy for Tetris application.

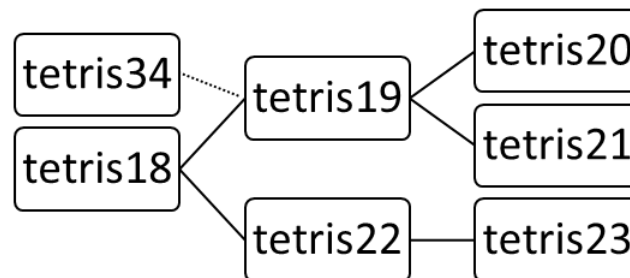


Figure 10.9: Alteration to Frame hierarchy hindering maintainability.

The addition of the Frame *tetris34* has complicated the dependencies of all Frames beneath it. When maintaining this program, a programmer must now keep all these dependencies in mind, thus increasing the chances of a fault being introduced into the program.

Best Practises. The issue of separated structure is best addressed by avoiding it. When abstracting code onto another Frame, we suggest that the division is made along structural lines. Figures 10.10 and 10.11 show the redesign of the *getConnection* function, previously seen in Figures 10.6 and 10.7. We are unable to imagine a circumstance where a structure would be better split over two Frames, rather than hierarchical abstraction being used to keep the entire structure on a single Frame.

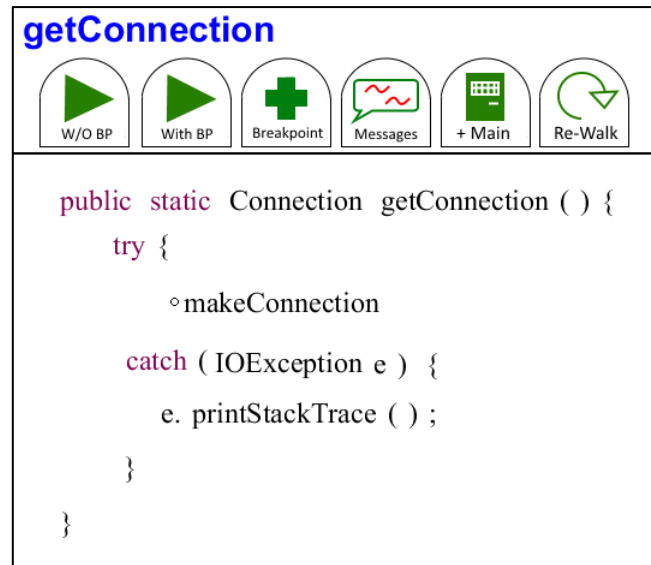


Figure 10.10: The `getConnection` function with the declaration and error handling (first two levels of structure in function) on a single Frame.

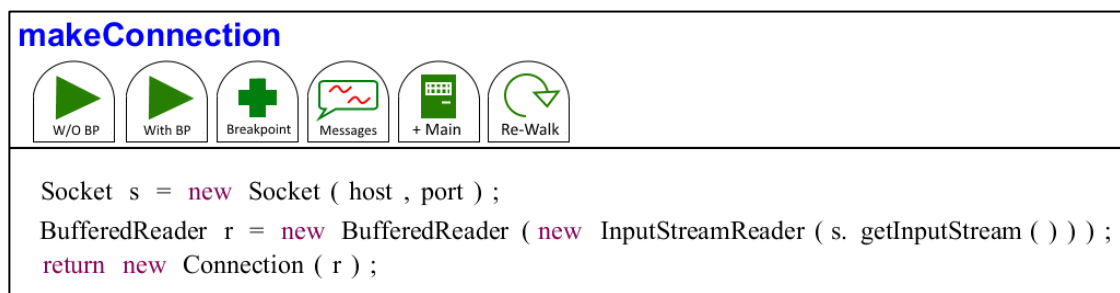
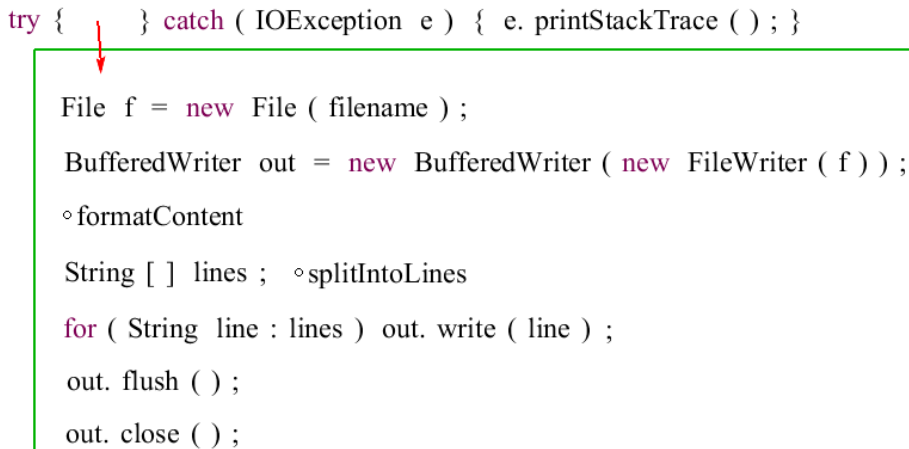


Figure 10.11: The inner content of the `getConnection` function.

The issue of multi-parent reliance is more nuanced. The ability to reference one Frame from multiple places is potentially very useful. Consider Figure 10.12 which shows the code present on the Frame *tetris19*, previously featured in Figure 10.9 as the Frame referenced by two parent Frames. Two linked Text Items, with content *formatContent* and *splitIntoLines* lead to frames *tetris20* and *tetris21* respectively.

No function declaration is present, the content of the page simply contains a code snippet that will function correctly as long as its parent Frame provides the necessary variables—in this case: a string variable named *filename* and any variables used on frames *tetris20* and *tetris21*. If this code was, for example, used to assist in debugging the application being written, then the programmer is able to create a temporary linked Text Item that leads to *tetris19* whenever they wish to send some information to the log file. No function declaration necessary.



```
try {  
    File f = new File ( filename ) ;  
    BufferedWriter out = new BufferedWriter ( new FileWriter ( f ) ) ;  
    ◦formatContent  
    String [ ] lines ; ◦splitIntoLines  
    for ( String line : lines ) out. write ( line ) ;  
    out. flush ( ) ;  
    out. close ( ) ;  
} catch ( IOException e ) { e. printStackTrace ( ) ; }
```

Figure 10.12: The content on Frame tetris19

This scenario is an example of Extracting as Frame, previously discussed in Section 9.2.5. As previously documented, there are trade-offs to consider when using this form of abstraction—while simpler to create, there is no formally defined label or context. A programmer should carefully consider their application before using multi-parent reliance:

- Is it necessary to provide context? If yes, you are not benefiting from the convenience of not having to provide context that an Extracting as Frame abstraction provides. In our example from Figure 10.12, the programmer is having to ensure that specific variables are declared on a parent Frame; minimal effort would be required to transition to providing a list of variables for a formally declared function.
- How transient are references to the multi-parent reliance structure? The longer a linked Text Item that points to a Frame exists, the more likely it becomes that it should be a more concrete and easier to maintain aspect of the program. In other words, if a multi-parent reliance structure is treated as a permanent fixture, it is likely worth creating a function.
- How complicated is the content? The content on a Frame being referenced by multiple parent Frames should be as simple as possible so as to minimise the chance of introducing an error into the application. In our example from Figure 10.12, the Frame being referenced by multiple parents contained linked Text Items itself—a red flag.

10.4 Summary: Recommendations for the use of Spatial Layout

Throughout the development of SpIDER we have been able to identify several problems relating to user-interface experience. For some of these problems, we have been able to refine the system to better handle the issue, for example: implicit boxing causes the flow walker to produce results that are closer to a user's expectations. Others, we have discussed in this section; attempting to resolve or minimise the issue.

We have identified three categories of problems that we are unable to resolve by refining the system—primarily due to our goals of maximising the use of Spatial Memory and providing flexibility in the environment. In place of a software solution, we analysed each problem and documented mitigating factors along with a set of best practises.

Personal experience gained from using the SpIDER prototype has lead us to believe that following these best practises will minimise the likelihood of these problems occurring. At the same time, we realise that each of these problems is a further avenue of research. For example, if we were able to improve the flow walker to identify ambiguous code layout, then we could notify the programmer of the potential issue and have them address it.

Chapter 11

Conclusion

Where does it say that the environments in which we edit and develop programs must be regimented and formally controlled? There is little evidence in the literature to show this has been seriously questioned. Yet the consequences of this rigidity pervade contemporary IDE design particularly in the interface components used and options provided for expression of content. This thesis has challenged the status-quo by exploring what happens when flexibility is designated as a core design requirement for an IDE. This has been achieved by using the Spatial Hypermedia application Expeditee as the base for a Spatial Hypermedia based IDE we have developed: SpIDER.

Featuring uniform treatment of elements (First Class Citizens), multimedia support, a Frame and Linking system and an algorithm that allows code to be spatially laid out on a canvas with absolute positioning, the development and evaluation of SpIDER as an IDE for the Java language and its novel functionality has been presented in this thesis. SpIDER includes the following IDE functionality: the creation of Java projects, packages and classes; syntax highlighting; a warnings and error system; compilation and debugging (including inspection of variables); context assist.

Central to SpIDER's design is the development and integration of the flow walker algorithm and magnet system which has produced an authoring environment where programmers can express functionality in a unique, interesting and most importantly, useful way. Throughout the thesis, most notably Chapter 8, examples have shown how programmers can use spatial behaviour to give emphasis to aspects of code in ways not possible in conventional IDEs. Furthermore, the approach followed allows for the integration of traditionally separate functionality.

11.1 Summary and Discussion

In this thesis we have identified, scrutinized and discussed weaknesses of conventional IDEs.

