OBJECT-ORIENTED LITERATE PROGRAMMING

OBJECT-ORIENTED LITERATE PROGRAMMING

By
MING YU ZHAO, B.SC.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree of

Master of Science
Department of Computing and Software
McMaster University

MASTER OF SCIENCE (2007)
(Computing and Software)

McMaster University
Hamilton, Ontario

TITLE: Object-Oriented Literate Programming

AUTHOR: Ming Yu Zhao, B.Sc. (Dalian University of Technology)

SUPERVISOR: Dr. Emil Sekerinski

NUMBER OF PAGES: x, 179

Abstract

During the past decades, programming methodology has seen an improvement by structured programming and object-oriented programming (OOP), leading to software that is more reliable and easier to develop. However, software engineers are still dealing with problems in processing associated documentation. Literate programming was introduced by Donald Knuth in the early 80's as an approach to produce programs together with their documentation in a way that is aimed at consumption by humans rather than by compilers. However, dated and complex features, dependence on formatting and programming language, and a lack of methodology prevented the method from gaining in popularity.

In this thesis, we propose an approach to "integrate" OOP with literate programming in order to present and document the whole design in a consistent and maintainable way. In our approach, both program code and corresponding documentation are generated from the same source. The resulting documentation consists of code chunks with detailed explanations, as well as automatically generated class diagrams with varying levels of detail. A tool, *Spark*, has been developed and applied to the design of a Transit Information System from requirement analysis to testing. Spark was also used in its own development.

Acknowledgements

I would like to give my sincere thanks to my supervisor, Dr. Emil Sekerinski, for his thoughtful guidance, constant encouragement and generous support throughout my study.

In addition, I am indebted to my examination committee, Dr. Ned Nedialkov and Dr. Jacques Carette, who have each taken time to offer suggestions and guidance to improve this work. Also, thanks to Mr. Jian Xu, Mr. Daniel Zingaro, as well as other friends, who have helped me in the passed two years.

Especially, special thanks go to my wife, Ru Wei, for her love, understanding and support and to my parents, who taught me how to navigate my life.

Contents

Abstract iii

Acknowledgements iv

1 Introduction 1

1.1 Why Object-Oriented Literate Programming 1

1.2 Contributions 3

1.3 Structure of the Thesis 3

2 Related Work 5

2.1 Object-Oriented Programming 5

2.1.1 Class and Object 5

2.1.2 Inheritance 6

2.1.3 Assertion 7

2.1.4 Garbage Collection 8

2.1.5 Object-Oriented Approaches 9

2.1.6 Critiques 15

2.2 Literate Programming 18

2.2.1 Advantages 19

2.2.2 Disadvantages 21

2.3 Summary 22

3 Object-Oriented Literate Programming 24

3.1 Overview 24

3.2 Document Formatting Languages 26

3.3 Programming Languages 28
3.4 Spark 29
3.5 Editors 30
3.6 Reference Developing Process 31
3.6.1 Analysis 31
3.6.2 Design 32
3.6.3 Implementing 33
3.6.4 Testing 33
3.7 Summary 34

4 Transit Information System Case Study 35
4.1 Transit Information System 35
4.1.1 Requirements 35
4.1.2 An Overview 36
4.1.3 Dictionary 36
4.1.4 Identifying Class 38
4.1.5 Identifying Operations 39
4.1.6 Consulting The Library of Model 39
4.1.7 Applying Design Patterns 39
4.1.8 Algorithms Design 44
4.1.9 Automatic Code Listing 71
4.1.10 Testing 72

5 Implementation 75
5.1 Introduction 75
5.2 Graphic Notation describer 76
5.3 Front End 78
5.4 Back End 100
5.5 Testing 109
5.5.1 Usability Testing 109
5.5.2 Unit Testing 110
5.5.3 Integration Testing 115
5.5.4 System Testing 115

