Visual, Aspect-Oriented tools for Component Pascal in Eclipse

Submitted By:
Abhishek P. Singh.
Bachelor of Information Technology (Software Engineering/Data Communications)
Programming Languages and Systems Research Center (PLAS)
Centre for Information Technology Excellence (CITI)

Principal Supervisor: Dr. Diane Corney
Associate Supervisor: Associate Professor Paul Roe
Associate Supervisor: Professor John Gough

Submitted for:
Masters of Information Technology (Research), 2005.
Abstract

Tools and environments have aided developers in producing software since compilers and editors became standard offerings with operating systems. A major challenge for the tools and environments community has been to find ways to build and integrate tools so that they, or capabilities between them, can be easily adapted for use in new contexts.

The “Eclipse” Project is an open source software development project dedicated to providing a robust, full-featured, commercial-quality, industry platform for the development of highly integrated tools. The mission of the “Eclipse” Project is to adapt and evolve the eclipse technology to meet the needs of the eclipse tool building community and its users, so that the vision of eclipse as an industry platform is realized. “Eclipse” uses an innovative plug-in architecture allowing near-infinite extensions to the base IDE. Unlike other open-source projects that don’t allow proprietary derivative works, “Eclipse” can be extended with proprietary plug-ins, repackaged, and sold commercially.

Aspect-Oriented programming (AOP) is a new programming paradigm based on the idea that computer systems are better programmed by separately specifying the various concerns of a system and some description of their relationships, and then relying on mechanisms in the underlying AOP environment to weave or compose them together into a coherent program. While the tendency in Object-Oriented Programming is to find commonality among classes and push it up in the inheritance tree, AOP attempts to realize scattered concerns as first-class elements, and eject them horizontally from the object structure. The primary goals of this research were

1. Incorporation of “Component Pascal” into “Eclipse”. “Component Pascal” was a command line compiler that is targeted to a variety of platforms including JVM and .NET.
2. Research and design visual programming tools for “Component Pascal” in “Eclipse”, in particular visual tools that support aspect-oriented views of software.

These objectives are now complete and a plug-in has been developed that enables the development of “Component Pascal” software within “Eclipse”. Aspect-Orientation has been incorporated directly into the “Component Pascal” compiler.
# Contents

1 STATEMENT OF ORIGINAL AUTHORSHIP 1

2 ACKNOWLEDGEMENTS 2

3 INTRODUCTION 3

4 RELATION TO PREVIOUS WORK 4

4.1 Software Engineering Tools and Development Environments 4
   4.1.1 Evolution of Software Engineering Tools 4
   4.1.2 Computer Aided Software Engineering (CASE) 5
   4.1.3 Component pascal and the eclipse platform 6
   4.1.4 Related Visual Programming Tools 8

4.2 Future of Software Engineering Tools and Environments 10
   4.2.1 Automatic Programming 10

4.3 Aspect Oriented Technologies 11
   4.3.1 Separation of Concerns 11
   4.3.2 Tyranny of the dominant decomposition 12

4.4 An Assessment of Aspect-Oriented Programming (AOP) 12
   4.4.1 Related works in Aspect-Orientation 13
   4.4.2 Alternatives to Aspect Oriented Programming 14
   4.4.3 Reflection and Meta-object protocols 14
   4.4.4 Subject-oriented programming 14
   4.4.5 Hyperspaces 15

4.5 Related works in eclipse 17
   4.5.1 Integrating the BETA language with eclipse [1] 17
   4.5.2 Leveraging Cognitive Support and Modern Platforms for Adoption-Centric Reverse Engineering (ACRE) [2] 17
   4.5.3 Hipikat: Recommending useful software artifacts [3] 18
   4.5.4 ArchJava IDE [4] 18
   4.5.5 Guard: Relative Debugging [5] 19
   4.5.6 Knowledge Based Reverse Engineering [6] 19
   4.5.7 Obasco (Object, Aspect, Components) [7] 19
   4.5.8 AspectJ, Aspect Oriented Programming (AOP) for Java 20

4.6 Summary 21

5 COMPONENT PASCAL AND ECLIPSE 22

5.1 Component Pascal 22

5.2 Eclipse 22
   5.2.1 Standard Widget Toolkit 24

5.3 Integration into eclipse 24
5.4 Plug-in Features
   5.4.1 Creation of new component pascal resources 26
   5.4.2 Creation of New component pascal Projects 28
     5.4.2.1 Project nature 28
   5.4.3 Creation of new component pascal Modules/Base Modules 31
   5.4.4 Editing of component pascal Modules 33
   5.4.5 Syntax Highlighting 33
   5.4.6 Compilation of component pascal Modules 36
   5.4.7 Error Logging 37
   5.4.8 Resource Navigation 38
   5.4.9 Layout Customization 39
   5.4.10 Project Builds 41
   5.4.11 Project Execution 42
   5.4.12 Content Outline 43
     5.4.12.1 Domain specification for the outline 44
     5.4.12.2 Parsing the component pascal source 45
     5.4.12.3 Inner workings of the content outline 46
     5.4.12.4 Integrating the content outline and editor 47

5.5 Summary 48

6 ASPECT ORIENTED EXTENSIONS 49

6.1 Initial Investigation 49
   6.1.1.1 Base program 51
     The base program is any existing component pascal module which aspect code needs
     to be applied to. 51
   6.1.1.2 Aspects and Advice 51
   6.1.1.3 Join Points and Point-cut Designators 51

6.2 Initial Plan 53

6.3 Aspect Syntax 55
   6.3.1 Features of the aspect module 56
   6.3.2 Point-cut Definition 56

6.4 Current Implementation 58
   6.4.1 Changes to compiler 58
   6.4.2 Aspect Weaving 64
     6.4.2.1 Dependence between base code and aspect code 67
     6.4.2.2 Implementing parameter awareness in advices 69
   6.4.3 A sample application 70
   6.4.4 Support within eclipse 71
     6.4.4.1 Changes to Parser 71
     6.4.4.2 Compilation Support 71

6.5 Summary 73

7 FURTHER WORK 74

7.1 Additional Features 74
   7.1.1 Look-ahead typing 74
   7.1.2 Debugging capability 74
   7.1.3 Preference and Property pages 75
1 Statement of original authorship

The work contained in this thesis has not been previously submitted for a degree or diploma at any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Abhishek P. Singh.

Signature: ________________________________

Date: ________________________________
2 Acknowledgements

I would like to thank my Principal Supervisor, Dr. Diane Corney, for her continued support and patience throughout my degree. Diane’s advice and insights were invaluable without which the completion of this thesis would not have been possible.

During the course of my degree, I got married, changed employers twice, renovated the apartment I was living in and also started my own company. Hence, it has been quite a long though interesting journey.

I would also like to thank my associate supervisors, Prof. John Gough, and Assoc. Prof. Paul Roe for their support.

Lastly, I would also like to thank my family and wife, Chandrika who provided the much required emotional support and inspiration.
3 Introduction

The goal of this project was to research visual tools to support programming using the “component pascal” [20]\(^1\) language within the “eclipse”\(^2\) integrated development environment \(^3\)(IDE). The aim was to support views of “component pascal” software to facilitate its development, maintenance and evolution. The essential idea of this project was to leverage the interface to program source code within the “eclipse” IDE to support a graphical view of “component pascal” software.

My initial objective was to support the JVM implementation of the “component pascal” programming language within “eclipse”, and to research visual tools to support component programming using this language. Some example tools are wizards, design pattern generators and visual component assembly tools.

Aspect-orientation is a relatively new programming paradigm. It is still in its infancy and used mostly in research circles. Aspect-orientation, like any new programming paradigm has still not gained widespread commercial adoption. It was intended to research aspect-orientation with a view to its incorporation into “component pascal”.

The research plan for the project initially had two main phases:

1. Research and design visual programming tools and incorporate “component pascal” into “eclipse”.
2. Research aspect-orientation and aspect-oriented tools in those particular tools that support aspect-oriented views of software, with the view to integrate aspect-orientation into “component pascal”.

At the completion of the project I hoped to have integrated “component pascal” into “eclipse”, and have integrated aspect-orientation into “component pascal”.

\(^1\) Component pascal - Component Pascal is Oberon microsystems’ refinement of the Oberon-2 language. It is specifically designed for programming software components and has enhanced programming safety through an advanced type system.

\(^2\) Eclipse - Eclipse is an open source community whose projects are focused on providing a vendor-neutral open development platform and application frameworks for building software.

\(^3\) Integrated Development Environment - programming environment that has been packaged as an application program, typically consisting of a code editor, a compiler, a debugger, and a graphical user interface builder.
4 Relation to Previous Work

Software engineering tools have been instrumental in the enormous growth of the software industry as they helped reduce the inherent complexity in producing reliable software. There has been a great deal of research into these tools, and their capabilities have increased from simply providing an environment for coding software to supporting the entire software development lifecycle.

Discussed in this chapter is the advent of these tools, their past present and future. We also look at the emergence of development tools based in “eclipse” and assess aspect-oriented programming, a novel-programming paradigm.

4.1 Software Engineering Tools and Development Environments

4.1.1 Evolution of Software Engineering Tools

Tools and environments have aided developers in producing software since compilers and editors became standard offerings with operating systems. Early environments did not provide any real means of integrating tools, coordinating their execution, or automating common tasks. Developers were required to employ appropriate usage conventions that would permit the developers to coordinate the use of the tools.

The first significant efforts in producing integrated development environments were those in the area of Programming Support Environments (PSE). PSEs were collection of tools that supported coding activities and generally provided one or more compilers, language sensitive editors and debuggers. Their major limitation however was that they supported a single software engineering activity and its artifacts. This identified the need for integrated support for software engineering activities throughout the software lifecycle and represented the genesis of Software Engineering Environments (SEE).

SEEs like PSEs are integrated collections of tools that facilitate software engineering activities. However they extend well beyond programming support tools, supporting software engineering across the software lifecycle. This helped broaden the domain across which tools could be applied. SEEs however did not provide support for the software engineering process that produced the software artifacts.
This gave rise to Process-Centered Software Engineering Environments (PSEE), which integrated tool support for software artifact development with support for the modeling and execution of the software engineering processes that were used to produce software artifacts.

A major challenge for the tools and environments community has been to find ways to build and integrate tools so that they can be easily adapted for use in new contexts [13]. Most visual programming environments today are used to build code artifacts and are tailored or best suited to a certain programming language though they can be used to build a diverse range of applications. In section 4.1.4 some of the prominent visual programming tools in use today are discussed.

### 4.1.2 Computer Aided Software Engineering (CASE)

Computer Aided Software Engineering (CASE) broadly refers to tools and environments that support software development activities.

Initially CASE tool developers concentrated to a large extent on the automation of isolated tasks such as document production, version control of source code, and design method support. While successes were achieved in supporting such specific tasks, the need for these *islands of automation* to be connected was recognized. A typical development scenario requires that designs be closely related to their resultant source code, that they be consistently described in a set of documentation, and that all of these artifacts be under centralized version control. The tools that support the individual tasks of design, coding, documentation, and version control must be integrated if they are to support this kind of scenario effectively. This raised the need for *Integrated Development Environments*.

A CASE environment is a collection of CASE tools and other components together with an integration approach that supports most or all of the interactions that occur among the environment components, and between the users of the environment and the environment itself. What distinguishes a CASE environment from a random amalgamation of CASE tools is that there is something that is provided in the environment that facilitates interaction of those tools. This *something* may be a physical mechanism such as a shared database or a message broadcast system, a conceptual notion such as a shared philosophy on tool
architectures or common semantics about the objects the tools manipulate, or some combination of these things.

Such tools are more often used as components in a much more elaborate software development support infrastructure that is available to software engineers. A typical CASE environment consists of a number of CASE tools operating on a common hardware and software platform. There are a number of different classes of users of a CASE environment. Some users, such as software developers and managers, wish to make use of CASE tools to support them in developing application systems and monitoring the progress of a project. On the other hand, tool integrators are responsible for ensuring that the tools operate on the software and hardware platform available and the system administrator’s role is to maintain and update the hardware and software platform itself.

Market pressure for integrated CASE environments eventually resulted in the development of new models of tool integration. Early CASE tools made generalized, tailor-able integrations difficult to achieve. The development of CASE architectures that were more open with respect to tool functions resulted in less egocentric tools [22].

An empirical case study of CASE technology productivity perceptions of developers has previously been carried out [23]. The study set out to investigate productivity perceptions of developers that use CASE technology. In the survey, fifteen CASE technological functions were compared with two behavioral functions. It was concluded that Integrated Development Environments must continue to encompass more of the systems development effort assisting the developers with technological, behavioral and cognitive support.