The identified weaknesses are:

- Limited ways for representing the structure and relationship of source code artefacts, thus constraining a programmer's ability to express their intent as clearly as they might like.
- Inflexibility in the support provided by IDE functionality, limiting the actions of programmers to a pre-programmed set.
- Limited multimedia integration, disrupting the development process, resulting in discontinuity for programmers by forcing them to use external tools.
- Lack of expressiveness in authored code, caused by adherence to a relative authoring environment and resulting limitations on the programmer when they wish to communicate the intent or functionality of authored code.

11.1.1 Literature Summary

Chapters 2–4 discussed literature relating to Spatial Memory, Traditional Programming and Spatial Hypermedia respectively. The topics were discussed with the goal of applying the lessons gleaned from the literature to inform the design of SpIDER. Chapter 2 covered several aspects of Spatial Memory. In particular it notes that people have both long and short term Spatial Memory which they use for navigation and object location. Software interfaces can make use of Spatial Memory by providing overviews and landmarks, fixing the size of the window and ensuring content is spatially stable.

After exploring some ways in which programmers currently utilise their Spatial Memory to assist in programming—enabled by aspects of traditional IDE design—we then expanded on the literature by discussing specific considerations that are important for methodically designing an IDE that purposefully encourages the use of Spatial Memory. The concepts of single view and viewport spatial interfaces from the literature were applied to authoring applications, leading to the terms Fixed Sized and Variable Sized Spatial Interfaces being introduced. SpIDER is a Fixed Sized Spatial Interface.

Chapter 3 examined traditional programming, using the insights gained to identify issues with the current state of IDEs. Section 3.1 discussed traditional IDEs, initially through an

historical lens. Early hints as to what the traditional IDEs would become could be seen when academic factors, such as the desire to simplify the compile-link-go process for students, led to the creation of tools that began to integrate parts of the software development process together. The 1970's saw the first commercially successful IDE: The Maestro I. It featured a screen and keyboard and in many ways resembled a modern personal computer.

Having established an historical view of IDEs, we then listed and discussed the core functionalities of an IDE: the creation and management of projects, packages and classes; syntax highlighting; a warnings and error system; compilation; breakpoints; stepping; examination of variables; and context assist.

Abstractions used by programmers throughout the software development processes, both in planning and authoring code were discussed in Section 3.2. We were able to differentiate abstractions as either rigorously defined or informal. Rigorously defined abstractions followed a protocol, allowing them to be exploited by applications but presented to the programmer as inflexible. Conversely, informal abstractions are flexible but rely on the judgement of the programmer to use them in a practical way.

An informal abstraction commonly used in programming, the use of whitespace, is also discussed. Evaluation carried out as part of this research, on the topic of whitespace use in code, shows the prevalence of blank lines in the middle of functions. Inspection of examples support the premise that they are predominately used to differentiate one code fragment from another in a function—we conjecture that this demonstrates that programmers find it useful to visually express information about the functionality of their code. The use of blank lines, an informal abstraction, results from the use of a relative authoring environment, not as a result of support provided by an IDE, and is therefore an informal abstraction of a relative authoring environment, not of the IDE.

Having noticed the lack of informal abstractions in traditional IDEs, Section 3.3 discussed the rigidity in IDE functionality this causes. Increasing the opportunity for programmers to make use of informal abstractions is a secondary goal of the thesis. The primary being, to allow programmers to spatially lay out code.

Chapter 4 covered a variety of information concerning Spatial Hypermedia. The chapter started by defining Spatial Hypermedia—we adopt a definition drawn from the literature.

Following this we defined the terms: Fundamental Element, System Representation and First Class Citizen. A Fundamental Element of an authoring application is the primary building block that is used to create content. An authoring application's System Representation describes how the elements in the application can be manipulated. An element of an authoring application that has a similar amount of flexibility to the Fundamental Element is referred to as a First Class Citizen. Using these terms, we analysed a series of authoring systems. To begin with we discussed traditional authoring systems. We explained how traditional text editors and pixel image editors functioned. We then examined more complicated authoring systems such as multimedia-editors and HTML authoring. Continuing the progression, we were then able to expand on our earlier definition of Spatial Hypermedia by constructing a System Representation of a theoretical Spatial Hypermedia application with minimal functionality.

Following our analysis of traditional authoring applications, and a theoretical Spatial Hypermedia application, we then analysed several different specific Spatial Hypermedia applications in Chapters 4 and 5. Three general purpose Spatial Hypermedia applications were reviewed: VIKI, VKB and Expeditee. Four Spatial Hypermedia systems crafted specifically for software development were also reviewed: Code Thumbnails, Code Canvas, Code Bubbles and Debugger Canvas. By comparing these systems, taking note of details we had determined as pertinent such as: whether the system allowed nesting of elements, whether the interface was a Variable or Fixed Spatial Interface, the prominence of First Class Citizens and support for multimedia elements, we were able 'tease out' the strengths and weaknesses of each.

Section 5.2 then brought together several topics of interest that had been previously mentioned throughout the thesis. From this discussion, Expeditee was identified as our Spatial Hypermedia application of choice to use as a base for developing SpIDER. As we had identified limited flexibility as a key issue in traditional IDEs, we wished to build an IDE with a flexible Fundamental Element that also heavily utilises First Class Citizens—of all the systems reviewed, Expeditee provided us with the best starting point.

11.1.2 Future Work: Chunking

Drawing again upon a field of cognitive psychology—as we have done with Spatial Memory—an intriguing area of research for future work is investigating the concept of

Chunking and its connection to programming. In particular, how it relates to source code arranged on a set of fixed sized canvases, each reached by following a links.

The term chunking was coined in the now famous paper “The Magical Number Seven, Plus or Minus Two” by Miller [80]. To summarise, Miller demonstrated that a several studies showed that humans can recall between five and nine units of information depending on the complexity of the units and the category they belong to. For example, if a subject was asked to remember and reproduce the ordering of a series of red and blue blocks, they would have more success than if there was red, blue, yellow, orange and black blocks.

Miller went onto discuss the positive measurable effect on recall that could be achieved by grouping information into more complex chunks.

The measurable effect that the amount of information able to be recalled is increased by forming groups. Extending our previous example, if a subject was given the given the task of remembering the order of a set of red and blue blocks, but was able to pick out certain meaningful patterns such as RED-BLUE-RED, and treat each instance of that pattern as a chunk, they would be able to recall a longer series before encountering problems. The concept of chunking has since been the topic of many papers that have demonstrated that it can help with both short and long-term memory [81, 82, 83].

It seems logical that, when a programmer creates a link to a new Frame in SpIDER, they are grouping the information on that Frame into a chunk. Consider the class declaration seen in Figure 11.1. Each member of the class, the fields, constructor and functions, have been abstracted onto their own Frame, represented only by the in situ link that remains.

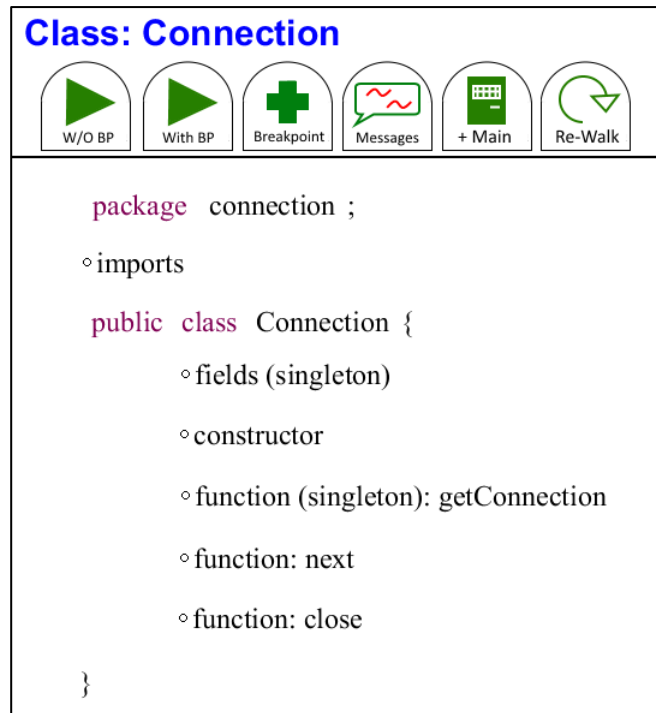


Figure 11.1: Class declaration with chunked content hidden behind links

This example demonstrates the possible connection between SpIDER's Frame and Linking system and the concept of chunking, with each linked Text Item representing a chunk. Investigation, in the form of further literature review and user evaluation of SpIDER, is required to establish whether the phenomenon of Chunking occurs in programming and if SpIDER's Frame and Linking system helps.

11.1.3 Design, Usage and Development of SpIDER

Chapter 6 bridged the discussion of literature with the discussion of the design and development of SpIDER by performing an analysis of the system Expeditee. This analysis extended the previous review of Expeditee from 4.3.3 whilst introducing aspects of Expeditee and the terms used to describe them later in the thesis.

The usage and implementation of SpIDER itself was presented in Chapter 7, along with the rationale behind the major design decisions made. Section 7.1 discussed how programmers are able to use spatial layout to convey additional meaning to their code by positioning tokens in absolute space, using boxes to separate or categorise fragments of code, and attaching arrows to boxes to create code that exists visually out of flow from other code on

the same Frame. We also discuss the structure of a Frame in SpIDER and how links can be used to provide an abstraction for code positioned on another Frame.

Section 7.2 documented how a programmer could interact with the authoring IDE functionality provided by SpIDER. Through a series of examples, we demonstrated how SpIDER handles syntax highlighting, warnings and errors, content assist and top level Java project artefacts such as packages and classes. Section 7.3 documented how a programmer can use SpIDER to run and debug a Java Project. Providing program arguments, providing content to ‘standard in’, inserting breakpoints, stepping through code and analysing the current state of a running program were all explored. Of particular note is the fact that, when making a request for content assist or inspecting the current state of a running program, the programmer is provided with results constructed entirely out of First Class Citizens.