6 Conclusion and Future Work **116**

A Installation **118**
A.1 Perl 118
A.2 Graphviz 118
A.3 AsciiDoc 119
A.4 Python 119
A.5 SmartEiffel 119

B Source Code of Case Study **120**

C Generated Code of Case Study **150**

D Reference Manual of Spark **170**
D.1 Code Block Tag 170
D.2 Graphic Notation Setting 170
D.3 Program Code Quotation 171

E Document Structure of AsciiDoc **172**

F Syntax of Dot **174**

List of Figures

2.1 The initial structure of the duck game 15
2.2 The refined structure using strategy design pattern 16
2.3 Dual usage of a WEB file (adapted from [24]) 19

3.1 An example of automatic class diagram. 25
3.2 Overview of OOLP environment 26
3.3 The Workflow of Spark 29

4.1 Object model of transit information system. 37
4.2 The attributes of class HPTA_TRANSIT_INFO 40
4.3 Class CONNECTION_FINDER 41
4.4 Class database 41
4.5 The hirarchy of databases 44
4.6 The attributes of FILE_DATABASE 45
4.7 The hierarchy of class CONNECTION_FINDER 50
4.8 The attributes of class KNOT 50
4.9 The methods of class HPTA_TRANSIT_INFO 59
4.10 The methods of class ROUTE 66
4.11 The methods of class STAFF 71

5.1 Top-level structure for Spark 76
5.2 Top-level structure for Spark front end 78
5.3 The structure of Module parseCode 81
5.4 Top-level structure for Spark back end 100
5.5 Vertical testing 114

5.6 Horizontal testing 114
5.7 Method only testing 115
5.8 Attribute only testing 115
5.9 Concise form testing 115
5.10 Brief form testing 115

List of Tables

5.1 The block structure of graphic notation describer. 77

E.1 The block structure of AsciiDoc. 173

F.1 Abstract grammar for the dot language 175

Chapter 1

Introduction

1.1 Why Object-Oriented Literate Programming

Although object-oriented programming (OOP) is common in the area of computer software development, it is only one of evolutionary extensions to get to a software revolution [10]. The past several decades saw the development from early imperative programming through to OOP and from unstructured programming to structured one. Each programming paradigm and every progress has shortened the gap between human beings and machines, the real world and computer software [36, 40]. However, none of them escape the limitation of instructing a computer what to do. To make a further progress, we propose a new approach, *object-oriented literate programming* (OOLP), which combines the benefits of both OOP and literate programming and is assured with automated support. Such an approach is desirable for the following reasons:

- **Language-independence**: Donald Knuth's literate programming encourages programmers to concentrate on explaining to human beings what they want a computer to do, which does lead to significantly better documentation. However, the fact that it fails to employ new programming techniques and its doomed complex features and language-dependence made one still wait for a better alternative [38]: in spite of the support of tools such as CWEB [25], noweb [38], FunnelWeb [1], FWEB [27], and OCAMAWEB [29], the dependence on certain programming language or documentation formatting language is predeterminate. Our approach is more flexible: software engineers can choose their favorites, both programming languages, like Eiffel [32], Lime [41], Java, or C#, and formatting languages, such as LaTeX [26, 33], AsciiDoc [2], and DocBook [45].

- **Consistent, accurate, and readable documentation**: it is hard to say that any software application without qualified documentation is valuable, because documentation absences, errors and even omissions can undoubtedly lead to disasters for both end-users and successive software engineers. In fact, all software development projects must generate a large amount of associated documentation; a high proportion of software process costs is incurred in producing documentation. In our approach, any change in program code can affect its documentation directly and every algorithm, even a single variable, is discussed properly. This kind of work of literature makes reading technical documentation fun.

- **Automated tools support**: one of the main benefits of this approach is the ability to use software tools to analyze program code and generate design diagrams. With such tool support, we not only hope for an increase in development speed, but also for progression towards a more standardized form of documentation.