4.1.3 Component Pascal and the Eclipse platform

“Eclipse” is an open source community whose projects are focused on providing a vendor-neutral open development platform and application frameworks for building software. The Eclipse Foundation is a not-for-profit corporation formed to advance the creation, evolution, promotion, and support of the Eclipse Platform and to cultivate both an open source community and an ecosystem of complementary products, capabilities, and services.
“Eclipse” has formed an independent open eco-system around royalty-free technology and a universal platform for tools integration. “Eclipse” based tools give developers freedom of choice in a multi-language, multi-platform, multi-vendor environment. “Eclipse” provides a plug-in based framework that makes it easier to create, integrate and utilize software tools, saving time and money. By collaborating and exploiting core integration technology, tool producers can leverage platform reuse and concentrate on core competencies to create new development technology. The Eclipse Platform is written in the Java language and comes with extensive plug-in construction toolkits and examples. It has already been deployed on a range of development workstations including Linux, HP-UX, AIX, Solaris, QNX, Mac OS X and Windows based systems.

The “eclipse” platform is designed and built to meet the following requirements:

- Support the construction of a variety of tools for application development.
- Support an unrestricted set of tool providers, including independent software vendors (ISVs).
- Support tools to manipulate arbitrary content types (e.g., HTML, Java, C, JSP, EJB, XML, and GIF).
- Facilitate seamless integration of tools within and across different content types and tool providers.
- Support both GUI and non-GUI-based application development environments.
- Run on a wide range of operating systems, including Windows ® and Linux TM.
- Capitalize on the popularity of the Java programming language for writing tools.

The “eclipse” platform’s principal role is to provide tool providers with mechanisms to use, and rules to follow for building IDEs that lead to seamlessly integrated tools. These mechanisms are exposed via well defined Application Programming Interfaces (APIs), classes, and methods. The Platform also provides useful building blocks and frameworks that facilitate developing new tools.

“Component pascal” [20] is Oberon Microsystems’ refinement of the “Oberon-2” programming language. “Component pascal” is a general-purpose language in the tradition

---

[20] Oberon Microsystems - Component Pascal is a trademark of Oberon Microsystems; Inc. Oberon is a trademark of Prof. Niklaus Wirth, Switzerland. Oberon Microsystems can be contacted at info@oberon.ch.
of “Pascal”, “Modula-2” and “Oberon”. Its most important features are provisions for block structure, modularity, separate compilation, static typing with strong type checking (also across module boundaries), type extension with methods, dynamic loading of modules, and garbage collection.

Type extension makes “component pascal” an object-oriented language. “Component pascal” covers most terms of object-oriented languages by using the established vocabulary of imperative languages in order to minimize the number of notions for similar concepts. Complete type safety and the requirement of a dynamic object model make “component pascal” a component-oriented language.

4.1.4 Related Visual Programming Tools

“AspectJ” [12] is one particular instance of a visual aspect-oriented programming tool, distinguished by the fact that it was designed from the ground up to be compatible with Java. “AspectJ” is a seamless aspect-oriented extension to the Java programming language and is freely available under an open source license. The “AspectJ” release includes a compiler, structure browser and a debugger. “AspectJ” enables the clean modularization of crosscutting concerns such as: error checking and handling, synchronization, context-sensitive behavior, performance optimizations, monitoring and logging, debugging support, multi-object protocols. The “AspectJ” project is based on over ten years of research at Xerox Palo Alto Research Center funded by Xerox, a U.S. Government grant (National Institute of science and technology, advanced technology program), and a Defense Advanced Research Projects Agency (DARPA) contract.

“Visual Studio” [30] is Microsoft's rapid application development tool for building applications. It includes a single integrated development environment (IDE) and provides deep support for multiple programming languages and also performs many common programming tasks automatically. In particular “Visual Studio” automates tasks to such an extent that programmers that are not very skilled can develop complex applications quickly and effectively.
“Visual Age” [31] is a visual programming tool developed by IBM. “Visual Age” allows multiple developers to work on multiple projects, with automatic version control. It is predominantly used to develop Java based applications.

“JBuilder”™ [47] is a comprehensive award-winning visual development environment for building applications, applets and JSP/Servlets as well as JavaBeans™, Enterprise JavaBeans and distributed J2EE applications for the Java 2 Platform. “JBuilder” provides multiple views of code, cross platform development, context-sensitive help, audio feedback, customizable code creation, wizards for rapid application development and visual customization features.

“Serendipity-II” [32] is a visual design environment developed at the Department of Computer Science at the University of Waikato. It was developed out of a need for improved component requirements engineering for developing multiple view, multiple user visual design environments. Aspect-oriented component engineering addresses difficult issues of component requirements engineering by analyzing and characterizing components based on different aspects of an overall system that the component addresses. Aspects help identify, categorize and reason about component requirements. Serendipity-II’s main functional requirements include visual, collaborative process modeling and software agent specification views, process enactment and view modification histories, to-do lists, and agent component information. The key non-functional requirements include supporting novice and experienced users, platform independence and mobile computer support, robustness and security.
4.2 Future of Software Engineering Tools and Environments

4.2.1 Automatic Programming

Automatic Programming [24] has long been a goal of computer science and artificial intelligence. It is defined as the synthesis of a program from a specification. Through this technology the generation of executable programs from specifications is made possible. Although compilers could be considered to perform this task, the specifications are generally at a higher level than ordinary programming languages.

Automatic programming systems will have three key features: They will be end-user oriented, communicating directly with end users; they will be general purpose, working as well in one domain as in another; and they will be fully automatic, requiring no human assistance.

Finally, as with all automation, the real promise of automatic programming is not just in automating what is done now but in completely changing the way things are done. This will mean re-examining the traditional model of the software lifecycle. It will also mean breaking down the conventional distinctions between languages, environments, and interfaces.
4.3 Aspect Oriented Technologies

Aspect oriented technologies have the potential to make programmers' work easier, less time consuming and less error prone. Proponents say aspects could also lead to less expensive applications, shorter upgrade cycles, and software that is flexible and more customizable.

4.3.1 Separation of Concerns

"Separation of concerns" is at the core of Software Engineering. In its most general form, it refers to the ability to identify, encapsulate and manipulate those parts of software (concerns) that are relevant to a particular concept, goal, task or purpose. Concerns are the primary motivation for organizing and decomposing software into smaller, more manageable and comprehensible parts, each of which address one or more concerns.

An appropriate separation of concerns aims to reduce software complexity and improve comprehensibility within and across artifacts and throughout the software lifecycle. Significant advances have been made in achieving some of the benefits of separation of concerns, with the aid of technologies such as structured programming, object-oriented design etc. As software continues to become larger and more complex, the associated separation of concerns also grows in scale and complexity.

Over the last few years, a number of researchers have developed advanced modularization mechanisms that help to overcome a range of problems associated with inadequate separation of concerns. Some of the key works in this area include adaptive programming [33], adaptive plug-and-play components [41], aspect-oriented programming [12], composition filters [34], conceptual models, feature-based requirements engineering [35], generative programming [36], multi dimensional separation of concerns and hyperspaces [37], role-modeling, subject-oriented programming [38,39], variation oriented programming [40] etc. A multitude of possible approaches to separation of concerns exist and each has different cost/benefit tradeoffs, but all of them share in common the ability to define and manipulate orthogonal and non-orthogonal concerns.
Though a need for separation of concerns is quite evident in most areas of Software Engineering, in this thesis we target separation of concerns on code artifacts.

4.3.2 Tyranny of the dominant decomposition

Tyranny of the dominant decomposition is a key cause of the problems that motivated the need for advanced separation of concerns. It occurs mainly in object-oriented systems where a developer needs to manipulate two decompositions of the software, one by class and one by features common to all classes in the system. The developer is however limited to a single tyrant decomposition i.e. by class.

4.4 An Assessment of Aspect-Oriented Programming (AOP)

Evaluating emerging software development technologies is by no means easy. To thoroughly evaluate the usefulness of AOP is out of the scope of this thesis. In this section it is assessed whether AOP is really useful and whether it is usable in a commercial environment. This has been done by reviewing previous experiments.

Exploratory semi-controlled experiments have been carried out at the Department of Computer Science, University of British Columbia. The researchers used a particular aspect-oriented language created by researchers at Xerox Parc, called AspectJ™ [12]. The experiments carried out considered whether the separation of concerns provided by “AspectJ” enhanced a developer’s ability to find and fix faults in a multi-threaded system and also considered the ease of changing an existing distributed system.

These experiments concluded by highlighting the importance of the aspect-core interface in achieving development benefits with aspect-oriented programming. The aspect-core interface refers to the boundary between code expressed as an aspect and the functionally decomposed code [25]. The results of their experiments state that by paying careful attention to the design of the aspect-core interface, builders of aspect-oriented programming environments may be able to help a programmer focus more easily on code relating to a task.

The experiments also indicated that aspect-oriented programming might alter the programming strategies used by developers. Programmers are more likely to first try and solve a problem related to a concern captured as an aspect by focusing on the aspect code
when the aspect cleanly captures the concern. Clearly this requires programmers to adopt this new technology to steer away from traditional methods of programming. The participants in the experiment first looked for a solution that could be modularized in an aspect\(^5\). When the solution was appropriate AOP was beneficial. However when the solution could not be encapsulated within an aspect, the participants took longer to reach a solution.

### 4.4.1 Related works in Aspect-Orientation

“AspectC++” [42] is another instance of AOP. With the “AspectC++” project it is intended to extend the “AspectJ” approach to C/C++. It is a set of C++ language extensions to facilitate aspect-oriented programming with C/C++. However, the project is a research project and not intended to be used for serious software development.

“AspectC” is a simple aspect-oriented extension to C, intended to support operating systems and embedded systems programming. “AspectC” is not as yet available publicly. However, the computing science department at the university of British Columbia are using it for their “a-kernel” project, whose goal is to determine if aspect-oriented programming can be used to improve OS modularity, and thereby reduce complexity and fragility associated with system implementation [43].

“AspectS” [44] is an approach to general-purpose AOP in the “Squeak” environment. “Squeak” is an open, highly portable Smalltalk-80 implementation whose virtual machine is written entirely in “Smalltalk”. The intent of “AspectS” is to extend the Smalltalk environment to allow for experimental aspect-oriented system development. In its first implementation, “AspectS” is realized using plain “Smalltalk” only, without extending either the “Smalltalk” language or its virtual machine.

JAC [45] is a Java framework for Aspect Oriented Programming allowing the separation of concerns when programming (distributed) applications. The fault-tolerance or real-time aspects can be considered independently from what the application is doing. It is also a

---

\(^5\) Aspect - An aspect is a subprogram that is associated with a specific property of a program. As that property varies, the effect “ripples” through the entire program. The aspect subprogram is used as part of a new kind of compiler called an aspect weaver.
runtime execution environment providing visualization, administration, and configuration tools for Aspect-Oriented applications.

4.4.2 Alternatives to Aspect Oriented Programming

The limitations of traditional, but fundamentally basic, techniques are increasingly making themselves felt, most notably through our inability to construct and evolve programs at the pace demanded by modern times. The current abstraction and composition mechanisms far from suffice and AOP and related activities offer glimpses of how we might be able to articulate and encapsulate concepts in qualitatively new ways. Although there may be no alternative to such mechanisms, the choice at the moment is not whether or not to adopt something like AOP but how best to adopt it [28].

Some alternatives and technologies related to AOP are discussed below.

4.4.3 Reflection and Meta-object protocols

Aspect-oriented programming has a deep connection with work in computational reflection and meta-object protocols. A reflective system provides a base language and (one or more) meta-languages that provide control over the base language's semantics and implementation. The Meta languages provide views of the computation that no one base language component could ever see, such as the entire execution stack, or all calls to objects of a given class. In AOP terms meta-languages are lower-level languages whose join points are the hooks that the reflective system provides. AOP is a goal, for which reflection is a powerful tool.

4.4.4 Subject-oriented programming

Subject-oriented programming [38, 39] is a language-independent technology developed at the IBM Thomas J. Watson Research Center in Hawthorne, New York. It is a program-composition technology that supports building object-oriented systems as compositions of subjects. A subject is a collection of classes or class fragments whose hierarchy models its domain in its own, subjective way. A subject may be a complete application in itself, or it may be an incomplete fragment that must be composed with other subjects to produce a
complete application. Subject composition combines class hierarchies to produce new subjects that incorporate functionality from existing subjects. Subject-oriented programming thus supports building object-oriented systems as compositions of subjects, extending systems by composing them with new subjects, and integrating systems by composing them with one another (perhaps with "glue" or "adapter" subjects).

The flexibility of subject composition introduces novel opportunities for developing and modularizing object-oriented programs. Subject-oriented programming-in-the-large involves determining how to subdivide a system into subjects, and writing the composition rules needed to compose them correctly. It complements object-oriented programming, solving a number of problems that arise when object-oriented technology is used to develop large systems or suites of interoperating or integrated applications.

Researchers at IBM research have built support for subject-oriented programming in C++.

4.4.5 Hyperspaces

**Multi-dimensional separation of concerns** (MDSOC) [37] refers to flexible and incremental separation, modularization, and integration of software artifacts based on any number of concerns. It overcomes limitations of existing mechanisms by permitting clean separation of multiple, potentially overlapping and interacting concerns simultaneously, with support for on-demand re-modularization to encapsulate new concerns at any time. Realizations of MDSOC can permit incremental identification and encapsulation of concerns, without requiring the use of new languages or formalisms. MDSOC promotes reuse, improves comprehension, reduces the impact of change, eases maintenance and evolution, improves trace-ability, and opens the door to system refactoring\(^6\) and reengineering. MDSOC addresses some fundamental limitations in software engineering.