Sections 7.4 and 7.5 explained the implementation details of novel algorithms present in SpIDER. The flow walker (Section 7.4) is the algorithm that extracts a linear version of code suitable for compilation from the spatially laid out code produced by a programmer using the techniques documented in Section 7.1. The flow walker algorithm is explained as being built out of two components: the within Frame component: responsible for inferring lines, dealing with boxes and out of flow code; and the director component: responsible for directing and stitching together the results from multiple invocations of the within Frame component.

Section 7.5 documented and discussed the implementation of a system designed to re-introduce certain behaviours that programmers are familiar with from traditional IDEs—the magnet system—without limiting the spatial freedom provided by SpIDER. The magnet system provides programmers with the ability to utilise what we have termed flow effects. Flow effects are the mechanisms that trivially occur in traditional IDEs as a result of their use of relative authoring. For example, the ability to insert a blank line and have the surrounding content respond by shuffling down is a flow effect.

11.1.4 Evaluation

Chapters 8–10 each contributed to our evaluation of SpIDER, and provided evidence towards demonstrating that an IDE can be designed and built to allow for Spatial Layout of

code whilst providing a flexible environment for a programmer to work in. In Chapter 8 we detailed an evaluation of the flow walker algorithm. Utilising a quiz format, 18 computer literature high school students were shown snapshots of examples of spatially laid out print statements and were then asked to choose from a set of answers based on what they thought the code would output. Participants were computer literate high school students. The format of the quiz that participants undertook, some examples of the questions posed and the results of the evaluation are documented. Questions in the quiz were modelled as simplified examples of spatial layout that are possible in SpIDER. This initial study showed that participants' opinions of the meaning of spatially laid out code matched the flow walker's implementation more often than not.

Notably, participant agreement with the algorithm declined when presented with questions that contained out of flow code. In order to better understand the nuances surrounding participant agreement with the flow walker algorithm when out of flow was being used, a follow up study was conducted. By examining the results of the initial study, two conjectures (Section 8.2) were formed and evaluated:

3. *The positioning of the arrow when performing out of flow would have a large effect on a participant's intuition of how it functions.*

The flow walker algorithm does not pay attention to the exact positioning of the head of the arrow, only which box it lands in. However the origin of the arrow can be used to pinpoint more precisely where the out of flow operation occurs at. Consequently providing space around and positioning the origin of the arrow in the centre of the surrounding lines would visually communicate its function more accurately.

4. *When creating a chain, people will treat separate boxes that are roughly aligned vertically as belonging to the same column.*

This suggested to us that avoiding this type of layout, by giving each box its own column, would be remove this confusion.

Pairs of questions using out of flow were designed to test these hypotheses. We were able to find evidence supporting the Hypothesis 1 and as a result, established some guidelines

for best practise when authoring out of flow code. These guidelines are documented in Chapter 10.

Chapter 9 used a series of realistic use cases to demonstrate how SpIDER's flow walker algorithm, First Class Citizens, multimedia support and Frame and Linking system can be used to integrate various parts of the development process on a canvas with absolute positioning. Each example purposefully and thoughtfully positioned software development artefacts (code, diagrams etc.) in space so as to communicate information not provided when authoring code in a traditional IDE—a technique we refer to as a Spatial Development Pattern.

Prior to providing examples of Spatial Development Patterns, Section 9.1, under the slogan of 'First, do no harm', used various examples to show how functionality provided by traditional IDEs can be replicated (or at the very least closely approximated) in SpIDER. Each example showed how a high-level concept used in traditional IDEs—hierarchical organisation for example—could be emulated in SpIDER. The design of each transformation used to transition a concept into Spatial Hypermedia was not intended to present as the ideal of SpIDER style, but rather to show the minimal amount of change necessary for a programmer to begin working within SpIDER.

Over sections 9.2 and 9.3 we documented several code examples that utilise Spatial Development Patterns. We organised these into two categories, Expressive Patterns and Process Patterns. An Expressive Pattern arranges content using spatial layout and the Frame and Linking system in order to communicate auxiliary information about the content being represented through a Spatial Metaphor. A Process Pattern is used to communicate progress or change over time, with the goal of improving efficiency. Six Expressive and Four Process Patterns are documented.

Chapter 10 presented a critical discussion of issues pertaining to the design of SpIDER that had become evident during its production and evaluation. These issues were: the potential for ambiguous code layout as a side effect of the flexibility provided by the flow walker; the necessity of careful arrow positioning when performing out of flow that had become evident from prior evaluation; and the unfortunate similarities between the *goto* programming statement and SpIDER's Frame and Linking system. We addressed each of

these issues, proposing software solutions—such as the snap to spot feature—where appropriate, and a combination of best practices and mitigating factors otherwise.

11.1.5 Future Work: Snap to Spot Feature

To expand upon the snap to spot idea, in the studies documented in Chapter 8 we found that participant agreement with the algorithm rose from 48% to 86% when arrows were carefully positioned following the guidelines we provided in Section 10.2. While the benefit of carefully positioned arrows is clear, enforcing specific positioning will limit the flexibility of the authoring environment. In doing so, we would risk negatively limiting the expressive options currently available to the user.

We propose an opt-in snap to spot feature as a solution. While creating an arrow, if the user is holding down a modifier key, they would be visually shown a set of recommended arrow head positions.

As a start to prototyping a solution, we have sketched Figures 11.2 and 11.3. Thatching has been used to show where the suggested snap to spot positions would be if the best practices outlined in Section 10.2 were used. The heavily thatched areas are the ideal suggested positions for arrow heads, the lightly thatched areas show fall-back positions if the ideal positions are not suitable.

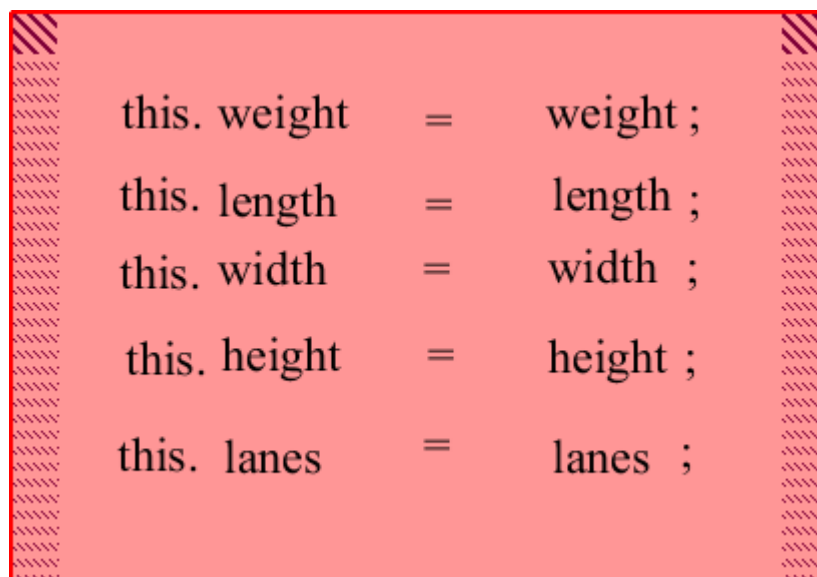


Figure 11.2: Suggested positioning of arrows head when arrows are being created from the left or right of the box.

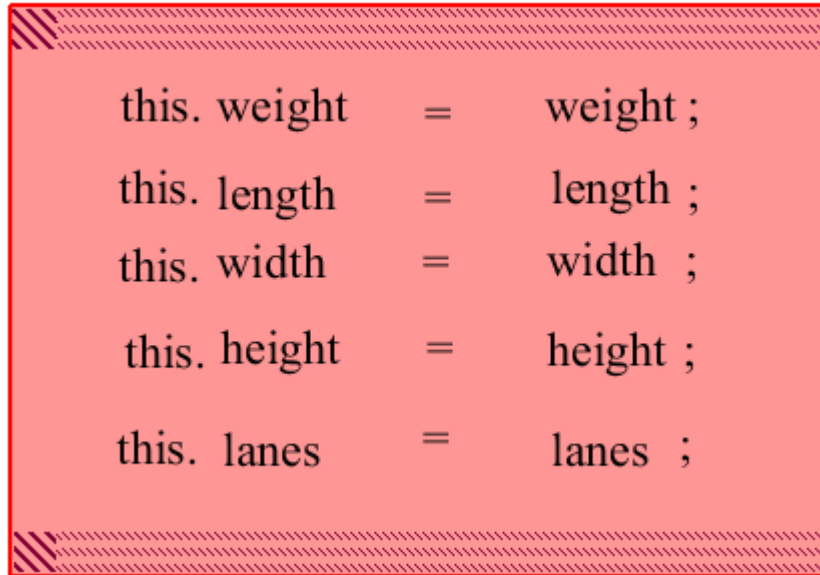


Figure 11.3: Suggested positioning of arrows when arrows are being created from the top or bottom of the box.

11.2 Hypotheses Revisited

Utilising our review and analysis of pertinent literature, we were able to design and build a Spatial Hypermedia environment capable of supporting the software development process. SpIDER features uniform treatment of elements, wide multimedia support, support for a set of linked Frames and spatial layout of code on each Frame. SpIDER's spatial layout has been evaluated and shown to be understandable and useful. We believe that SpIDER—as a first generation Spatial Hypermedia IDE—and the research surrounding it, is promising and worth pursuing further.

In the thesis introduction two hypotheses were posited:

1. A process can be established that allows for spatially arranged code to be unambiguously understood by programmers and compilers. This process should allow code layout practice to range from serial (as seen in conventional IDEs such as Visual Studio), through hierarchical, to diagrammatic.
2. A Spatial Hypermedia-based IDE can be used to integrate many stages of the software development process. The uniform treatment of elements within a Spatial Hypermedia system will allow programmers to intertwine forms of documentation that are traditionally kept separate from the code (test results, variable dumps etc.) and allow programmers to better customise their environment for each project.

Hypothesis 1 was validated through the development of the flow walker algorithm. Examples through the thesis have shown spatially arranged code ranging from close to serial, right through to diagrammatic. Examples of code closer to the serial layout can be seen in Chapter 6, whereas examples closer to the diagrammatic cluster around Chapter 8. Hierarchically laid out code is demonstrated through the use of SpIDER's Frame and Linking system.

The analysis of the flow walker algorithm presented in Chapter 7 has demonstrated that spatially laid out code produced utilising the flow walker can be clearly understood—although with minimal briefing of participants, not 100% unambiguously. The results of this analysis showed confusion, not only between the algorithm and participants, but between participants. Regardless of this, given the purposefully limited complexity of the questions—limited so as to avoid testing code understanding of the programming language syntax—the results gathered are promising; showing that refinement and further pursuit of the spatial code layout is worthwhile.

The ability to integrate many stages of software development, presented in Hypothesis 2, has been demonstrated in Chapter 8—markedly in Section 8.3 where process patterns are examined. Examples have shown SpIDER's use of Expeditee's First Class Citizens being intermixed to visually communicate information about the code while the functionality of that code is retained. Environment customisation has been shown, for example, through the creation of linked Items, enabling navigation directly between code and test code and the ability to create 'toolboxes' for storing frequently used artefacts.

The significance of the research presented in this thesis lies in:

- Its review and subsequently drawn connections between three distinct areas of literature—Spatial Memory, Traditional Programming and Spatial Hypermedia.
- A review of the Spatial Hypermedia system Expeditee.
- The development of SpIDER by extending Expeditee.
- The evaluation of SpIDER's flow walker algorithm.
- An exploration into the possibilities provided by an IDE such as SpIDER.

11.3 Next Steps

We look to the possibility of expanding SpIDER to support other languages. In the following sections we look at aspects for providing this support. Section 11.3.1 examines the Python programming language. In particular, its use of whitespace to represent scope is discussed. Section 11.3.2 then looks at the possibility of using SpIDER's Frame and Linking system as a substitute for programming language functionality. We use an example to demonstrate how SpIDER's Frame and Linking system can be used to 'fake' some aspects of Java-like inheritance in the C programming language. We flesh out these two examples as samplers of what is possible if we push the envelope.

11.3.1 Python

Consider the Python programming language. Python delineates scope by using whitespace characters rather than the curly braces and semi-colons prevalent in many other languages. This reliance on whitespace characters conflicts with SpIDER's absolute positioning; which in effect, does away with whitespace characters.

Prior to the development of SpIDER, Expeditee implemented some support for the Python programming language. This support provides the ability to run Python scripts and give error feedback, but does not include any other IDE functionality. Figure 11.4 shows a Python script that loops through an array and prints its content. In a traditional text editor, the content of the loop would be indented with whitespace. In Expeditee, a box is used to communicate scope. A user runs the *runPythonFrame* action to execute the code. Prior to executing the code, Expeditee serialises the Python content, substituting whitespace wherever boxes are present. Boxes can be used within boxes to represent multiple levels of scope.


```
arr = [ 'entry1', 'entry2', 'entry3', 'entry4', 'entry5']  
concat = "  
for entry in arr:  
    print (entry)  
    concat = concat + entry  
print (concat)
```

Figure 11.4: Python code in Expeditee using boxes in place of whitespace

This is one possible solution to the conflict between Python's reliance on whitespace and SpIDER's lack of whitespace. However, it is a solution that limits the utility of boxes. As boxes currently function in SpIDER, the left and right sides of a Java assignment statement can be placed in separate boxes. If SpIDER's support for Python was to adopt Expeditee's solution, this possibility would not be maintained when authoring Python code.

We expect that other languages, when adding support for them to SpIDER, will each pose their own interesting challenges. How do the ideas expressed in this thesis play out when writing code using a scripting language? What forms of abstraction can be developed that enhance each of these languages? What about a functional language like Haskell? Are language features such as pattern matching suitable for spatial layout? Are there specific Expressive Patterns that are suitable in a functional but not imperative language? These questions are all logical next steps for further research.

11.3.2 Inheritance

Language features, such as inheritance, can be simulated using SpIDER's Frame and Linking system. Figure 11.5 shows a mock-up of how this might look if SpIDER supported the C programming language. Three Frames are visible in the image, each containing some C code. The top Frame contains several linked Text Items that each lead to a Frame containing a function definition. The other two Frames each contain only a single link: to the Frame at the top of the picture. This gives the bottom two Frames access to the

functions declared in the top Frame. Taken together, these three C code files can be used to produce ASCII triangles and boxes.

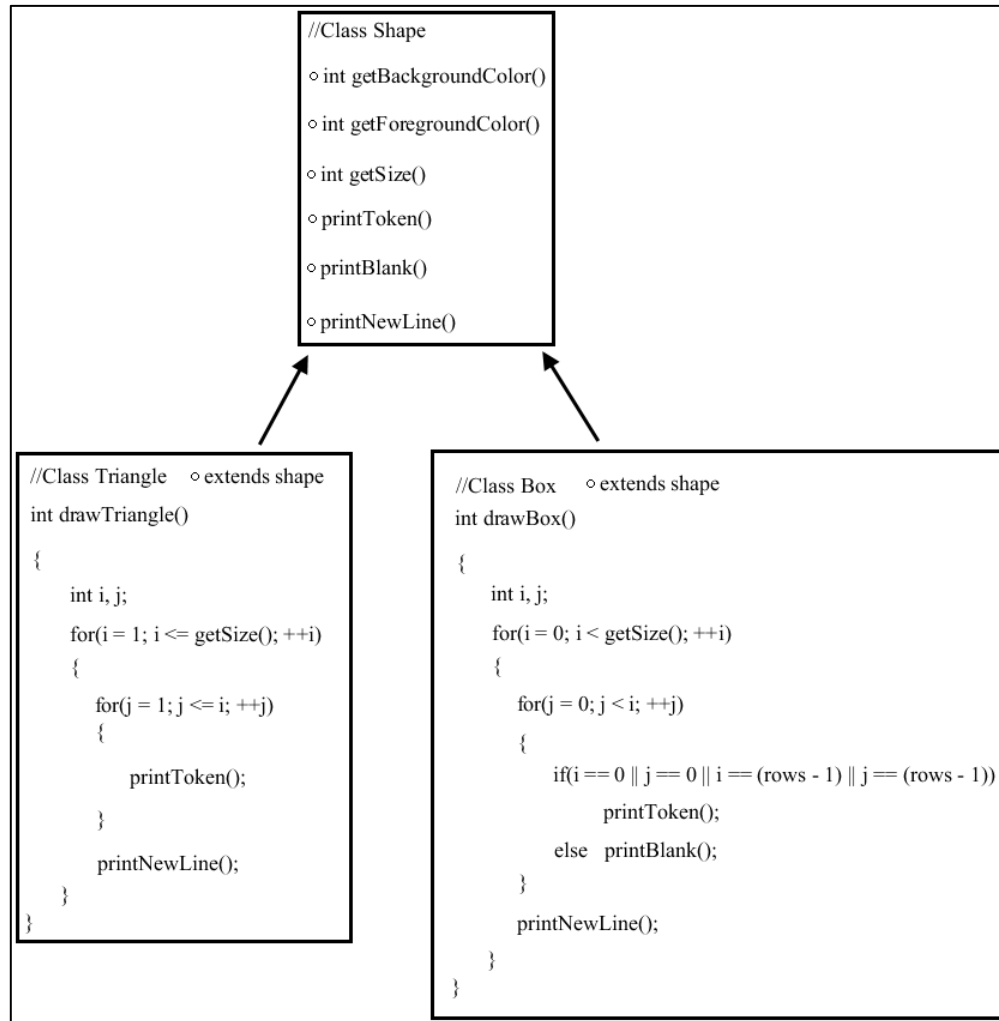


Figure 11.5: Two C code Frame referencing the same Frame which contains a set of ‘inherited’ functions.

This examples shows only one aspect of inheritance: shared content. Further thought is required to determine if other aspects, such as polymorphic referencing, are also possible using SpIDER’s Frame and Linking system. What about other programming language functionality? Can pattern matching be added to Java?

11.3.3 Spatial Programming Languages

Now that we have demonstrated that we can use a Spatial IDE to enhance the presentation of code in conventional programming languages, a natural next step is to consider the

concept of a truly spatial programming language. In the educational sphere, for example, we could go beyond how visual programming languages, such as Scratch [84], are used to teach programming by allowing instructors to show diagrams, taken from textbooks, that also run as active code.

As a more specific example, consider a language where spatial behaviour is syntactically meaningful and multimedia artefacts are treated as First Class citizens. Figure 11.6 shows code written in such a theoretical programming language. Two Frames are shown. In the left Frame we have fairly standard code logging the state of a traffic light before it changes. Notice however, that the variable name *lights* is assigned to a multimedia object—an image of traffic lights in this case.

The functional behaviour of the traffic light image, as a First Class Citizen of the programming language rather than simply an image, becomes apparent when the Frame on the right is examined. A set of arrows and boxes, also with functional behaviour as First Class Citizens, implements the functionality provided by the traffic lights image. They describe a cyclical enumeration, where the active light cycles from red, to green, to orange before arriving back at red. An arrow with a square on its tail is used to specify the initial state of the traffic light enumeration.

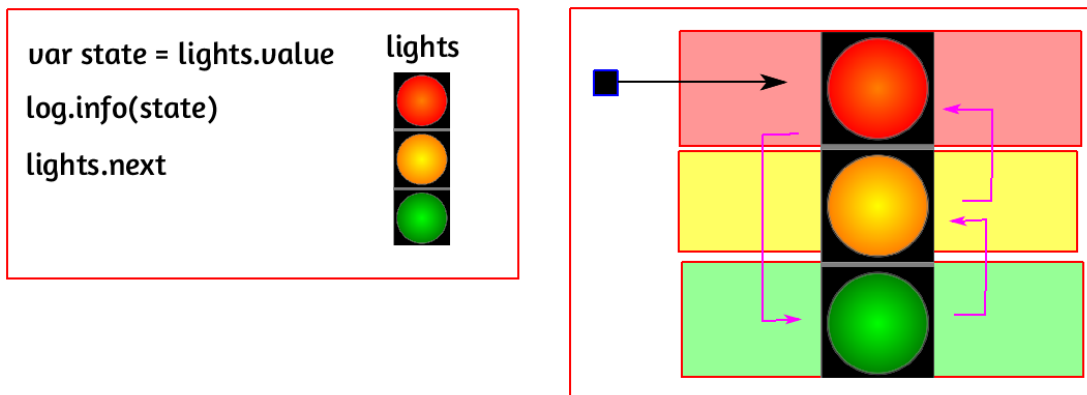


Figure 11.6: Example from theoretical spatial programming language.