Hyperspaces\(^7\) [37] have been developed at IBM research to achieve MDSOC. Hyperspaces also provide a powerful composition mechanism that facilitates non-invasive integration, adaptation, and "plug-and-play." The approach has a low entry barrier, since it does not affect programming languages or processes--developers can continue to use their programming languages, development processes, development environments, and

---

\(^6\) Refactoring - the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure

\(^7\) Hyperspaces – an approach to achieving multi-dimensional separation of concerns. **multi-dimensional separation of concerns** refer to flexible and incremental separation, modularization, and integration of software artifacts based on any number of concerns
compilers of choice, while still reaping the benefits of multi-dimensional separation of concerns. Researchers at IBM have defined a tool, called HyperJ [47], which provides support for hyperspaces in Java.
4.5 Related works in eclipse

“Eclipse” is an open, universal tool platform or tool base. “Open” means that “eclipse” is an open-source project. Universal means that “eclipse” uses an innovative plug-in architecture allowing near-infinite extensions to the base IDE. These plug-ins can do anything from syntax highlighting to interfacing with a source code control system.

Unlike other open-source projects that don’t allow proprietary derivative works, “eclipse” can be extended with proprietary plug-ins, repackaged, and sold commercially. Hence, the “eclipse” user and developer community is growing quite rapidly. There is also a commercial IBM version of “eclipse” called “Web Sphere Studio Workbench”. This is the code base used by IBM for its Studio family of products as well as business partner tools vendors [14].

Following are some of the research and education projects that are underway in the “eclipse” community. They range from using “eclipse” as a teaching resource in the classroom to projects designed to explore the next generation of computing ideas and theories. What they have in common is that they are using “eclipse” in their work.

4.5.1 Integrating the BETA language with eclipse [1]

The center for pervasive computing at Aarhus University, Aarhus, Denmark is currently involved in an “eclipse” research project. The goal of their BETA “eclipse” project is to integrate their BETA programming environment Mjølner Tool into the “eclipse” platform. The Mjølner Tool is a BETA programming tool (written entirely in BETA) that supports: syntactic editing, graphical interface building, UML editing etc.

They plan to integrate their mjølner tool completely into “eclipse”. Initially they plan to create an editor for BETA programs.

4.5.2 Leveraging Cognitive Support and Modern Platforms for Adoption-Centric Reverse Engineering (ACRE) [2]

The computing science department at the University of Victoria, Canada is currently carrying out research on software reverse engineering tools.
The project states that research tools in software engineering often fail to be adopted and deployed in industry and so aims to develop tools that will be more likely to be adopted by users.

They also hypothesize that the interoperability of these tools can be improved significantly by leveraging recently developed middleware technologies. By exploiting such technologies, especially those with plug-in architectures, like “eclipse”, they plan to build prototypes of reverse engineering tools.

They feel that the experience gained would be beneficial for academic research and industrial practice.

4.5.3 Hipikat: Recommending useful software artifacts [3]

Researchers at the University of British Columbia, Canada are developing “Hipikat”, a tool intended to solve the problem involved with management of the various artifacts created during the software development process.

The artifacts in questions are source code, documentation, bug reports, e-mail, newsgroups articles, and version information. It is obvious that these artifacts are integral to the software development process and can be invaluable during the process of development and maintenance.

The project wishes to support access to this data and acknowledges that the requirement for this support becomes more important in the case of distributed development teams.

“Hipikat” is an ongoing research project and is being developed as an “eclipse” plug-in to support development of the “eclipse” platform.

4.5.4 ArchJava IDE [4]

“ArchJava” is an extension to Java developed at the University of Washington, U.S.A that seamlessly unifies software architecture with implementation, ensuring that the implementation conforms to architectural constraints.
Researchers are working on an “eclipse” plug-in for developing in “ArchJava”. The plug-in so far includes compilation support, an outline view, and (soon) a graphical architecture browser.

4.5.5 Guard: Relative Debugging [5]

Relative debugging has been defined as a technique that allows a user to compare the contents of data structures between two executing programs. It was devised to aid the testing and debugging of programs that are either modified in some way, or are ported to other computer platforms.

Traditional debuggers force the programmer to understand the expected state and internal operation of a program, relative debugging makes it possible to trace errors by comparing the contents of data structures between programs at run time. In this way, the programmer is less concerned with the actual state of the program, and more concerned with finding when and where differences between the old and new codes occur.

Researchers at Monash University have developed Guard, a Parallel Relative Debugger. Guard supports the execution of both sequential and parallel programs on a range of platforms, and exists for a number of different development environments. They are currently working on a prototype for eclipse that supports relative debugging for Java programs.

4.5.6 Knowledge Based Reverse Engineering [6]

Researchers at the University of Ottawa, Canada are developing infrastructure for browsing information about software architecture. This is being developed as an “eclipse” plug-in and interfaces to a database. Parsers generate the information in the database. They are also looking at adding visualization and metrics computation in the “eclipse” context.

4.5.7 Obasco (Object, Aspect, Components) [7]

Researchers at the Ecole des Mines de Nantes in France are developing an “eclipse” plug-in to integrate event-based AOP into “eclipse”.
The main assumption of their work is that crosscut definitions should be expressive enough to relate different events occurring during program execution and explicate state information belonging to those events. They acknowledge that this assumption is in contrast to other approaches to AOP, which restrict crosscuts to individual points during program execution, and the state associated to them to information local to those execution points. They also state that the model of EAOP is general enough to accommodate, in principle, any other model of AOP.

Work is currently in progress on the formalization and implementation of EAOP as well as some restricted setting to the framework.

4.5.8 AspectJ, Aspect Oriented Programming (AOP) for Java

“AspectJ” is one particular instance of a visual programming tool that supports development of software using AOP. It is distinguished by the fact that it was designed from the ground up to be compatible with Java.

“AspectJ” has been discussed in detail earlier in this section.
4.6 Summary

It is now evident that software engineering tools will continue to automate software development. As new programming paradigms emerge it is quite possible that the software development process itself may change completely.

There have probably been very few software engineering tools that have seen as much widespread usage as eclipse has. This has of course partially been due to that fact that “eclipse” has been able to capitalize on the Java programming language and the open source model. There have also been a large number of research grants made available by IBM to encourage its use among the research community.

“Eclipse” continues to grow in popularity as it transitions from a development tool to a development platform. The “eclipse” foundation now consists of the biggest names in the industry; these include IBM, BEA, Borland, Sun Microsystems and Sybase to name but a few. The members of the “eclipse” foundation are also strategic developer members and continue to invest in the “eclipse” platform.

As has been seen even though aspect-oriented programming promises to take away much of the complexity from traditional software engineering it will definitely be a while before it becomes a commercial reality, if it ever does so. There will have to be more widespread usage before it is accepted as an industry standard.
5 Component pascal and eclipse

“Component pascal” support has now been integrated into “eclipse” and “Visual Studio”. A plug-in has been developed for eclipse to allow the development of “component pascal” programs within “eclipse”.

Discussed in this chapter are the features of the component pascal plug-in for “eclipse” and strategies used for its integration into “eclipse”.

5.1 Component Pascal

The “component pascal” distribution consists of four programs, and a number of libraries. The programs are the compiler, \texttt{gpcp}; the \texttt{make} utility \texttt{CMake}; a module interface browser tool \texttt{Browse}; and a tool for producing interfaces from class files written in java, \texttt{J2CPS}.

The compiler produces either Microsoft.NET intermediate language or Java Virtual Machine (JVM) byte-codes as output. The compiler can be bootstrapped on either platform.

5.2 Eclipse

The “eclipse” platform is built on a mechanism for discovering, integrating and running modules called plug-ins [19]. Tool providers have to write their own plug-ins to enable eclipse to be aware of new content types and do things with them. Plug-ins are coded in Java, they declare their interconnections to other plug-ins and the platform via manifest files, which are basically configuration files containing Xml. Manifest files mainly define extension points, and extensions to other extension points defined in other plug-ins.
The platform runtime core implements the runtime engine that starts the platform base and dynamically discovers plug-ins. A plug-in is a structured component that describes itself to the system using a *manifest* (plugin.xml) file. The platform maintains a registry of installed plug-ins and the functions they provide.

Functionality is added to the system using a common extension model. Extension points are well-defined function points in the system that can be extended by plug-ins. When a plug-in contributes an implementation for an extension point, we say that it adds an extension to the platform. Plug-ins can define their own extension points, so that other plug-ins can integrate tightly with them.

The extension mechanisms are the only means of adding functionality to the platform and other plug-ins. All plug-ins use the same mechanisms. Plug-ins provided with the “eclipse” SDK do not use any private mechanisms in their implementation.

Extensions are typically written in Java using the platform APIs. However, some extension points accommodate extensions provided as platform executables, ActiveX components, or developed in scripting languages. In general, only a subset of the full platform function is available to non-Java extensions.

A general goal of the runtime is that the end user should not pay a memory or performance penalty for plug-ins that are installed, but not used. A plug-in can be installed and added to the registry, but the plug-in will not be activated unless a function provided by the plug-in has been requested according to the user's activity.
5.2.1 Standard Widget Toolkit

Apart from an innovative architecture the “eclipse” framework also makes use of the Standard Widget Toolkit (SWT), which is a huge improvement over previous Java-based widget toolkits such as AWT and Swing.

A common issue in widget toolkit design is the tension between portable toolkits and platform integration.

SWT addresses this problem by defining a common portable API that is provided on all supported platforms, and implementing the API on each platform using native widgets where possible. This allows the toolkit to immediately reflect any changes in the underlying OS GUI look and feel, while maintaining a consistent programming model on all platforms.

The "least common denominator" problem is solved by SWT in several ways.

- Features that are not available on all platforms but generally useful for the workbench and tooling plug-ins can be emulated on platforms that provide no native support. For example, the OSF/Motif 1.2 widget toolkit does not contain a tree widget. SWT provides an emulated tree widget on Motif 1.2 that is API compatible with the Windows native implementation.
- Features that are not available on all platforms but not widely used can be omitted from SWT. For example, Windows provides a widget that implements a calendar, but this is not provided in SWT.
- Features that are specific to a platform, such as ActiveX integration, are only provided on the relevant platform. Platform specific features are separated into separate packages that clearly denote the platform name in the package.

5.3 Integration into eclipse

Integrating “component pascal” into “eclipse” required a great deal of investigation into its architecture. It was initially quite clear that to develop in eclipse it was essential to be skilled in java and xml.
Development of the “component pascal” plug-in required investigation into the “eclipse” API and the xml schemas for the manifest files.

Development was started by initially reading the documentation provided with eclipse and the articles available online. This proved to be a good starting point and helped gain familiarity with the different components of “eclipse”. The source code for the Java Development Tool (JDT) provided with “eclipse” was also used to a great extent to see how the various parts were implemented; the plug-in for component pascal was modeled along those lines.

The general methodology for development was to first define the new component in the manifest file. Once the required classes for the same were developed, these were then packaged, deployed and tested.

The following components were built:

- Component Pascal resource creation.
- Component Pascal navigator.
- Component Pascal editor.
- Component Pascal perspective.
- Component Pascal compilation support.
- Top-level and popup menus.
- Error logging.
- Component Pascal outline.

---

8 Manifest files – configuration files in xml format containing structural information for an eclipse plug-in.
5.4 Plug-in Features

5.4.1 Creation of new component pascal resources

The central hub for all files related to a project is called a workspace. The platform workbench is a tool that allows the user to navigate and manipulate the workspace. "Eclipse" contains a system plug-in that provides APIs for creating, navigating, and manipulating resources in a workspace. The workbench uses these APIs to provide this functionality to the user. The component pascal plug-in makes extensive use of these APIs.

From the standpoint of a resource-based plug-in, there is exactly one workspace, and it is always open for business as long as the plug-in is running. The workspace gets opened automatically when the resources plug-in is activated and closed when the platform is shut down. If a plug-in requires the resources plug-in, then the resources plug-in will be started before the plug-in, and the workspace is available.

The workspace contains a collection of resources. There are three different types of resources: projects, folders, and files. A project is a collection of any number of files and folders. It is a container for organizing other resources that relate to a specific area. Files and folders are just like files and directories in the file system. A folder contains other folders or files. A file contains an arbitrary sequence of bytes. Its content is not interpreted by the platform.

A workspace’s resources are organized into a tree structure, with projects at the top, and folders and files underneath. A special resource, the workspace root resource, serves as the root of the resource tree. The workspace root is created internally when a workspace is created and exists as long as the workspace exists.

A workspace can have any number of projects.
New resources can be created through the new resource creation wizard as shown in figure 5.4.1.1.

![New Resource Creation Wizard](image)

Figure 5.4.1.1 New Resource Creation Wizard

Resources are represented in the “component pascal” plug-in as shown in figure 5.4.1.2.