References

- [1] G. L. Allen, *Human Spatial Memory: Remembering Where*, Mahwah, New Jersey: Lawrence Erlbaum Associates, Inc., Publishers, 2004.
- [2] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification.," *IBM Systems Journal*, vol. 41, no. 1, pp. 4-12, 2002.
- [3] J. Andrade and P. R. Meudell, "Is spatial information encoded automatically in memory?," *Quarterly Journal of Experimental Psychology*, vol. 46, no. 2, pp. 365-375, 1993.
- [4] C. F. Doeller and N. Burgess, "Distinct error-correcting and incidental learning of location relative to landmarks and boundaries.," *Proceedings of the National Academy of Sciences*, vol. 105, no. 15, pp. 5909-5914, 2005.
- [5] J. Scarr, A. Cockburn and C. Gutwin, "Supporting and Exploiting Spatial Memory in User Interfaces," *Foundations and Trends in Human-Computer Interaction*, vol. 6, no. 1, pp. 1-84, 2013.
- [6] E. Johnson, "A Study of the Effects of Immersion on Short-term Spatial Memory," 2010.
- [7] A. Baddeley and S. D. Sala, "Working memory and executive control," *Philosophical Transactions: Biological Sciences*, vol. 351, no. 1346, pp. 1397-1404, 1996.
- [8] A. Baddeley, "The episodic buffer: a new component of working memory?," *Trends in cognitive sciences*, vol. 4, no. 11, pp. 417-423, 2000.

- [9] M. M. Chun and Y. Jiang, "Contextual Cueing: Implicit Learning and Memory of Visual Context Guides Spatial Attention," *Cognitive psychology*, vol. 36, no. 1, pp. 28-71, 1998.
- [10] R. P. Darken and J. L. Sibert, "Wayfinding Strategies and Behaviors in Large Virtual Worlds," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, Vancouver, BC, Canada, 1996.
- [11] A. W. Siegel and S. H. White, "The Development of Spatial Representations of Large-Scale Environments," *Advances in child development and behavior*, vol. 10, pp. 9-55, 1977.
- [12] R. DeLine, M. Czerwinski, B. Meyers, G. Venolia, S. Drucker and G. Robertson, "Code Thumbnails: Using Spatial Memory to Navigate Source Code," in *Visual Languages and Human-Centric Computing*, Brighton, UK, 2006.
- [13] W. Mou, C. Xiao and T. P. McNamara, "Reference directions and reference objects in spatial memory of a briefly viewed layout.," *Cognition*, vol. 108, no. 1, pp. 136-154, 2008.
- [14] V. Kaptelinin, "Item Recognition in Menu Selection: The Effect of Practice," in *Companion on Human Factors in Computing Systems*, Amsterdam, Netherlands, 1993.
- [15] J. Scarr, A. Cockburn, C. Gutwin and A. Bunt, "Improving Command Selection with CommandMaps," in *SIGCHI Conference on Human Factors in Computing Systems*, Austin, Texas, 2012.
- [16] K. O'Hara and A. Sellen, "A Comparison of Reading Paper and On-Line Documents," in *SIGCHI Conference on Human Factors in Computing Systems*, Atlanta, GA, USA, 1997.
- [17] A. Cockburn, C. Gutwin and J. Alexander, "Faster Document Navigation with Space-Filling Thumbnails," in *SIGCHI Conference on Human Factors in Computing Systems*, New York, USA, 2006.
- [18] C. G. Gutwin and A. Cockburn, "Improving List Revisitation with ListMaps," in *Advanced visual interfaces*, New York, USA, 2006.

- [19] A. Kuhn, D. Erni and O. Nierstrasz, "Embedding spatial software visualization in the IDE: an exploratory study.," in *Proceedings of the 5th international symposium on Software visualization.*, Salt Lake City, UT, USA, 2010.
- [20] R. Minelli, A. Mocci and M. Lanza, "I know what you did last summer: an investigation of how developers spend their time.," in *International Conference of Program Comprehension*, Florence, Italy, 2015.
- [21] G. Goth, "Beware the March of this IDE: Eclipse is overshadowing other tool technologies," *IEEE Software*, vol. 22, no. 4, pp. 108-111, 2005.
- [22] S. Monsell, "Task Switching," *Trends in Cognitive Sciences*, vol. 7, no. 3, pp. 134-140, 2003.
- [23] B. Shneiderman, "A Model Programming Environment*," *Advances in human-computer interaction*, vol. 1, pp. 105-130, 1985.
- [24] P. Carbonnelle, "TOP IDE index," Pierre Carbonnelle, January 2018. [Online]. Available: <http://pypl.github.io/IDE.html>. [Accessed 18 January 2018].
- [25] Object Technology International, Inc, "Eclipse Platform Technical Overview," 2003.
- [26] International Business Machines Corp, "Eclipse Platform Technical Overview," 2006.
- [27] C. Lüer, "Evaluating the Eclipse Platform as a Composition Environment," in *International Conference on Software Engineering*, Portland, Oregon, 2003.
- [28] A. Sarkar, "The impact of syntax colouring on program comprehension," in *Conference of the psychology of programming interest group*, Brighton, United Kingdom, 2015.
- [29] R. P. Buse and W. Weimer, "Learning a Metric for Code Readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546-558, 2010.
- [30] M. Allamanis and C. Sutton, "Mining Source Code Repositories at Massive Scale using Language Modeling," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, San Francisco, 2013.

- [31] L. Francisco-Revilla, "Mainstream Spatial Hypermedia," in *Symposium on Interactive Visual Information Collections and Activity (IVICA)*, Austin, Texas, USA, 2009.
- [32] D. Kolb, "Other Spaces for Spatial Hypertext," *Journal of Digital Information*, vol. 10.3, 2009.
- [33] C. Strachey, "Fundamental concepts in programming languages.," *Higher-order and symbolic computation*, vol. 13, no. 1, pp. 11-49, 2000.
- [34] Unknown, "W3Schools: JavaScript HTML DOM," Refsnes Data, 1998-2017. [Online]. Available: http://www.w3schools.com/js/js_htmlDOM.asp. [Accessed 18 January 2018].
- [35] D. Goodman, *The Complete HyperCard Handbook*, Bantam Books, 1987.
- [36] C. C. Marshall and F. M. Shipman III, "Spatial Hypertext: Designing for Change," *Communications of the CMD*, vol. 38, no. 8, pp. 88-97, 1995.
- [37] C. C. Marshall, F. M. Shipman III and J. H. Coombs, "VIKI: Spatial hypertext supporting emergent structure," in *ACM European conference on Hypermedia technology*, Edinburgh, United Kingdom, 1994.
- [38] F. M. Shipman III, H. Hsieh, P. Maloor and M. Moore, "The visual knowledge builder: a second generation spatial hypertext.," in *Proceedings of the 12th ACM conference on Hypertext and Hypermedia*, Aarhus, Denmark, 2001.
- [39] F. M. Shipman III, H. Hsieh, R. Airhart, P. Maloor, J. M. Moore and D. Shah, "Emergent Structure in Analytic Workspaces: Design and Use of the Visual Knowledge Builder.," in *INTERACT*, Tokyo, Japan, 2001.
- [40] F. Shipman, R. Airhart, H. Hsieh, P. Maloor, J. M. Moore and D. Shah, "Visual and spatial communication and task organization using the visual knowledge builder.," in *Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work*, Boulder, CO, USA, 2001.
- [41] R. M. Akscyn, D. L. McCracken and E. A. Yoder, "KMS: a distributed hypermedia system for managing knowledge in organizations," *Communications of the ACM*, vol. 31, no. 7,

pp. 820-835, 1988.

- [42] C. C. Marshall and F. M. Shipman III, "Spatial hypertext and the practice of information triage.," in *ACM conference on Hypertext*, Southampton, United Kingdom, 1997.
- [43] D. Bainbridge, B. J. Novak and S. J. Cunningham, "A spatial hypertext-based, personal digital library for capturing and organizing musical moments," *International Journal on Digital Libraries*, vol. 12, no. 2-3, pp. 89-103, 2012.
- [44] M. Walmsley, "Expeditee for Software Engineering Students," University of Waikato ENGG492-08Y, Hamilton, 2008.
- [45] B. C. Smith, "Procedural Reflection in Programming Languages (Volume 1)," Massachusetts Institute of Technology, Massachusetts, 1982.
- [46] R. M. Akscyn and D. L. McCracken, "ZOG and the USS Carl Vinson: Lessons in system development.," Carnegie Mellon University, Pittsburgh, 1984.
- [47] R. Akscyn, E. Yoder and D. McCracken, "The Data Model is the Heart of Interface Design," in *Conference on Human Factors in Computing Systems (CHI)*, Washington, DC, USA, 1988.
- [48] R. M. Akscyn, *Private Discussion concerning Expeditee*, 2017.
- [49] E. Yoder, R. Akscyn and D. McCracken, "Collaboration in KMS, A Shared Hypermedia System," in *Conference on Human Factors in Computing Systems*, Austin, 1989.
- [50] R. M. Akscyn and D. L. McCracken, "Design of Hypermedia Script Languages: The KMS Experience," in *Proceedings of the fifth ACM conference on Hypertext*, Pittsburgh, 1993.
- [51] R. DeLine and K. Rowan, "Code canvas: zooming towards better development environments," in *32nd ACM/IEEE International Conference on Software Engineering*, Cape Town, South Africa, 2010.
- [52] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra and J. J. LaViola Jr., "Code bubbles: a working set-based interface for code understanding and maintenance.," in *Conference on Human Factors in Computing*

Systems, Atlanta, GA, USA, 2010.