![Resource representation](image)

Figure 5.4.1.2 Resource representation

ComponentPascalProject and ComponentPascalModule are both resources and inherit from ComponentPascalResource.
5.4.2 Creation of New component pascal Projects

Projects within "eclipse" are contained in the workspace. A project corresponds to a folder in the workspace folder (by default) with all its associated resources under the project folder. Each project is associated with and contains a project description file named ‘.project’ that contains xml defining the project attributes.

5.4.2.1 Project nature

Project natures allow a plug-in to tag a project as a specific kind of project. For example, the Java development tools (JDT) use a "Java nature" to add Java-specific behavior to projects. Project natures are defined by plug-ins, and are typically added or removed per-project when the user performs some action defined by the plug-in.

<!-- Project Natures -->
<extension point="org.eclipse.core.resources.natures"
id="ComponentPascalNature" name="Component Pascal Nature">
    <runtime>
        <run class="org.eclipse.componentpascal.resources.ComponentPascalNature"/>
    </runtime>
</extension>

Figure 5.4.2.1.1 Component Pascal Nature

A project can have more than one nature. However, when a project nature is defined, special constraints can be defined for it as well:

- one-of-nature - specifies that the nature is one of a named set. Natures in a set are mutually exclusive; that is, only one nature belonging to the set can exist for a project.
- requires-nature - specifies that the nature depends on another nature and can only be added to a project that already has the required nature.
All “component pascal” projects only have a single nature and are associated with a component pascal nature. The project description file for a component pascal project is displayed in Figure 5.4.2.1.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
  <name>g8</name>
  <comment>BASEMODULE:base.cp</comment>
  <projects>
    <buildSpec>
      <natures>
        <nature>org.eclipse.cpdt.ComponentPascalNature</nature>
      </natures>
    </buildSpec>
  </projects>
</projectDescription>
```

Figure 5.4.2.1.2 Xml markup containing project description

New projects are created using the component pascal project creation wizard.

Wizards are used to guide the user through a sequenced set of tasks. A plug-in can contribute wizards at predefined extension points in the workbench. It can also create and launch its own wizards.

On contributing to a workbench wizard extension point, the actions that launch the wizard are already set up by the workbench. Only the wizard that is to be used needs to be supplied.

A wizard is composed of an initial Dialog and subsequent Pages.

The component pascal wizard is an extension of the project creation wizard provided with eclipse. Each project has a set of pages, which may contain different configuration information for setting up the project. The component pascal project creation wizard has a single page associated with it that allows a user to enter a name for the project and allows the user to select a location for the project. The wizard checks that the project has a unique name and creates a corresponding folder in the file system. A project is created by selecting File→New→Project as shown in Figure 5.4.2.1.3.
The user is then presented with the project creation wizard as shown in Figure 5.4.2.1.4.

On clicking next, the project creation page is displayed as shown in Figure 5.4.2.1.5.
The above functionality is implemented via two classes:

- ComponentPascalProjectCreationWizard
- ComponentPascalProjectCreationPage.

ComponentPascalProjectCreationWizard extends org.eclipse.jface.wizard.Wizard.

As described earlier all functionality is inherited from the pre-defined classes.

### 5.4.3 Creation of new component pascal Modules/Base Modules

The “eclipse” plug-in contains wizards for the creation of component pascal modules and base modules. These are extensions of the file creation wizards that come with “eclipse”. The wizards can be accessed from the main menus or from the pop-up menus in the navigator.
Each component pascal project should contain only one base module. All component pascal projects identify a base module. If a new base module is created the user is presented with a warning. The base module name is contained in the project description file as can be seen in Figure 5.4.2.1.2.

A component pascal project may contain multiple modules; however they must have unique names. Modules are created through the module creation wizard as shown in Figure 5.4.3.1.

![Module Creation Wizard](image)

Figure 5.4.3.1 Module Creation Wizard

By default the project selected is the one currently selected in the project navigator. It is also possible to select another project from the navigator within the wizard. Once it is verified that the module name is unique, the module is created in the appropriate folder and can be opened on creation as selected by the user. On creation of a base module the project description is updated to reflect the new base module.
5.4.4 Editing of component pascal Modules

Plug-ins can define specialized file extensions and contribute editors that provide specialized editing features for these file types. An editor is a workbench part that allows a user to edit an object (often a file). Editors operate in a manner similar to file system-editing tools, except that they are tightly integrated into the platform workbench UI.

The “component pascal” plug-in defines the component pascal editor and associates it with the ‘.cp’ file extension. As a result any file resource with the extension ‘.cp’ opened in the eclipse workbench is displayed with the component pascal editor by default.

The component pascal editor is an extension of the default text editor that comes with eclipse. The default text editor provides much of the functionality required of editors in the eclipse workbench.

The component pascal editor is always associated with an input object. The input object is the component pascal module that is being edited. Changes made in an editor are not committed until the user saves them.

Only one component pascal editor can be active for any particular editor input in a workbench page. For example, if the user is editing hello.cp in the workbench, opening it again in the same perspective will activate the same editor. The component pascal editor type may be open many times within one workbench page for different modules.

The custom functionality added to the editor provides syntax highlighting as described in the next section.

5.4.5 Syntax Highlighting

Syntax highlighting is achieved in “eclipse” via a damager-repairer mechanism. Each time a change is made in the editor, we are effectively damaging the content, and
whenever any damage occurs the repairer comes into action, it re-scans the content and re-highlights the syntax.

Different language constructs are highlighted in different colours as can be seen in Figure 5.4.5.1.

Syntax highlighting is achieved via a set of classes as defined below

- **ComponentPascalCodeEnvironment** – This is the main class that co-ordinates syntax highlighting. The component pascal editor connects to this class and obtains the functionality required to perform syntax highlighting.
- ComponentPascalSourceViewerConfiguration – This class provides the damager-repairer mechanism via a Presentation Reconciler. The presentation reconciler implements the damager-repairer mechanism.

- ComponentPascalCodeScanner – Provides a scanner for “component pascal” code. The scanner extends the RuleBasedScanner provided with “eclipse”, which defines a set of rules and returns tokens depending on the rule applied. Different colours are associated with different tokens and thus syntax highlighting is achieved.

- ComponentPascalWhiteSpaceDetector – This class detects white space within the current content.

- ComponentPascalColorProvider – This class is basically a configuration class that defines tokens and the colours to associate them with.
5.4.6 Compilation of component pascal Modules

The initial aim was to be able to compile component pascal modules within “eclipse”. It was not intended to include the compiler within the eclipse plug-in; rather we intended to invoke the compiler externally.

Initially the compiler output all errors to stdout in simple text format. We however required all errors to be displayed within eclipse in a graphical format. The compiler was then modified to output errors in xml format; this output is obtained via a compiler switch (-xmlerror). The xml output is then parsed within “eclipse” making use of the java API for xml parsing (JAXP).

Displayed below is the Document Type Definition (DTD) for the xml output from the compiler.

```xml
<!DOCTYPE CompilerErrors [
<!Element CompilerErrors (Error)>  
<!ATTLIST errorsContained type CDATA #REQUIRED>  
<!Element Error ( Line, Position, Description)>  
<!Element Line (#PCDATA)>  
<!Element Position (#PCDATA)>  
<!Element Description (#PCDATA)>  
]>  

Through the “eclipse” plug-in it is possible to compile the currently selected module in the editor pane. This is done by clicking on Build→Compile File. If the file selected in the editor pane is not a component pascal module an error is returned. It is also possible to
compile the entire project by selecting Build→Compile Project. The project that the active module belongs to is compiled.

5.4.7 Error Logging

On being able to successfully compile modules within “eclipse”, it was then required to display the errors in a graphical format. Initially we planned to implement a Table Viewer widget to display all the errors, but later realised that it would be simpler to display the errors in the tasklist view provided with “eclipse”. We made use of the resource marker mechanism available within “eclipse”.

A marker is like a “yellow sticky note” stuck to a resource and all markers are displayed automatically as tasks within the tasklist. Within a marker it is also possible to define attributes such as location, severity of error etc. Clicking on a marker within the tasklist navigates directly to the file resource at the error location. This made the task of error-logging much easier. Now all errors are obtained from the xml output from the compiler, each error tag corresponds to an error. From each error obtained a marker is created and associated with the file resource being compiled. On each re-compilation all markers are removed and re-added.

The component pascal plug-in for “eclipse” displays errors in a neat and readable format in the tasklist as shown in Figure 5.4.7.1
Clicking on an error in the tasklist highlights the corresponding line in the editor where the error is encountered. This way it is easier to correct all errors in a module. On successful compilation all errors are removed from the tasklist, and the module is saved successfully.

5.4.8 Resource Navigation

The “eclipse” workbench has a default resource navigator. Initially the component pascal navigator was an extension of the default navigator. However there was a need for greater customization, which was not possible with the default navigator. The component pascal navigator is now an eclipse view containing a SWT tree widget that is populated by the contents from the workspace. Only projects with the component pascal nature are displayed. The navigator integrates seamlessly with the editor pane and the active selection in the navigator is displayed in the editor pane and vice versa.

The component pascal navigator (figure 5.4.8.1) displays all component pascal projects in the workspace as well as all modules and files contained within. It is also possible to filter its contents to display component pascal modules only. Double-clicking on any
module in the navigator opens the module for editing in the editor pane. If the module was open previously it appears selected in the editor pane. Selecting a module in the editor pane also selects the corresponding entry in the navigator.

The navigator also has a context-sensitive pop-up menu associated with it.

![Resource Navigator](image)

Figure 5.4.8.1 Resource Navigator

### 5.4.9 Layout Customization

Perspectives provide an additional layer of organization inside a workbench window. Users can switch between perspectives as they move across tasks. A perspective defines a collection of views, a layout for the views, and the visible action sets that should be used when the user first opens the perspective.

The component pascal perspective is a simple extension of a perspective factory that comes with "eclipse". The perspective also defines a layout for the elements that it displays.
The component pascal perspective can be opened by clicking Window→Open Perspective→component pascal. The perspective displays the various elements of the plug-in as shown in Figure 5.4.9.1.

It is possible to customize the perspective by dragging the different elements and placing them in different positions according to one’s preference. When eclipse is opened the next time the perspective appears in the same state. However, if the perspective is closed and re-opened it appears in its default state as shown in Figure 5.4.9.1.
5.4.10  Project Builds

The project build is performed by calling the ‘CPMake’ command on the base module in the component pascal project (CPMake is the make utility that comes with the component pascal compiler; it is used to determine a valid order of compilation for component pascal projects). The project must contain a base module or else an error is returned.

To perform a project build the xml output from the compiler had to be changed. On compiling a single module the compiler returned a single ‘compilererror’ tag containing error tags for each error. The resource the errors were associated with was, of course, the current resource being compiled. This however, proved to be a shortcoming while compiling an entire project, as the resource being compiled was unknown. The errors were still displayed in the tasklist however as they were associated with the project they were not navigable to the module in the editor pane.

As a result a new attribute had to be added to the xml and the compilererrors tag now contains an attribute that reflects the resource the errors are associated with. Hence on compiling a project multiple compilererror tags will be returned each containing error tags for the resource they refer to.

A project build is performed by selecting a project in the Navigator and selecting ‘Compile Project’ from the Build menu as shown in Figure 4.2.10.1. It is also possible to compile a project by selecting one of its modules in the editor pane. Any compilation errors are displayed in the tasklist as shown previously in Figure 5.4.7.1.

![Figure 5.4.10.1 Project Compilation](image)
5.4.11 Project Execution

A project can be run by selecting a project in the navigator and selecting ‘Run Project’ from the Build menu. It is also possible to run a project by selecting one of its modules in the editor pane. The project must contain a base module or else an error is returned.

All output is displayed in the component pascal console which is displayed on running a project.
5.4.12 Content Outline

The content outline displays a structured view of all program members and seamlessly integrates with the component pascal editor allowing navigation between the outline and the source code.

There already existed a scanner and parser for component pascal written in component pascal; however it was not possible to integrate the results from these with the existing “eclipse” Java API and the component pascal Plug-in. Hence, developing the outline required re-writing a scanner and parser for component pascal in Java.

Due to a lack of experience in compiler theory it was initially preferred to use a tool to build a parser for the language. This was attempted using a book “Building Parsers with Java” by Steven John Metsker.
The scanner as expected was defined first. The methodology and code available in the book makes it possible to define a scanner. The scanner at any given time is in a particular state, dependent on what the current token is and what tokens it has just encountered or is going to encounter. Based on its current state it can move to different states, all of which contribute to putting together tokens from the source code.

These “states” are basically Java classes that are used to identify domain objects that define the tokens that it will encounter e.g. if the scanner encounters the tokens “(* “ this puts the scanner into a commentState. In the comment state the scanner verifies whether the tokens that it encounters form a valid comment in the language. Once the end of a string of valid tokens that form a comment is reached the scanner returns to the normal state.

To use the pre-built scanner it was necessary to identify these “states” and create classes for the same that the scanner would use to form tokens from a component pascal program.