- [53] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra and J. J. LaViola Jr, "Code bubbles: rethinking the user interface paradigm of integrated development environments," in *ACM/IEEE International Conference on Software Engineering*, Cape Town, South Africa, 2010.
- [54] W. Kuhn and B. Blumenthal, "Spatialization: spatial metaphors for user interfaces.," in *Human factors in computing systems*, Vancouver, 1996.
- [55] R. DeLine, G. Venolia and K. Rowan, "Software development with code maps," *Communications of the ACM* 53.8 (2010), vol. 53, no. 8, pp. 48-54, 2010.
- [56] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen and S. P. Reiss, "Debugger Canvas: Industrial Experience with the Code Bubbles Paradigm," in *34th International Conference on Software Engineering*, Zurich, Switzerland, 2012.
- [57] V. Silva, *Practicle Eclipse Rich Client Platform Projects*, New York: Apress, 2009.
- [58] R. Sedgewick, *Algorithms In C++ Third Edition*, Addison-Wesley, 1998.
- [59] D. E. Knuth, "Literate Programming," *The Computer Journal*, vol. 27, no. 2, pp. 97-111, 1984.
- [60] H. Sharp and H. Robinson, "Collaboration and co-ordination in mature eXtreme programming teams.," *International Journal of Human-Computer Studies*, vol. 66, no. 7, pp. 506-518, 2008.
- [61] F. P. Brooks Jr, *The Mythical Man-Month*, Addison-Wesley, 1995 (1975 Reprint).
- [62] J. L. Schnase and J. J. Leggett, "Computational hypertext in biological modelling.," in *Proceedings of the second annual ACM conference on Hypertext*, Pittsburgh, Pennsylvania, USA, 1989.
- [63] D. L. McCracken and R. M. Akscyn, "Experience with the ZOG human-computer interface system.," *International Journal of Man-Machine Studies*, vol. 21, no. 4, pp.

293-310, 1984.

- [64] C. Parnin, G. Carsten and S. Rugaber, "CodePad: interactive spaces for maintaining concentration in programming environments.," in *Proceedings of the 5th international symposium on Software visualization*, Salt Lake City, UT, USA, 2010.
- [65] S. McConnell, *Code Complete*, Microsoft Press, 2004.
- [66] J. Bloch, *Effective Java Programming Language Guide*, New Jersey: Pearson Tech Group, 2001.
- [67] M. Fowler and J. Highsmith, "The Agile Manifesto," *Software Development*, vol. 9, no. 8, pp. 28-35, 2001.
- [68] K. Schwaber, "Scrum development process.," in *Business object design and implementation*, London, Springer, 1997, pp. 117-134.
- [69] S. Balaji and M. Murugaiyan, "Waterfall vs. V-Model vs. Agile: A comparative study on SDLC.," *International Journal of Information Technology and Business Management*, vol. 2, no. 1, pp. 26-30, 2012.
- [70] Oracle, "Code Conventions for the Java Programming Language," 1999 (April 20).
- [71] M. Allamanis, E. T. Barr, C. Bird and C. Sutton, "Learning Natural Coding Conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Hong Kong, 2014.
- [72] S. Greenberg and M. Roseman, "Using a Room Metaphor to Ease Transitions in Groupware," in *Sharing expertise: Beyond knowledge management*, London, The MIT Press, 2003, pp. 203-256.
- [73] A. A. diSessa and H. Abelson, "Boxer: a reconstructible computational medium," *Communications of the ACM*, vol. 29, no. 9, pp. 859-868, 1986.
- [74] E. W. Dijkstra, "Letters to the editor: go to statement considered harmful," *Communications of the ACM*, vol. 11, no. 3, pp. 147-148, 1968.

- [75] W. A. Wulf, "A case against the GOTO.," *ACM annual conference*, vol. 2, pp. 791-797, 1972.
- [76] D. E. Knuth, "Structured Programming with go to Statements.," *ACM Computing Surveys*, vol. 6, no. 4, pp. 261-301, 1974.
- [77] O.-J. Dahl, E. W. Dijkstra and C. A. R. Hoare, *Structured programming*, Academic Press, 1972.
- [78] H. D. Mills, "Structured programming: Retrospect and prospect," *IEEE Software*, vol. 3, no. 6, pp. 58-66, 1986.
- [79] D. Gries, "On Structured Programming," in *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*, Springer-Verlag, 1978, pp. 70-79.
- [80] G. A. Miller, "The Magical Number Seven, Plus or Minus Two," *Psychological review* , vol. 63, no. 2, p. 81, 1956.
- [81] F. Gobet, P. C. Lane, S. Croker, P. C.-H. Cheng, G. Jones, I. Oliver and J. M. Pine, "Chunking mechanisms in human learning," *TRENDS in Cognitive Sciences*, vol. 5, no. 6, pp. 236-243, 2001.
- [82] D. E. Egan and B. J. Schwartz, "Chunking in recall of symbolic drawings.," *Memory & Cognition*, vol. 7, no. 2, pp. 149-158, 1979.
- [83] W. G. Chase and H. A. Simon, "Perception in chess.," *Cognitive psychology*, vol. 4, no. 1, pp. 55-81, 1973.
- [84] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman and Y. Kafai, "Scratch: programming for all.," *Communications of the ACM*, vol. 52, no. 11, pp. 60-67, 2009.

Appendix A

Evaluating SpIDER Spatial Layout: Questions

A.1 Questions from Initial Study

A.1.1 Part 1

For each question in part 1 participants were asked:

If this pseudocode executed, what would the output look like?

They were provided with five options:

1. a e d c b f
2. a d b e c f
3. a b c d e f
4. b a d e c f
5. a d e b c f

Question 1:

```
print "a";    print "d";    print "b";  
print "e";    print "c";    print "f";
```

Question 2:

```
print "a";    print "d";    print "b";  
print "e";    print "c";    print "f";
```

Question 3:

```
print "a";  
  
print "d";    print "b";  
  
print "e";    print "c";    print "f";
```

Question 4:

print "a";	print "d";
print "b";	print "e";
print "c";	print "f";

A.1.2 Part 2

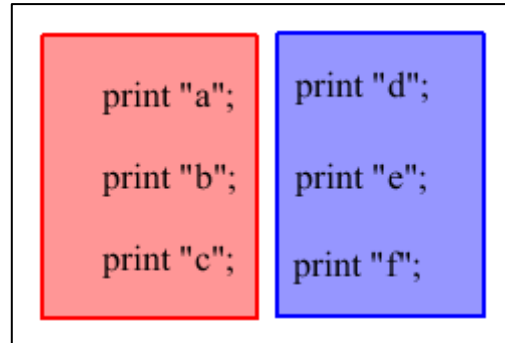
For each question in part 2 participants were asked:

If this pseudocode executed, what would the output look like?

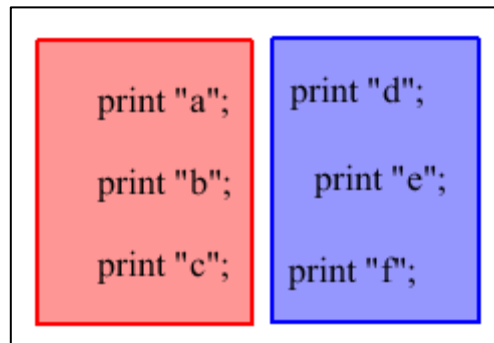
They were provided with six options:

1. f a d b e c
2. d e f a b c
3. f a b c d e
4. a b c d e f
5. a d b e c f
6. a b c d f e

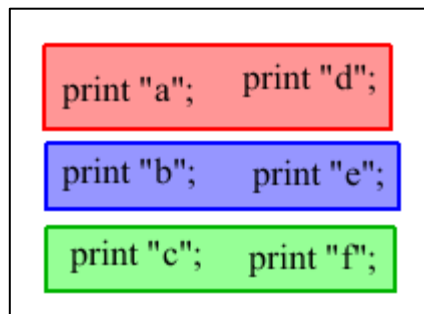
Question 1:



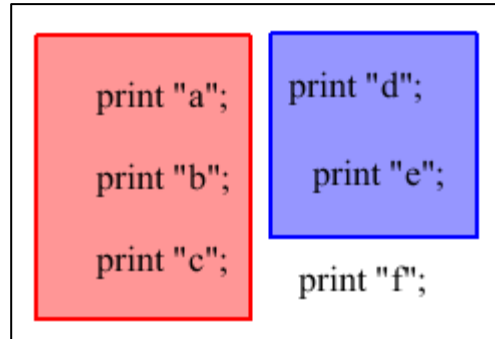
Question 2:



Question 3:



Question 4:



A.1.3 Part 3

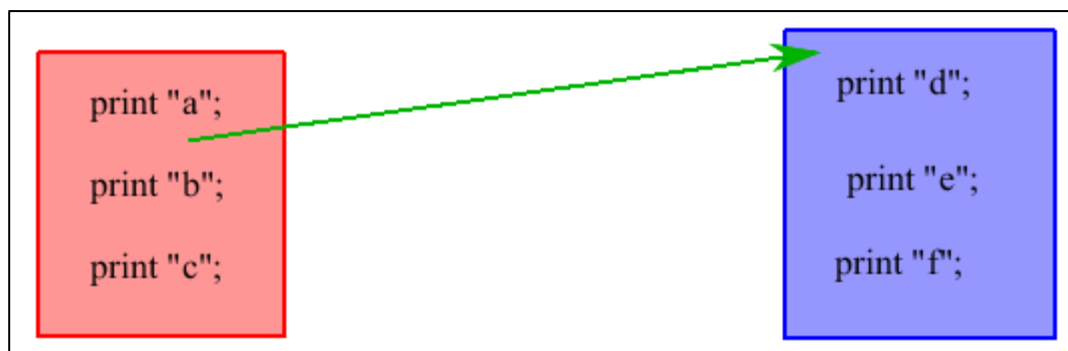
For each question in part 3 participants were asked:

If this pseudocode executed, what would the output look like?

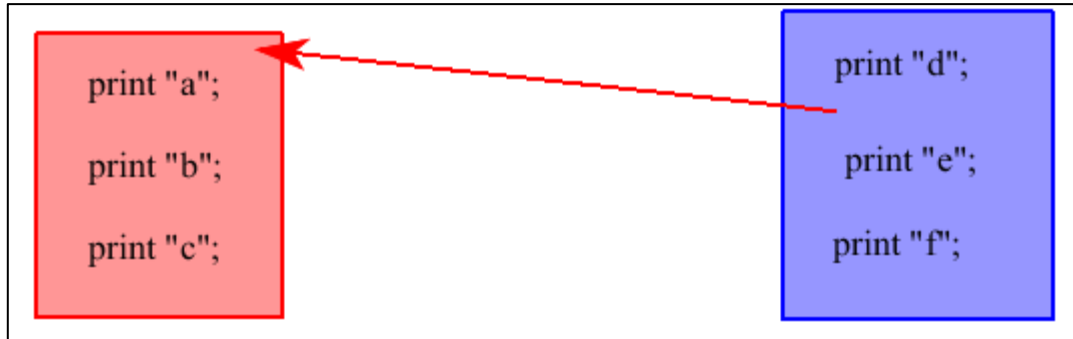
They were provided with six options:

1. d a b c e f
2. a e f b c d e f
3. a d e f b c
4. a b c d e f
5. d e f a b c
6. a b c d e f a b c

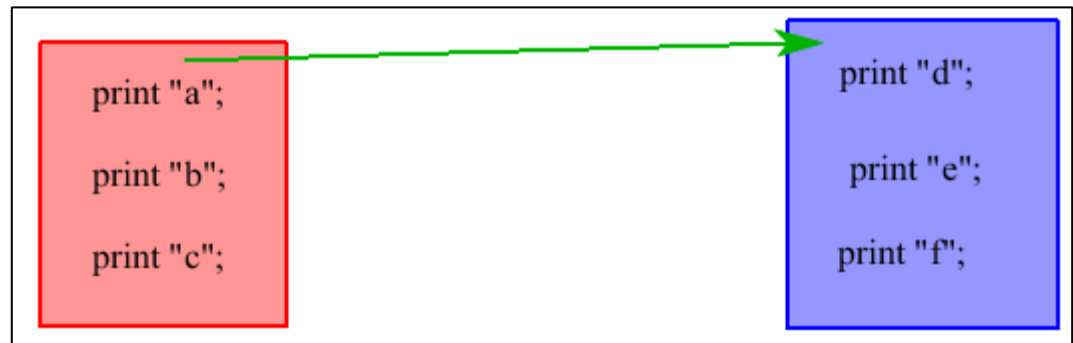
Question 1:



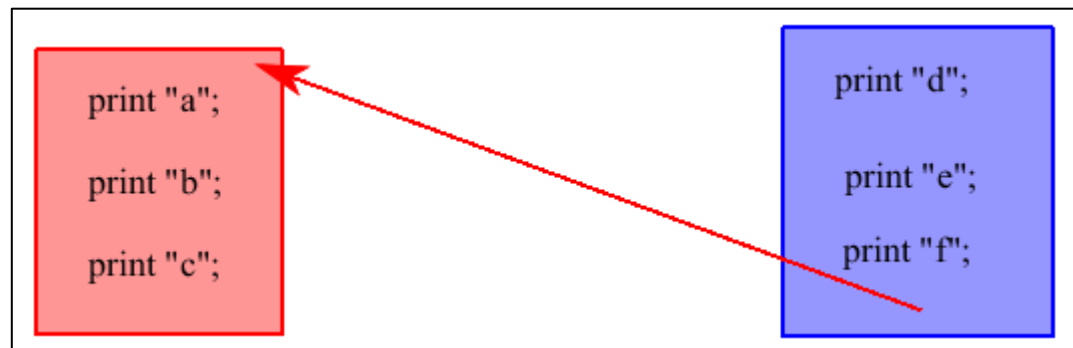
Question 2:



Question 3:



Question 4:



A.1.4 Part 4

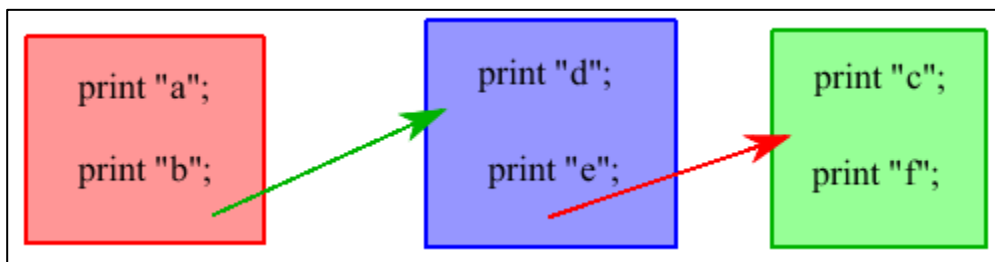
For each question in part 4 participants were asked:

If this pseudocode executed, what would the output look like?

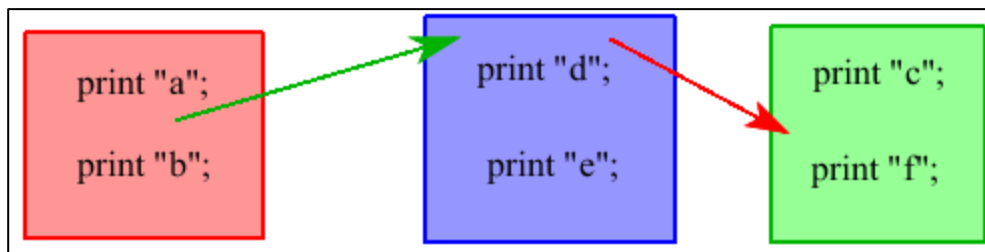
They were provided with eight options:

1. c a b d e f
2. a b d e a b c f
3. c d a b e f
4. a d e c f b
5. a b d e c f
6. d e c f a b
7. a c f d e b
8. a b d e c f c f

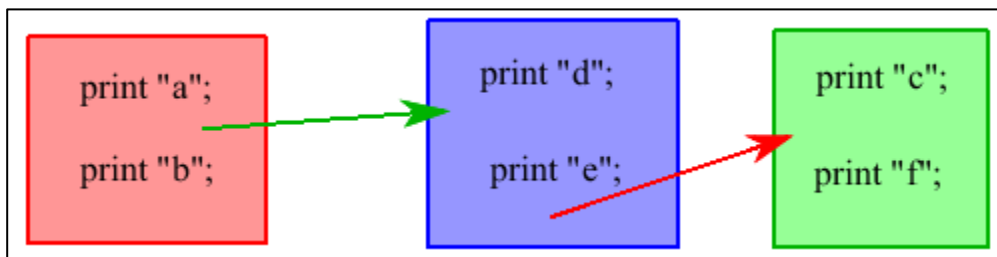
Question 1:



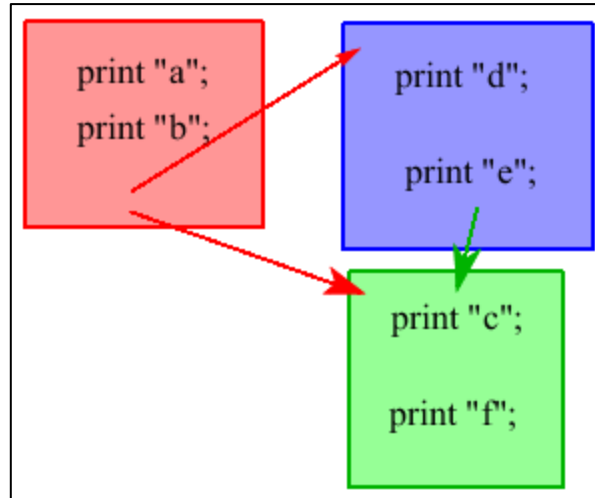
Question 2:



Question 3:



Question 4:



A.1.5 Part 5

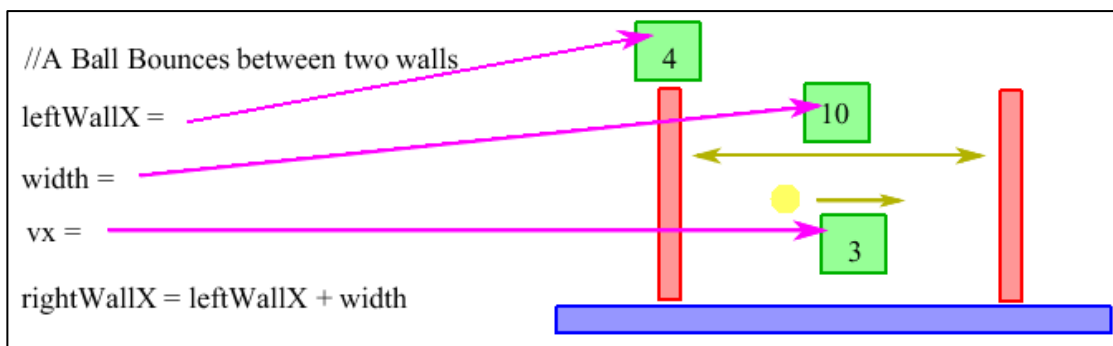
Question 1:

Participants were asked:

Below is the first part of a program snippet that models a ball bouncing between two walls. What is the x-position of the right-hand wall?

They were provided with five options:

1. 4
2. 14
3. 10
4. V_x
5. 17



Question 2:

Participants were asked:

Here is the complete snippet. At the end of the 3rd iteration through the loop, which walls have been hit?