Once the scanner was perfected it was then time to modify the parser to be able to parse component pascal code. However before this was done it was necessary to plan what would be displayed in the content outline, as this is what the parser would be creating. It was decided that the content outline would display all import declarations, constants, variables, types and procedures. Domain classes were created for each and these domain objects were created and stored by the parser as it parsed the program. Once parsing was complete these objects were passed to the Content outline, which displayed them.

5.4.12.1 Domain specification for the outline

The following classes were used to specify the domain objects for component pascal. These were all contained in the package org.eclipse.cpdt.lang.members

- ComponentPascalRoot – Container for all other language domain objects
- ComponentPascalModuleDeclaration – the top-level container for a component pascal module.
- ComponentPascalConstants – List of all constants
- ComponentPascalTypes – List of all types
- ComponentPascalVariables – List of all variables
- ComponentPascalProcedures – List of all procedures
- ComponentPascalMember
- ComponentPascalConstant
- ComponentPascalType
- ComponentPascalVariable
- ComponentPascalProcedure

ComponentPascalRoot contains collections of types, variables, import declarations, procedures and constants. These collections are populated as the parser finds these objects in the component pascal source.

5.4.12.2 Parsing the component pascal source

The parser as defined by the book used a mechanism it termed as assemblers. An assembler (Java class) had to be created for each sequence of tokens that the parser would encounter. Hence, some of the assemblers would be ComponentPascalConstantAssembler and ComponentPascalProcedureAssembler.

The assembler contained the rules that dictated what constituted a valid declaration within the language. With the initial success in defining the scanner it was decided go down this path and assemblers were defined for “component pascal” using the EBNF grammar for the language. The mechanism also included pre-defined assemblers like “Sequence” which made it possible to define a declaration constituting multiple assemblers, “Alternation” which made it possible to define assemblers where the sequence of the declaration may not be known initially. These are to name but a few.

As the skeleton of the parser was defined using this method it seemed it would be quite successful and relatively quicker then writing a parser from scratch. However as soon as
more complicated “assemblies” were attempted it was obvious that the parser was not able to handle these. In particular multiple combinations of assemblers were required for a complex language like “component pascal” and it became apparent that the parser would require some major modification to it. It was then decided that it would be better to write a parser from scratch using the pre-defined scanner.

It was decided to use a similar methodology as last time, but this time instead of using the earlier framework and defining “assembler” methods, methods were defined in Java. A specific method was created for the various declarations in component pascal. The parser contained instances of the domain objects. It is important to note that this was not meant to be a fully functional parser. Its main objective was to create domain objects for the content outline. Hence, it was considered safe to ignore a lot of source code that was not required to be displayed in the outline e.g. in a procedure all that was required was the name of the procedure and the line it started on, all the code in the middle was not required and could be ignored till the end of the procedure was reached. Even though the code could be ignored it still had to be parsed correctly so that the parser would not fall over and be aware of its current state.

Writing the parser was quite a rewarding experience. Parsing is quite fundamental to software development as we write parsers almost everyday without knowing that they are parsers. The theory and know-how behind doing this properly will certainly be beneficial in the future.

5.4.12.3 Inner workings of the content outline

The “eclipse” API has classes pre-defined for creating content outlines. These are meant to integrate with an existing editor via a set of pre-defined events. The content outline also contains a TreeViewer that enables it to display program members in a structured format. This TreeViewer is linked to the save event in the editor so that it can update itself each time a change is saved to the source code.

Looking further into the content outline the display mechanism works using a content provider and label provider mechanism. The content provider is used to provide the
actual content for the Treeviewer within the content outline. The label provider dictates how each element is meant to be displayed.

Once the parser completes parsing the component pascal source, it passes the ComponentPascalRoot to the content provider. The ComponentPascalRoot object is converted to an array of Objects and the toString method of each object is used to display the text in the tree. The label provider formats the display further by adding the images.

5.4.12.4 Integrating the content outline and editor

The content outline integrates with the component pascal editor. Clicking on any of the elements in the tree highlights the corresponding line in the source code. This effect was quite simple to achieve. Each domain object that was represented in the tree contained the line number that the parser encountered it on. On clicking the element in the tree the line number was retrieved and using a reference to the editor the corresponding line was highlighted.
5.5 Summary

The “eclipse” plug-in for “component pascal” has made extensive use of the “eclipse” application programming interface (API) and provides a fully functional development environment.

Building the plug-in was particularly rewarding as it provided an understanding of how large software frameworks are structured. Of particular interest were large frameworks based on Java and XML, which are the core technologies for most business systems today.

Prior to developing the plug-in experience in building compiler components and software development tools was limited. Building the eclipse plug-in has provided an appreciation of the finer details of compiler development and systems programming. Though these skills may not be used on a daily basis, it has provided a deeper understanding of software engineering, which will definitely help in all aspects of future work.
6 Aspect Oriented Extensions

On completion of the eclipse plug-in the second objective of this research was to incorporate aspect-orientation into component pascal. Aspect-orientation is a new programming paradigm that extends object-orientation and provides a better separation of concerns for object-oriented programs.

Discussed in this chapter are:
- aspect-orientation in general;
- insights into its incorporation into component pascal and the
- details of modifications made to the compiler to support aspect-orientation.

6.1 Initial Investigation

It was initially required to understand what aspect-orientation really was. Once an initial understanding of aspect-orientation was completed it would have to be determined how this paradigm would have to be implemented to further my understanding of the same. There is a lot of documentation around on the subject, however initially it was hard to see the gains to be made through aspect-orientation. What was its actual benefit? It was hard to implement it without being convinced of its actual benefits.

After reading a myriad of documentation on the subject it was quite clear that the central idea was the separation of crosscutting concerns. The level of abstraction provided by this separation set apart various aspect-oriented implementations. However, simply providing a separation and integration mechanism did not seem to be sufficient. Programmers have used the principle of separation of concerns for a long time. Looking at object-oriented programming, the act of creating a class providing specific functionality and then including calls to it in various places in the related code is a separation of a concern. We do not have to re-define the functionality at every place we need to use it; in fact this mechanism is also available in procedural programming where existing bodies of code can be “included” into a source where it needs to be used. What then made aspect-oriented programming special?
The first thing that became apparent was that in all previous cases the code had to be aware that a separated concern was required at some stage and an explicit call to the same had to be coded into the source. With aspect-orientation this was not the case. The code where the separated concern was being used did not need to be aware of the same; in other words the programmer did not need to explicitly specify this. The level of abstraction in the specification of the separated concerns would indicate the power in the mechanism and the benefits to be gained. The aspect weaver would scan existing code and when it would find a declaration that needed to have code included (as defined by an aspect) it would be included as a pre-processing step. This immediately made it clear that this was quite a powerful mechanism; however one that required a precise definition of an advice within an aspect that would dictate where the code was to be included. It was also necessary to determine how this code was to be included, and whether it would run before, after or instead of the existing piece of code.

An aspect-weaver would have to written. This would be a complex piece of software that would have to parse existing code and then determine whether an advice had to be applied. The aspect weaver has been defined as similar to the Observer process [21] where the aspect code monitors the execution of the base program and when certain sequences of events occur, additional code that belongs to the aspect is run. This indicates that the weaving of aspect code is a dynamic process, and occurs at runtime. We have however been describing a process where aspects are included at compile time. The runtime weaving of aspects will be expensive and harder to implement into the existing component pascal compiler.

Having written a parser for the language, I initially thought that the aspect weaver would be an extension of the parser and would carry out static weaving of aspect code. It would parse the relevant component pascal modules and advices, build abstract syntax trees for the two and then determine what code inclusions needed to be made.

Once the aspect weaver had applied the necessary code inclusions this body of code could be supplied to the component pascal compiler and they could be compiled as

---

9 Advice – A construct contained in an aspect module, contains the code that is applied to the base code
normal component pascal code. Hence aspect-orientation would exist as a pre-processing step.

Once a broad understanding of the paradigm was made I needed to have a clearer picture of how it was all going to fit together. The terminology used in aspect-oriented programming is easily misunderstood and so it was initially important to know what components were to be created and how they would be addressed. Following is a discussion of the key components:

6.1.1.1 Base program

The base program is any existing component pascal module which aspect code needs to be applied to.

6.1.1.2 Aspects and Advice

An advice is the construct that contains the code that is applied to the base code. Advice constructs will be contained in a single aspect module. The aspect module will have a structure similar to a normal module.

6.1.1.3 Join Points and Point-cut Designators

Join Points and Point-cut Designators together define where the advice code is to be applied. The join point is the point in the Base Program where the advice code is applied whereas a point-cut designator specifies these join points. A single point-cut designator may designate multiple join points in the code.

It was quite clear that the hardest part would be writing the aspect weaver. This would have to be a complete overhaul of the parser. The parser as mentioned earlier was not complete as it was tailored for the content outline and so was able to ignore a lot of code. The aspect weaver would not have this luxury, so in addition to creating a complete parser an additional comparison mechanism would have to be written which would include advice code into the existing source.
Another point of ambiguity here was the dependence between the base program and the advice code. The advice code would normally be executed based on the existence of a particular sequence in the base code. In such a situation either the advice code would execute completely independently or be dependent on the base code for parametric data. If the advice code were dependent on the base program then a mechanism would be required to make data from the base program available to the advice code.

A classic example of a benefit of aspect-orientation is of code used to write to logs in a web-server. It is possible to define via an advice a sequence in the existing code where it was necessary to write to the log. This would for example reduce the numerous lines of code that write a line of text to the log. However, how would the advice access what it needs to write to the log? This meant that there would have to be some dependence, in other words the advice code needed to access variables in the existing code.

In the initial assessment of aspect-oriented programming it was thought that one of its benefits was no such dependence. While such dependence will be required, the level of abstraction of the advice code and the underlying mechanism that implements the aspect-orientation would have to ensure that this dependence is not visible in the base program, and that the developer writing the base program would not need to be concerned with this. How this dependence would be abstracted was yet to be determined.
6.2 Initial Plan

It had initially been decided to implement aspect-oriented extensions to component pascal through the eclipse plug-in. This would have required no changes to the component pascal compiler and all aspect-oriented extensions would be added through the plug-in. The aspect-orientation would be taken care of by the plug-in and then passed as normal source code to the underlying compiler, which would compile it to executable Java byte code.

![Diagram of the framework for aspect compilations](Image)

Figure 6.2.1 Framework for aspect compilations

It was initially decided to pursue this direction, as there were stronger Java skills available as compared to component pascal skills. Moreover, there was no previous
experience developing with component pascal, let alone modifying a large and complex 
compiler, which had been perfected through years of work.

Moreover modifying the plug-in to include aspect-orientation as described earlier 
required writing a complex aspect weaver and was not an easy task.

After much discussion further insights into the workings of the compiler were gained and 
it was decided that it would be more beneficial to modify the compiler as it might be 
possible to add small modifications to the code that would enable us to do what was 
required. Using the initial method where aspect-orientation was not going to be 
incorporated into the compiler, the base programs would need to be parsed a second 
time and textual insertions of code would need to be made at points where advice code 
need to be applied.

This was obviously not a very efficient method and quite expensive in terms of 
processing time as well. Integrating aspect-orientation directly into the compiler would be 
more efficient as direct access to internal data structures would be possible and minimal 
modification to existing code would be required.

The presence of eclipse for the compilation of aspect-oriented component pascal 
programs would not be required. However eclipse would enhance the visual experience 
by supporting aspect-oriented extensions to component pascal via the following:

- Editor for component pascal aspect modules.
- Content outline for aspect modules.
- Identification of aspect module in the component pascal project nature.

The following were also required initially

- Investigation into the architecture of the component pascal compiler.
- Definition of syntax for aspect modules.
6.3 Aspect Syntax

It was proposed to store all aspect code in a single aspect module initially. The proposed syntax for an aspect module is displayed in the following sample aspect module. The following sample aspect module is intended to keep a count of all procedures called and all procedures completed successfully.

ASPECT MODULE Sample;

VAR bCount,sCount:INTEGER;

ADVICE pBCount;
BEFORE pCall();
BEGIN
INC(bCount);
END pBCount;

ADVICE pACount;
AFTER pCall();
BEGIN
INC(sCount);
END pACount;

END Sample;

Variables bCount and sCount are global to the aspect module.

As can be seen the aspect syntax is very similar to normal component pascal code and reads like a normal component pascal module. The only difference is the existence of advice declarations. The advice declarations, though simpler are very similar to procedure declarations, the only difference being the ADVICE keyword and the BEFORE of AFTER directive indicating how the advice is to be used.
6.3.1 Features of the aspect module

- It is possible to declare constants, variables and types before declaring any advices.

- An advice may be called before and after a procedure call. It is not intended to implement around calls (*An around call in an advice replaces the code it is intended for*). It is also only intended to apply advice code to procedures; this level of granularity may be increased later as seen fit.

- Code within the body of an advice is much the same as code within a procedure; however variables may not be passed into an advice. Variables may be declared locally.

- A “before or after” construct in an advice constitutes a point cut and may refer to one or more join points. Initially it is intended to only refer to a single join point in a point-cut.

- A code insertion mechanism is required to insert code at the appropriate join point in the source component pascal program.

6.3.2 Point-cut Definition

As can be seen in the example code above a point-cut is defined in the second line of the declaration of an advice.

```
ADVICE pACount;
AFTER pCall();
BEGIN
INC(sCount);
END pACount;
```

In the above advice the second line states that the advice code is to be called after the procedure pCall has executed. Hence the point where the advice code cuts in is after the procedure pCall has completed execution.
It was also discussed that it might be possible to define point-cuts visually within the component pascal plug-in. In “eclipse” it is possible to define a breakpoint in the code by clicking on the marker bar in the editor and clicking on “Add Breakpoint” (in figure 6.1.4.1 this has already been done). This helps when debugging code.

As seen above it would be possible to add an option to add a point-cut at a specific point visually. It would however then be required to get input from the user detailing what advice they would like code included from. Based on the position of the point-cut it would be possible to determine whether the advice code was to be run before, after or around the existing body of code. This information would have to be stored temporarily and made available to the aspect weaver. Hence, using this approach a point-cut would not be persistent, but only available for the duration of the session with “eclipse’. It would however be much easier to add point-cuts. Multiple point-cuts would have to be added manually, however if the point-cuts were defined textually it would be possible to define more generic point-cuts.

It would be ideal to have a combination of textual and visual point--cuts that could complement each other. In the initial implementation only textual point-cuts have been implemented.
6.4 Current Implementation

After some investigation it was found that the parser for “component pascal” could be modified to parse aspect modules as well. An aspect module can be parsed exactly like a normal module, as it is the same in structure. The only difference is that in addition to normal component pascal elements it also contains advices. Hence if the parser were able to parse advices it would be able to parse an aspect module.

6.4.1 Changes to compiler

The high-level process for parsing a component pascal procedure is as shown below

![Diagram of procedure parsing]

Figure 6.4.1.1 procedure parsing

DeclarationSequence is the top-level procedure, which parses all declaration in a module. As it encounters procedure declarations it first parses the procedure heading (signature) and then the procedure body.

The data structure, which stores the abstract syntax tree (AST) for a component pascal module, is the Blkld (Figure 6.4.1.2). The Blkld stores a collection of procedures. Procs is the data structure, which stores all data, related to a single procedure. As a component pascal module is parsed Procs are added to the collection of procedures and so the abstract syntax tree is built.
Figure 6.4.1.2 BlkId

Figure 6.4.1.3 shows the procedure for parsing an advice.

```
TYPE
  BlkId* = POINTER TO EXTENSIBLE RECORD (D.Scope)
  (* ... inherited from Idnt ... *)
  (* kind-* INTEGER; (* tag for unions *)
  (* token* : Scanner.Token; (* scanner token *)
  (* type* : D.Type; (* typ-desc | NIL *)
  (* hash* : INTEGER; (* hash bucket no *)
  (* vMod- : INTEGER; (* visibility tag *)
  (* dfScp* : D.Scope; (* defining scope *)
  (* tgNtn* : ANYPTR; (* ... inherited from Scope ... *)
  (* symTb* : SymbolTable; (* symbol scope *)
  (* endDecl* : BOOLEAN; (* can't add more *)
  (* ovfChk* : BOOLEAN; (* check overflow *)
  (* locals* : IdSeq; (* varId sequence *)
  (* scopeNm* : L.CharOpen (* external name *)
  (* ]---------------------------]*)
  mcDBody* : D.Stmt; (* mod init-stmts *)
  mcDClose* : D.Stmt; (* mod finaliz'n *)
  impOrd* : INTEGER; (* implement ord. *)
  mcDKey* : INTEGER; (* module magicNm *)
  main* : BOOLEAN; (* module is main *)
  proc* : ProSeq; (* local proclist *)
  expRecs* : D.TypeSeq; (* exported recs. *)
  xAttr* : SET; (* external types *)
  xName* : L.CharOpen; (* ext module nam *)
  pkgNam* : L.CharOpen; (* package name *)
  clsNam* : L.CharOpen; (* dummy class nm *)
  verNm* : POINTER TO ARRAY 6 OF INTEGER;
  isAsp* : BOOLEAN;
END; (* --------------------------- *)
```

```
Figure 6.4.1.3 Advice Parsing
```

```
```
On comparing procedure parsing and advice parsing it is noticed that there is not much difference in the two. This is due to the fact that an advice and procedure are very similar in structure. A procedure has a more complex signature; the signature of an advice is less complex declaring only the keyword ‘ADVICE’ and the advice name followed by a semicolon. ProcedureBody is a common procedure used to parse the body of the procedure and advice. AdviceStuff and ProcedureStuff have a similar structure, they differ in that they call advice heading and procedure heading respectively to parse the name and signature of an advice and procedure respectively.

In actual fact there is not much difference in the declaration of a procedure and an advice. As can be seen in figure 6.4.1.4. Procs is the main data structure that is used to store procedure data. A PrcId is a pointer to Procs. An Advclid is a pointer to a PrcId with three extra elements:

- advcType, an integer value that indicates the type of the advice construct (before – causes execution of the advice code before the procedure, after – causes execution of the advice code after the procedure)
- procId, which is the AST for the current advice
- callPrc which stores the name of the procedure where the advice code needs to be applied.

As all the existing code in the component pascal compiler is coded to work with PrcIds and the fact that an advice extends a PrcId, all this code immediately became portable and hence was able to be re-used.

Before we describe how the aspect weaving was done it is necessary to describe the framework that supports this.

When compiling a module using the command line it is possible to provide various options for compilation. To compile an aspect module it is necessary to use the new aspect option. The syntax for the same is as follows

cprun gpcp –aspect <aspect module name> <cp module name>
TYPE
  Procs* = POINTER TO ABSTRACT RECORD (D.Scope)
    (* ---- inherited from Idnt ---- ---- *
    * kind* : INTEGER; (* tag for unions *)
    * token* : Scanner.Token; (* scanner token *)
    * type* : D.Type; (* typ-desc | NIL *)
    * hash* : INTEGER; (* hash bucket no *)
    * vMod* : INTEGER; (* visibility tag *)
    * dScp* : Scope; (* defining scope *)
    * tGxtn* : ANYPTR;
    (* ---- inherited from Scope ---- ---- *)
    * symTb* : SymbolTable; (* symbol scope *)
    * endDecl* : BOOLEAN; (* can't add more *)
    * ovfChk* : BOOLEAN; (* check overflow *)
    * locals* : IdSeq; (* varId sequence *)
    * scopeNm* : L.CharOpen; (* external name *)
    (* ------------------------------------ *)
    proCNm* : L.CharOpen; (* external name *)
    body* : D.Stmt; (* procedure-code *)
    except* : LocId; (* except-object *)
    resolve* : D.Stmt; (* except-handler *)
    resolve* : Procs; (* fvd resolution *)
    rtsFram* : INTEGER; (* RTS local size *)
    nestFas* : PrcSeq; (* local proclist *)
    pAttr* : SET; (* procAttributes *)
    lxDepth* : INTEGER; (* lexical depth *)
    bndType* : D.Type; (* bound RecTp *)
    xhrType* : D.Type; (* XHR rec. type *)
    basCl* : BaseCall; (* for stors only *)
END;

PrSSeq* = RECORD
  tide* : INTEGER;
  a* : POINTER TO ARRAY OF Procs;
END;

PrId* = POINTER TO EXTENSIBLE RECORD (Procs)
  clsNm* : L.CharOpen; (* external name *)
  stdOrd* : INTEGER;
END;

MthId* = POINTER TO RECORD (Procs)
  mthAtt* : SET; (* mth attributes *)
  rcvrFnm* : ParId; (* receiver fnmal *)
END;

AdvCId* = POINTER TO EXTENSIBLE RECORD (PrId)
  advcType* : INTEGER;
  procId* : PrId; (* desc of proc for before/after call *)
  callPrc* : INTEGER; (* hash bucket for the called procedure *)
END;

Figure 6.4.1.4 Declarations
When compiling a normal component pascal module the following steps are taken (this is depicted visually in figure 6.4.1.5)

1. Initialisation
2. Parsing
4. Type Erasure
5. Dataflow Attribution
6. Symbol File creation
7. Code Generation

On encountering the aspect option, the compiler proceeds to get the next command line parameter, which is the filename that stores the aspect module. The aspect module is then parsed by the same code that previously parsed a normal module. Certain code changes were included to save the aspect state. Once the aspect module is compiled it is stored into the current instance of the AST.

Once compilation is complete this current instance is saved in another instance of the AST, which is meant to store the aspect state. Once the aspect module is compiled the state of the compiler is re-initialised. The re-initialisation also initialise the current instance of the AST in preparation for storage of the target module (the module to which the aspect code will be applied). When compilation of the target module is underway it is checked if the state of an aspect module was saved previously and if it is found it is not re-initialised. This ensures that the aspect module state is not lost when the base module is being compiled and re-initialisation of the compiler state takes place.
Visual, aspect-oriented programming tools for Component Pascal in Eclipse

Figure 6.4.1.5 Compilation process

1. Initialisation
2. Parsing
3. Statement Attribution
4. Type Erasure
5. Dataflow Attribution
6. Is Aspect module
   - Yes: Create Symbol File
   - No: Does aspect code need to be included?
     - No: Code Generation
     - Yes: Include Aspect Code
6.4.2 Aspect Weaving

As described earlier the AST for a module stores the state for the aspect module. When a base module is compiled it looks for the previous occurrence of an aspect module, if an occurrence is located the aspect weaving process starts.

The aspect weaving process as envisioned earlier was complex, as it required parsing of source code, creation of ASTs and subsequent manipulation of the same to weave in the aspects. By including aspect-orientation into the compiler directly and storing the state of the aspect module it was not required to re-parse the source code. We were able to manipulate the AST directly and this process was quicker and more efficient.

The data structure used to store an advice is an extension of the data structure used to store a procedure. When the advices are stored into the AST the sequence of advices are stored as a sequence of procedures.

It was initially decided to implement BEFORE and AFTER advices. One of the methods of implementing this would be to insert a call to the advice as the first statement of the procedure it was to run before or the last statement of the procedure it was to run after. This would involve modifying the data structure for a procedure and storing a call to the advice in it. It would then have to be ensured that a call to the advice was generated when the byte code was being generated.

However, later a different method was chosen. The component pascal compiler defines data structures for storing calls to procedures. These data structures store information on the procedure being called and the parameters being supplied to it. When code is generated for these calls, as is normal practice the parameters are pushed on the stack and then the procedure is called. We chose to store the advices in these data structures. Hence, the data structure that stored information on a procedure call would now also store information on all advices that needed to be called before or after the procedure being called.
When compilation of a base module is underway a check is made for the previously stored state of an aspect module. As compilation continues each time a call to a procedure is encountered, it is first checked if the current call is relevant and whether application of an advice is possible. This is due to the fact that there would be various calls to procedures some of which would not warrant the use of advices.

When details of a procedure or and advice are stored into its relevant data structure its name is stored using a hash bucket. These hash buckets are used to compare names and so no complex string manipulation is required. When a call to a procedure, which may require the use of advices, is being stored, a comparison is done against all advices stored in the compiler state. All advices that match are returned as a collection.

Figure 6.4.2.1 shows the procedure GetAdviceDetails used to retrieve the advice collection.

```pascal
PROCEDURE GetAdviceDetails(var seq: PrcSeq; hashCx: INTEGER): ProcS;
VAR i: INTEGER;
advoc: AdvId;
BEGIN
  FOR i := 0 TO seq.tide-1 DO
    advoc := seq.a[i] (AdvId);
    IF advoc.procId.hash = hashCx THEN
      RETURN seq.a[i];
    END;
  END;
  RETURN NIL;
END GetAdviceDetails;
```

Figure 6.4.2.1 Retrieval of advices

Figure 6.4.2.2 shows the data structures used to store calls to procedure calls. A call to a procedure is normally stored as a CallX. The CallX data structure has been modified to contain two procedure sequences. These are intended to store advices, which will be called before or after a procedure call.
As a last step when all data structures are created these are now converted to byte-codes and stored in Java class files. Once again at this code-generation stage when the data structure for a `CallX` is being converted, before the arguments are pushed on the stack, all the advices to be run before are checked for and if any are located these are then converted. Once the code has been generated then lastly the occurrence of any advices to be run after the procedure call are located and if found code is then generated for them.
The above process completed the aspect weaving.

The final code changes we had to make to implement the aspect weaver were a fraction of what would have to be made if we had gone down our earlier path. However reaching this point took a great deal of investigation into the structure of the compiler.

On reaching a final solution it is found that there is no separate aspect weaver. The aspect weaving functionality has not been modularized, rather has been tightly integrated with the original compiler. This provided us with more compact code and a simpler, efficient and more elegant solution to a problem that was thought initially to be a more complex process.