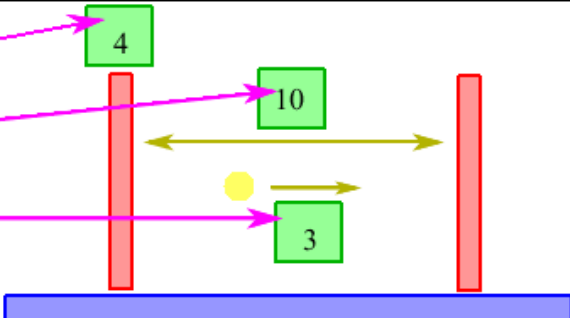
They were provided with four options:

1. Left wall only
2. Right wall only
3. Neither wall
4. Both walls

```
//A Ball Bounces between two walls
leftWallX = 
width = 
vx = 
rightWallX = leftWallX + width

x = leftWallX + 2

while true
  x = x + vx
  if x > rightWallX
    x = rightWallX
    vx = -vx
  if x < leftWallX
    x = leftWallX
```



The diagram illustrates a ball bouncing between two walls. A yellow ball is positioned between two red vertical walls on a blue floor. Green boxes with numbers 4, 10, and 3 are placed at the left wall, the ball, and the right wall respectively. A double-headed arrow indicates the distance between the walls. A red box labeled $vx = -vx$ is shown with arrows pointing to the `if` conditions in the code snippet.

A.1.6 Part 6

Question 1:

Participants were asked:

Here is a function for doing some hit testing on a square. We are interested if it is the left or the right side of the square has been hit, and if it is the right-hand side, whether it is towards the top or the bottom. If `hitSquare(8,3,10)` is executed, what will print?

They were provided with four options:

1. l
2. ll
3. r t
4. r b

//Hit box testing: which part of the square are we hitting?

```
func hitSquare(x, y, size)
```

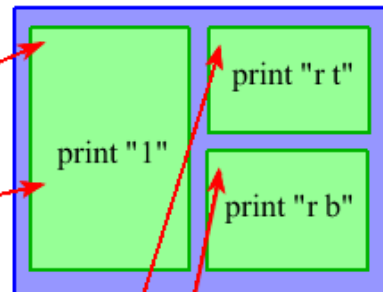
```
    half = size / 2
```

```
    if x < half AND y < half
```

```
    else if x < half AND y >= half
```

```
    else if x >= half AND y < half
```

```
    else
```



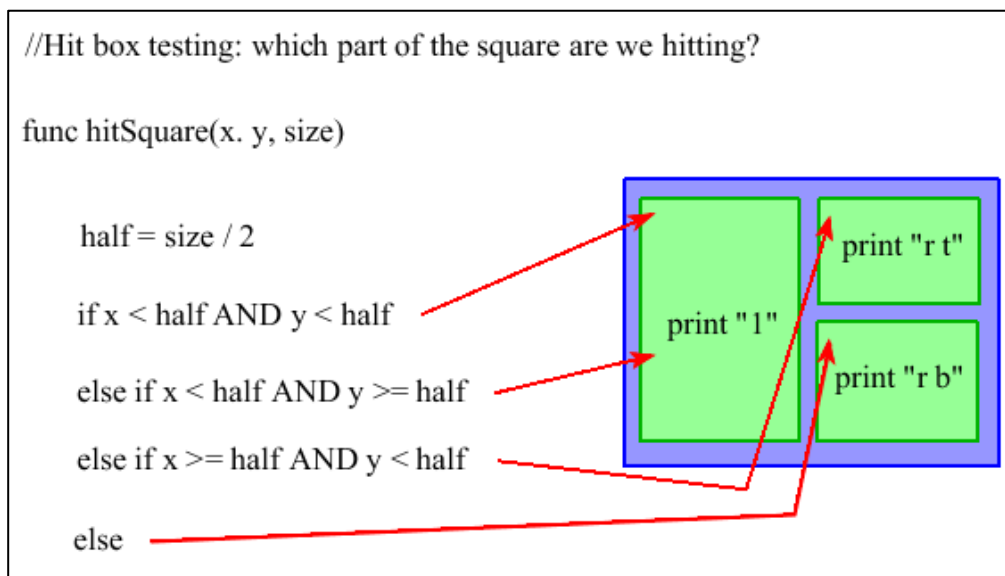
Question 2:

Participants were asked:

Here is the same code again. If `hitSquare(3, 7, 10)` is executed, what will print?

They were provided with four options:

1. l
2. ll
3. r t
4. r b



Appendix B

Evaluating Spatial Layout: Confidence Level

B.1 Initial Study Confidence Level

The following concerns the first four parts of the initial study. The average (mean) agreement for each section, respectively, is: 72%, 78%, 47% and 33%. We state that overall agreement with the algorithm is: 57.5% \pm 17.39%. This is calculated as follows:

- The overall average of 57.5% is calculated with the following formula:
 $\Sigma(\text{averages}) \div N$, where N is 4 (the number of individual means).
- The standard deviation is used to calculate the margin of error. It is calculated using the following formula: $\sqrt{\Sigma((x - \mu)^2 \div (N - 1))}$. Figure B.1 shows this equation being applied to the individual means listed above.
- The margin of error of 17.39% is calculated with the following formula: $z * (\sigma \div \sqrt{N})$, where z is 1.645 so as to attain a confidence level of 90% and σ is the previously calculated standard deviation (restored to a percent).

$$\begin{aligned} (0.72 - 0.575)^2 &= 0.021025 \\ (0.78 - 0.575)^2 &= 0.042025 \\ (0.47 - 0.575)^2 &= 0.011025 \\ (0.33 - 0.575)^2 &= 0.060025 \\ \\ \text{std dev} &= \sqrt{\text{sum(above)} / (4 - 1)} \\ &= \sqrt{0.1341 / 3} \\ &= \sqrt{0.0447} \\ &= 0.2114 \end{aligned}$$

Figure B.1: Calculating the standard deviation of the individual means.

Appendix C

Ethics Application and Approval

C.1 Ethics Application

Project Title

Understanding SpIDER Spatial Layout

What is this research project about?

SpIDER (Spatial Integrated Development Environment Research) is an IDE developed with the goal of improving the programming experience. It aims to do this by allowing the programmer to lay code out spatially; using both the height and width of the page; to communicate meaning.

Programming languages are ridged and precise. Traditional IDEs retain this rigidity through the use of a flat file text editor in combination with specially built widgets (such as the Package Explorer in Eclipse). SpIDER attempts to shake off this rigidity in various ways, one of which is through the use of its 'out of flow' walker. The programmer is able to use spatial positioning, boxing and arrows in order to communicate meaning. For example, the programmer may want to emphasize the base and recursive cases in a recursive function. The 'out of flow' walker works as a middle man, decoupling the code the programmer writes from what the compiler is given.

Purpose

The 'out of flow' walker has been designed with the aim of being mostly invisible to the programmer. The programmer should be able to quickly understand how SpIDER allows code to be laid out. A quiz has been designed to show some code laid out in a SpIDER style for participants to look at and understand.

What will you have to do and how long will it take?

You will be given a survey to complete online, it will take no more than 30 minutes.

What will happen to the information collected?

The information collected will be used by the researcher to craft a specific section of their Doctoral thesis. Only the researcher and supervisors will be privy to any notes and raw data collected. After analysis is complete any notes and raw data collected will be destroyed/erased. Participants can request their own results and receive them once analysis is complete. No participants will be named in the publications and every effort will be made to disguise their identity.

Declaration to participants

If you take part in the study, you have the right to:

- Refuse to answer any particular question, and to withdraw from the study before analysis has commenced on the data.
- Ask any further questions about the study that occurs to you during your participation.
- Be given access to your own answers once analysis is complete.
- Be given access to a summary of findings from the study once analysis is complete.

Who's responsible?

If you have any questions or concerns about the project, either now or in the future, please feel free to contact either:

Researcher: Bryce

bnemhauser@gmail.com

Supervisor: David Bainbridge

davidb@waikato.ac.nz

Supervisor: Bill Rogers

coms0108@cs.waikato.ac.nz

Research Consent Form



Ethics Committee, Faculty of Computing and Mathematical Sciences

Understanding SpIDER Spatial Layout

Consent Form for Participants

I have read the **Participant Information Sheet** for this study and have had the details of the study explained to me. My questions about the study have been answered to my satisfaction, and I understand that I may ask further questions at any time.

I also understand that I am free to withdraw from the study before the end of the day. I understand I can withdraw any information I have provided up until the researcher has commenced analysis on my data. I agree to provide information to the researchers under the conditions of confidentiality set out on the **Participant Information Sheet**.

I agree to participate in this study under the conditions set out in the **Participant Information Sheet**.

Signed: _____

Name: _____

Date: _____

Additional Consent as Required

I agree / do not agree to my responses being recorded.

Signed: _____

Name: _____

Date: _____

Researcher's Name and contact information:

Researcher: Bryce
bnemhauser@gmail.com

Supervisor's Name and contact information:

Supervisor: David Bainbridge

davidb@waikato.ac.nz

Supervisor: Bill Rogers

coms0108@cs.waikato.ac.nz

C.2 Ethics Approval

Faculty of Computing and
Mathematical Sciences
Rorohiko me ngā Pūtaiao Pēngarau
The University of Waikato
Private Bag 3105
Hamilton
New Zealand

Phone +64 7 838 4322
www.cms.waikato.ac.nz



2 September 2016

Bryce Nemhauser
C/- Department of Computer Science
THE UNIVERSITY OF WAIKATO

Dear Bryce

Request for approval to conduct a user study with human participants

On the basis of the information you have provided on the FCMS Preliminary Ethics Application Form relating to your PhD research "Understanding SpIDER Spatial Layout", the Committee has given you approval to proceed with your proposed study, with two conditions:

1. There will be a consent form which will be signed by the teacher-in-charge;
2. Data will be kept for up to 5 years, if necessary.

We wish you well with your research.



Mark Apperley
Human Research Ethics Committee
Faculty of Computing and Mathematical Sciences