6.4.2.1 Dependence between base code and aspect code

The dependence between the base code and aspect code was discussed earlier. In our implementation of aspect-orientation, the only dependence is the name of the procedure affected by the advice. The base module can therefore be developed independently of the aspect module and no provisions need to be made for the inclusion of the aspect code in the base module. Of course the developer of the aspect module needs to have knowledge of the base program for successful implementation of the aspects. This knowledge of the base program is the dependence.

While such dependence will be required this can be eliminated as well. Discussed below is an implementation for further generalization of aspects to reduce the dependence even further.

For example, as shown in the following code, procedure Parameters expects two parameters, an integer and a Boolean in that order. Advice parameterAware declares that it will be applied to procedures named Parameter that expect and integer and Boolean parameter in that order.
PROCEDURE Parameters (INTEGER a, BOOLEAN b)
.
.
.
.
END Parameters.

ADVICE parameterAware;
BEFORE Parameters(INTEGER, BOOLEAN);
BEGIN
.
.
.
.
END parameterAware;

There is still dependence as knowledge of the name of the procedure the advice is being applied to is required. Note that in the above code the advice accepts parameters as well. A possible implementation of this is discussed later.

To remove the above dependence and in the process make the advice more generic, the advice may be declared as below:

ADVICE parameterAware2;
BEFORE (INTEGER, BOOLEAN);
BEGIN
.
.
.
.
.
END parameterAware2;

As shown above, advice parameterAware2 is generic and is to be applied to any procedure expecting two parameters in the same order. This can be further generalized by including return types as well as shown
ADVICE parameterAware3;
BEFORE (INTEGER, BOOLEAN): BOOLEAN;
BEGIN
  .
  .
  .
  .
END parameterAware3;

Another possibility is to only have the return type

ADVICE parameterAware4;
BEFORE (): BOOLEAN;
BEGIN
  .
  .
  .
  .
END parameterAware4;

Using these combinations will provide the programmer more flexibility and allow for a more robust solution.

6.4.2.2 Implementing parameter awareness in advices

As described previously, including the advice code in the data structures for calls to procedures required a simple comparison of the hash buckets for the called procedure and the stored advice. Implementing advices that accept parameters and return types will require more comparisons, the parameters for the call will have to be retrieved, and their types checked. If a successful comparison occurs then advice code is included. Checking for return types is also possible as the return type is stored in the data structure for the call to the procedure. In such a situation a comparison will have to be
made with the stored return type and the return type specified in the advice. If a successful comparison is made then advice code inclusion can be carried out.

### 6.4.3 A sample application

The following application tests the implemented aspect-orientation.

There is an aspect module and a base module that the aspect module is applied to.

The aspect module has three procedures that are either run before or after certain named procedures. The base module simply calls its procedures and tests if the advice code is called. The code can be found in the Appendix, section 5.2.

![Command Prompt]

**Figure 6.4.3.1 Sample application results**

The above results show the order the advices are called in the values stored in the counters as the application progresses.
6.4.4 Support within eclipse

6.4.4.1 Changes to Parser

Once aspect-orientation was integrated directly into the compiler, the “Component Pascal” plug-in for “eclipse” was only required to provide high-level visual support for aspect modules.

Once again the similarity between aspect modules and normal “component pascal” modules was beneficial. The previously written java parser for the content outline was modified to parse aspect modules. This required modifying the parser to recognize the heading of an aspect module and enabling the parser to parse advices.

Parsing advices was similar to parsing procedures, as their structure was the same. Extra code needed to be added to parse an advice heading. The advice body is the same as a procedure and hence the code here could be re-used.

As described earlier in section 5.4.12.1 domain objects were created for the different members of a component pascal program. Domain objects needed to be created for advices as well. The following domain classes were added.

ComponentPascalAdvice – similar to ComponentPascalProcedure, used to represent an advice.
ComponentPascalAdvices – collection of ComponentPascalAdvice objects.

An instance of ComponentPascalAdvices is contained in ComponentPascalRoot and all advices are displayed in the outline for an aspect module.

6.4.4.2 Compilation Support

The “component pascal” plug-in contained support for the component pascal compiler. On a request for compilation the plug-in retrieves the base module for the project and uses the cpmake utility to compile the entire project.
The compilation support needed to be aware of aspect modules within the project. An extra option was added in the menus for the creation of aspect modules. On creation of an aspect module the project description file is retrieved and a comment added to indicate the existence of an aspect module, and its name. Figure 5.4.2.1.2 shows a project description file. The comment tag in the xml markup indicates the base module. The aspect module is contained in similar fashion. Below is markup for the project description for the sample application previously described in section 5.5.8.

```xml
<projectDescription>
    <name>test</name>
    <comment>BASEMODULE:base.cp</comment>
    <comment>ASPECTMODULE:sample.cp</comment>
    <projects/>
    <buildSpec/>
    <natures>
        <nature>org.eclipse.cpdt.ComponentPascalNature</nature>
    </natures>
</projectDescription>
```

On compilation the plug-in checks for the existence of an aspect module within the project. If one is found it is included in the compilation.
6.5 Summary

Incorporating aspect-orientation into “component pascal” was a complex task. After being convinced of the benefits of this programming paradigm it was required to have a detailed understanding of the inner workings of the component pascal compiler to incorporate aspect-orientation.

Many benefits were gained from the existing structure of the compiler. As component pascal is object-oriented, design considerations for aspects were made keeping in mind the existing design of the compiler. The existing code base was re-used and as a result minimal code changes were required in the end.

Aspect-orientation is now fully functional, though there are many opportunities to further enhance support for aspect-orientation within the “eclipse” plug-in. Further extensions would make its implementation as generic as possible and in the process provide an advanced separation of concerns.
7 Further work

As with any research there is always scope for further extensions and refinements to the existing software. Further work to this research can be classified broadly as additional features to the existing implementation and further visual support. These are described further in this chapter.

7.1 Additional Features

The following additional features will further enhance the software development experience within the component pascal plug-in.

7.1.1 Look-ahead typing

Once added to the component pascal editor, “look ahead typing” would prompt the developer and enhance code completion time. This would save time that would normally be spent browsing APIs. This would require an in-memory representation of all the modules in a project and their members. This could be easily done, as a parser already exists for the component pascal outline that creates domain objects from source code and populates the outline. The same domain objects can be used further to provide look ahead typing.

7.1.2 Debugging capability

The plug-in does not provide any debugging capability for component pascal programs at present. It would aid in program construction and maintenance if it were possible to watch the state of a running program and create breakpoints in the code to examine the call stack. This will be quite a complex task and would require support from the compiler that could be further enhanced with visual support within “eclipse”.

Page 74
7.1.3 Preference and Property pages

Preference and property pages will allow developers to set preferences for a component pascal project and view properties of component pascal modules and projects. Support for the same already exists in the eclipse API and can be customized to provide the required functionality.

7.1.4 Advanced wizards

The wizards in the “component pascal” plug-in currently perform basic tasks such as creation of new resources and creation of basic code skeletons. These could be further enhanced to be multi-page wizards that could gather additional details and create detailed code skeletons thus reducing the possibility of errors in programs. Moreover these wizards need not be invoked only on the creation of resources but could also be invoked for tasks such as the addition of new program members such as types and procedures. For example in the creation of a new type, instead of having to write one from scratch a developer would be able to assemble one by selecting its key components and then select from existing types in the project that it may want to extend or include in its implementation. This would help as all the information relevant to a task would be available in a single location and would not require the developer to navigate the project multiple times.

7.1.5 Extensions to component pascal outline

The component pascal outline currently displays the program members and makes the source code navigable. However it does not allow the modification of these members. Development environments like “Microsoft Visual Studio” and “BEA Weblogic Workshop” provide data palettes. A data palette is like a property editor and allows a developer to change the data types of members or set values for constants. Figure 7.1.5.1 shows a data palette that is used in weblogic workshop and allows the modification of program members in addition to listing them.
7.2 Further visual support

There is opportunity to further refine the existing plug-in to provide more advanced levels of visual support for developing component Pascal programs. Such visual support would further automate development and less code would have to be written. Following are the mechanisms that could be implemented to further enhance the visual support.

7.2.1 Relational view of modules

At the moment it is only possible to view the various modules within a component Pascal project in a simple listing in the component Pascal navigator. It would be highly beneficial to have a map representation of a project that would display the object hierarchy as well as the aspects that crosscut the objects. This would enable a developer maintaining an existing code base to immediately understand the structure of a project and be enabled to make more informed decisions.

Building the above would by no means be a simple task. One way would be to maintain and XML representation or unified markup language (UML) representation of the project. The XML or UML representation could then be parsed into domain objects that could provide the basis for building a visual representation. This method would however require maintenance and would need to be manually updated each time a change is made to the project. Another method would be to directly parse the existing code base
and then ascertain the hierarchy and build the visual representation. This would be harder to develop but would require no maintenance at all.

### 7.2.2 Aspect Map

An aspect map would graphically display a many-to-many mapping between aspect modules and normal component pascal modules. The aspect map would list relevant program members, advices for aspect modules and procedures for normal modules and display connections between them to represent which aspect code is applied where. A similar model is used in “BEA Weblogic Workshop” as shown in figure 7.2.2.1 where the mapping defines data transformation between two separate xml schemas.

![Figure 7.2.2.1 XML Schema Mapping in Weblogic Workshop](image)

The above could be further enhanced from being a simple representation to an additional feature using which a developer could map advice code application and have skeleton code for an aspect module created automatically. The developer would then
have to simply provide the exact behaviour. This would result in a more maintainable and uniform code base.

7.2.3 Multiple views of software

For any module in a component pascal project it should be possible to have different views to look at the modules differently. Possible views would be

1. Source view – to display the source code.
2. Design view – to display the source code graphically and provide the opportunity to add program members visually.
3. Flow/Structural view – this could be a cut-down version of the previously described relational view and would display graphically the position of a module in the project hierarchy.

Together the above views would ease the software development process. They would have to be integrated into the component pascal editor and could be accessed via a tabbed interface, making it easy to switch between different views.
7.3 Summary

The aspect-oriented extensions to “component pascal” can be extended to provide greater functionality to developers. However, much of this depends on the visual support that is available within “eclipse”.

The visual support within “eclipse” is set to increase with the inclusion of BEA\textsuperscript{10} as a strategic developer. BEA is also involved with Project Pollinate, an Eclipse-based development environment and toolset designed to integrate with Apache Beehive. (BEA donated the application framework in its WebLogic Workshop Java IDE, code-named Beehive, to the Apache Software Foundation last May) [48]. BEA also plans to retool “Weblogic Workshop” [49] around the “Eclipse” framework.

This will mean more tools within the “eclipse” API for integrating advanced visual support into eclipse plug-ins.

The future of development tools lies in high-end visual IDEs where developers no longer need to concern themselves with low-level details as this is abstracted by the IDEs and the developer simply needs to assemble and integrate applications. It would be reasonable to expect the same from the component pascal plug-in as more tasks are automated and the plug-in evolves into a visual environment.

The issue with the above however is that, as developers no longer need to be concerned with low-level details they will no longer know what is happening “under the hood” and may only have a superficial understanding of the software they are developing. A lack of detailed internal knowledge may also compromise the security and reliability of applications that are developed. It will also affect application integration and application

\textsuperscript{10} BEA Systems, Inc. is a leading application infrastructure software company, providing the enterprise software foundation that allows companies to benefit from service-oriented architectures. BEA provides the enterprise software foundation for more than 15,000 customers around the world, including the majority of the Fortune Global 500. BEA helps companies evolve their existing enterprise software applications from inflexible, redundant, legacy architectures to highly responsive, mature Web infrastructures.
maintenance, as these usually require a detailed understanding of the working of the application.

Hence, it will be important to find an appropriate balance. A successful developer will be one who will be able to understand internal workings of applications and at the same time be able to use sophisticated high-level tools to create these applications.
8 Conclusion

The sophisticated commercial tools in use today accelerate the software development process and greatly reduce development time. Open source tools like "eclipse" have made a great contribution to the community as they make these tools available to all developers; moreover, they also allow developers to be involved in the continued development of the tool itself. The eclipse community too is growing at a rapid rate. The "eclipse" foundation continues to grow and new members are added to the strategic developer board frequently. There is no doubt that an extremely successful future lies ahead for eclipse and this is the definitive development platform available today.

The component pascal plug-in for eclipse enables the development of component pascal projects in a comprehensive visual environment. It too is open source and the code is available by request to anyone wishing to further enhance the same.

During the course of writing this thesis the benefits of aspect-oriented programming became clear. However like any other technology it has to mature further before it is used in mainstream commercial development. The adoption of aspect-orientation requires a change in the mindset of developers since for them to successfully implement this technology they need to change their fundamental approach to solving problems. The more this technology is used, the faster this will happen. However it can be said that aspect-orientation is probably now at the stage where object-orientation was in its infancy. Whether aspect-orientation progresses as a successor to object-orientation remains to be seen.

Aspect-orientation has been incorporated into component pascal. While fully functional it is to be noted that there is still room for additional features. Though not implemented as yet, some of these have been discussed previously.

I continue to use "eclipse", almost on a daily basis, and hope that there is eventually wider acceptance of technologies that allow a better-defined separation of concerns.
9 Appendix

9.1 Manifest File

```xml
<?xml version="1.0" encoding="UTF-8"?>

<!-- Component Pascal Plugin -->

<plugin name="Component Pascal Project" id="org.eclipse.cpdt" version="1.0.0" provider-name = "Abhishek Singh" class="org.eclipse.cpdt.core.ComponentPascalPlugin">
  <requires>
    <import plugin="org.apache.xerces"/>
    <import plugin="org.eclipse.core.resources"/>
    <import plugin="org.eclipse.core.runtime"/>
    <import plugin="org.eclipse.help"/>
    <import plugin="org.eclipse.swt"/>
    <import plugin="org.eclipse.search"/>
    <import plugin="org.eclipse.ui"/>
  </requires>

  <runtime>
    <library name="ComponentPascal.jar">
      <export name="*"/>
    </library>
  </runtime>

  <!-- Accelerator Configuration -->
  <extension point="org.eclipse.ui.acceleratorConfigurations">
    <acceleratorConfiguration id="org.eclipse.cpdt.buildAcceleratorConfiguration" name="Component Pascal" description="Component Pascal Build accelerator configuration">
    </acceleratorConfiguration>
  </extension>

  <!-- Accelerator Sets -->
  <extension point="org.eclipse.ui.acceleratorSets">
    <acceleratorSet configurationId="org.eclipse.cpdt.buildAcceleratorConfiguration" scopeld="org.eclipse.ui.globalScope">
      <accelerator id="org.eclipse.cpdt.CompileProjectAction" key="F3">
        </accelerator>
    </acceleratorSet>
  </extension>

  <!-- Action Sets -->
  <extension point="org.eclipse.ui.actionSets">
    </extension>
  </extension>
</plugin>
```
<actionSet id="org.eclipse.cpdt.ui.CompileActionSet"
label="Compile"
visible="true">

<menu id="compileMenu"
label="Build"
path="project">
<separator name="slot1"/>
<separator name="slot2"/>
<separator name="slot3"/>
</menu>

$action id="org.eclipse.cpdt.ui.CompileActionSet.CompileAction"
menubarPath="compileMenu/slot1"
label="Compile File"
icont="icons/compile.gif"

class="org.eclipse.cpdt.internal.core.builder.CompileCurrentAction"/>

<action id="org.eclipse.cpdt.RunProjectAction"
menubarPath="compileMenu/slot2"
label="Run Project"
icont="icons/run.gif"

class="org.eclipse.cpdt.actions.RunProjectAction">
</action>

<action id="org.eclipse.cpdt.CompileProjectAction"
menubarPath="compileMenu/slot2"
label="Compile Project"
icont="icons/compile.gif"
accelerator="org.eclipse.cpdt.CompileProjectAction"
class="org.eclipse.cpdt.actions.CompileProjectAction">
</action>

<action id="org.eclipse.cpdt.navigator.NewModuleAction"
label="Module"
menubarPath="Normal/JavaWizards"
class="org.eclipse.cpdt.navigator.NewModuleAction"
icont="icons/cp.gif"
enablesFor="1">
</action>

</actionSet>
</extension>

<!-- Editors -->
<extension point="org.eclipse.ui.editors">
<editor id="org.eclipse.cpdt.editors.ComponentPascalEditor"
name="Component Pascal Editor"
class="org.eclipse.cpdt.ui.editors.ComponentPascalEditor"
icont="icons/cp.gif"
extensions="cp">
</editor>
</extension>

<!-- Views -->
<extension point="org.eclipse.ui.views">
<category id = "org.eclipse.cpdt"
name = "Component Pascal"/>
<category id = "org.eclipse.cpdt.navigator"
Visual, aspect-oriented programming tools for Component Pascal in Eclipse

name = "Navigator" />

<view id="org.eclipse.cpdt.ComponentPascalConsole"
name = "Component Pascal Console"
category = "org.eclipse.cpdt"
icon = "icons\console.gif"
class = "org.eclipse.cpdt.ComponentPascalConsole">
</view>

<view id="org.eclipse.cpdt.ComponentPascalProjectView"
name="Component Pascal Navigator"
category = "org.eclipse.cpdt.navigator"
icon = "icons\navigator.gif"
class="org.eclipse.cpdt.navigator.ComponentPascalProjectView">
</view>

</extension>

<!-- Pop up Menus -->
<extension point = "org.eclipse.ui.popupMenus">
<objectContribution id="org.eclipse.cpdt.navigator.popupmenu"
objectClass="org.eclipse.cpdt.navigator.ComponentPascalProject">
<menu id="NewMenu"
path="additions"
label="New">
<separator name="NewGroup"/>
<separator name="ActionsGroup"/>
</menu>

<action id="org.eclipse.cpdt.navigator.PropertiesAction"
label="Properties"
menubarPath="additions"
class="org.eclipse.cpdt.navigator.PropertiesAction"
enablesFor="1">
</action>

<action id="org.eclipse.cpdt.navigator.NewModuleAction"
label="Module"
menubarPath="NewMenu/NewGroup"
class="org.eclipse.cpdt.navigator.NewModuleAction"
icon="icons\cp.gif"
enablesFor="1">
</action>

<action id="org.eclipse.cpdt.navigator.NewBaseModuleAction"
label="Base Module"
menubarPath="NewMenu/NewGroup"
class="org.eclipse.cpdt.navigator.NewBaseModuleAction"
icon="icons\cp.gif"
enablesFor="1">
</action>

<action id="org.eclipse.cpdt.navigator.DeleteAction"
label="Delete"
menubarPath="additions"
class="org.eclipse.cpdt.navigator.DeleteAction"
enablesFor="1">
</action>
</objectContribution>

<objectContribution id="org.eclipse.cpdt.navigator.popupmenu2"
objectClass="org.eclipse.cpdt.navigator.ComponentPascalModule"/>
Visual, aspect-oriented programming tools for Component Pascal in Eclipse

<action id="org.eclipse.cpdt.navigator.PropertiesAction"
label="Properties"
menubarPath="additions"
class="org.eclipse.cpdt.navigator.PropertiesAction"
enablesFor="1">
</action>

<action id="org.eclipse.cpdt.navigator.DeleteAction"
label="Delete"
menubarPath="additions"
class="org.eclipse.cpdt.navigator.DeleteAction"
enablesFor="1">
</action>

<action id="org.eclipse.cpdt.navigator.OpenAction"
label="Open"
menubarPath="additions"
class="org.eclipse.cpdt.navigator.OpenAction"
enablesFor="1">
</action>

</objectContribution>
</extension>

<!-- Resource Filters -->
<extension point="org.eclipse.ui.resourceFilters">
<filter selected="false" pattern="*.cp">
</filter>
</extension>

<!-- Wizards -->
<extension point = "org.eclipse.ui.newWizards">

<category
id = "org.eclipse.ui.componentpascal.new"
name="Component Pascal">
</category>

<wizard id = "org.eclipse.cpdt.wizards.new.project"
name = "Component Pascal Project"
class="org.eclipse.cpdt.wizards<ComponentPascalProjectCreationWizard"
category="org.eclipse.ui.componentpascal.new"
project="true"
icon="icons\cp.gif">
<description>
Create a new Component Pascal File
</description>

<selection
class="org.eclipse.core.resources.IResource">
</selection>
</wizard>

<wizard id = "org.eclipse.cpdt.wizards.new.file"
name = "Module"
class="org.eclipse.cpdt.wizards.ComponentPascalModuleCreationWizard"
Visual, aspect-oriented programming tools for Component Pascal in Eclipse

category="org.eclipse.ui.componentpascal.new"
icon="icons\cp.gif">

<description>
Create a new Component Pascal File
</description>

<selection>
class="org.eclipse.core.resources.IResource">
</selection>

</wizard>

<wizard id = "org.eclipse.cpd.wizards.NewBaseModule"
name = "Base Module"
category="org.eclipse.ui.componentpascal.new"
icon="icons\cp.gif">
<description>
Create a new Component Pascal Base Module
</description>

<selection>
class="org.eclipse.core.resources.IResource">
</selection>

</wizard>

<extension>

<!-- Preferences -->
<extension point = "org.eclipse.ui.preferencePages">
<page id="org.eclipse.ui.componentpascal.Page1"
class="org.eclipse.ui.componentpascal.ReadmePreferencePage"
name="Component Pascal">
</page>

</extension>

<!-- Perspectives -->
<extension point="org.eclipse.ui.perspectives">
<perspective id="org.eclipse.cpd.componentpascalPerspective"
name="Component Pascal"
class="org.eclipse.cpd.core.ComponentPascalPerspective"
icon="icons\cp.gif">
</perspective>
</extension>

<!-- Project Natures -->
<extension point="org.eclipse.core.resources.natures">
id="ComponentPascalNature"
name="Component Pascal Nature">

<runtime>
<run
class="org.eclipse.componentpascal.resources.ComponentPascalNature">
</run>

</runtime>
Visual, aspect-oriented programming tools for Component Pascal in Eclipse

</extension>
</plugin>
9.2 Sample application

9.2.1 Aspect module

ASPECT MODULE Sample;

IMPORT Console;

VAR bCount,aCount,dCount:INTEGER;

ADVICE pBCount;
BEFORE bCall();
BEGIN
INC(bCount);
Console.WriteString("called pBCount");
Console.WriteLn;
Console.WriteInt(bCount,2);
Console.WriteLn;
END pBCount;

ADVICE pACount;
AFTER aCall();
BEGIN
INC(aCount);
Console.WriteString("called pACount");
Console.WriteLn;
Console.WriteInt(aCount,2);
Console.WriteLn;
END pACount;

ADVICE pDCount;
BEFORE dCall();
BEGIN
INC(dCount);
Console.WriteString("called pDCount");
Console.WriteLn;
Console.WriteInt(dCount,2);
Console.WriteLn;
END pDCount;

END Sample.
MODULE Base;
IMPORT CPmain, Console;

PROCEDURE bCall();
BEGIN
  Console.WriteString("**BCALL**");
  Console.WriteLn;
END bCall;

PROCEDURE aCall();
BEGIN
  Console.WriteString("**ACALL**");
  Console.WriteLn;
END aCall;

PROCEDURE dCall();
BEGIN
  Console.WriteString("**DCALL**");
  Console.WriteLn;
END dCall;

BEGIN
  bCall();
aCall();
bCall();
dCall();
dCall();
END Base.
10 Bibliography

1. Integrating the BETA language with Eclipse, 2002 [online], available
3. Hipikat: Recommending useful software artifacts, 2002 [online], available
   http://www.cs.ubc.ca/labs/spl/projects/hipikat/
5. Guard: Relative Debugging, 2002 [online], available
6. Knowledge Based Reverse Engineering, 2002 [online], available
   http://www.site.uottawa.ca/~tcl/kbre/
7. Obasco (Object, Aspect, Components), 2002 [online], available
   http://www.emn.fr/x-info/Eclipse
8. AspectJ – Aspect-Oriented Programming (AOP) for Java, [online], available
   http://www.aspectj.org
14. Matt Liotta, What’s Eclipse, October 2002 [online], available
   http://portals.devx.com/ibm/Article/6884
15. Evolutionary Development, 1996 [online], available
   http://www.rap.ucar.edu/technology/software-eng/evolutionary-development.html
16. Object Technology International [online], available http://www.oti.com
17. Centre for Information Technology Innovation [online], available http://www.citi.qut.edu.au
18. Distributed Systems Technology Centre (DSTC) [online], available http://www.dstc.edu.au
Visual, aspect-oriented programming tools for Component Pascal in Eclipse

29. Chris Preimesberger, Eclipse (12 million downloads) shows big progress [online], available http://www.newsforge.com/software/03/10/02/0130242.shtml?tid=11
30. Microsoft Visual Studio [online], available http://msdn.microsoft.com/vstudio/
31. IBM Visual Age [online], available http://www-06.ibm.com/software/awdtools/vajava/
33. Connections between Demeter/Adaptive Programming and Aspect-Oriented Programming (AOP) [online], available http://www.ccs.neu.edu/home/lieber/connection-to-aop.html
34. Aspect-Oriented Research on Composition Filters [online], available http://trese.cs.utwente.nl/oldhtml/composition_filters/
35. Feature-Based Design Rationale Capture Method for Requirements Tracing [online], available http://www.sei.cmu.edu/str/descriptions/featbased_body.html
36. Generative programming [online], available www.generative-programming.org
38. Subject-oriented programming [online], available http://www.research.ibm.com/sop/


42. AspectC++ [online], available http://www.aspectc.org/

43. Yvonne Coady, Gregor Kiczales, Mike Feeley, Greg Smolyn, Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code, *FSE 2001*.

44. AspectS [online], available http://www.prakinf.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/

45. JAC - A Framework for Aspect-Oriented Programming in Java [online], available http://jac.objectweb.org/

46. HyperJ [online], available http://www.alphaworks.ibm.com/tech/hyperj

47. Borland JBuilder, [online], available http://www.borland.com/jbuilder/


49. BEA Weblogic Workshop [online], available http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/workshop