- **Object-oriented**: programming in an object-oriented language, like Java, is neither a necessary nor sufficient condition for being object-oriented; the key is the object-oriented design technique [5]. This approach maintains the concepts and techniques of OOP, so programs still may be seen as a collection of cooperating objects, which makes the code characterized by flexibility and maintainability, as opposed to traditional view in which a program may be seen as a list of instructions to the computer.

- **Simplicity**: it is because of the feature of language-independence that little extra training is needed. Experienced programmers can begin their OOLP immediately with their favorite OOP language, like Eiffel, and documentation formatting language, say LaTeX; as to beginners, they can choose the simplest but powerful enough ones and get jobs done perfectly in a short term. In addition, there is no extra cost needed on other application software or equipments; a text editor is good enough. Configurable, platform-independent and open-source text editors such as VIM [35] and Emacs [7], are perfect for our job.

1.2 Contributions

My contributions include:

- design a scheme to combine OOP and literate programming,
- design and implement Spark, a set of tools supporting this technique,
- propose a reference development process,
- develop a case study, *Transit Information System*, in the way of OOLP.

OOLP, as well as Spark, is an ongoing research project and many aspects are likely to evolve over time. Therefore, the design of schemes and the implementation of Spark should be as general as possible. The code of Spark is written entirely in *perl* [11, 44], which is good at text processing. *Graphviz* [14, 20] is employed to produce diagrams, since we want to avoid looking deeply into layout algorithms.

1.3 Structure of the Thesis

The remainder of this thesis is organized as follows:

- Chapter 2 surveys the related work on object-oriented techniques, and literate programming. Both advantages and limitations of them are discussed.
- Chapter 3 introduces OOLP by describing its features and predominance as well as its supporting tools, Spark.
- Chapter 4 studies the case of Transit Information System, which is developed entirely in the way of OOLP.
- Chapter 5 focuses on the implementation and testing of Spark.

- Chapter 6 draws the conclusions of our work, in addition to discussing future work.

Chapter 2

Related Work

In this chapter, a survey of the existing work on both object-oriented techniques and literate programming implementations is developed where both its advantages and disadvantages are analyzed.

2.1 Object-Oriented Programming

After the 1990s, OOP became a mainstream technique in software development. It is widely used successfully in various applications including compiler, graphics, user interfaces, databases, object-oriented languages, computer-aided design systems, games, and control system [39]. Basing on abstraction, encapsulation, and polymorphism, OOP has more predominance than traditional approaches on *reliability*, *modularity*, *compatibility*, *portability*, and *efficiency* [31]. As studied in detail in subsequent sections, object-oriented techniques, some of which come from previously established paradigms, can significantly improve these quality factors, which is why it is so attractive.

2.1.1 Class and Object

In OOP, rather than just a list of instructions to the computer, an executing program may be seen as a collection of cooperating objects, which distinguishes object-oriented approach from other non-object-oriented methods [12]. Objects could be anything, including not only the entities, such as a student, an university, a hospital and a car, but also the abstract concept, such as a plan and an event, so the models or programs conceived in such a natural way are more understandable.

As the smallest module of programs, an object is equipped with essential attributes and behaviors and becomes active by executing one of its methods, in which it may change its state or send messages to other objects, which in turn invokes the execution of the corresponding methods of those objects. Compare to structured methods, they focus on functional decomposition; once a complex program has been decomposed into some programmable sub-functions, the software certainly will work mechanically just like an assembly line. It is very likely that in order to produce some new "parts", developers have to reconstruct a new "assembly line" for both new "parts" and old ones or maintain two or more "assembly lines" for all at the same time. The following workload and risk of new bugs could be a disaster for everybody concerned. While for object-oriented methods, developers only need to prepare some new class(es) for such an extension. Unfortunately, in practice the changes of whole workflow are more common than the ones of concrete methods used to process certain object. Hence, object-oriented programs can enjoy better extendibility and stability.

However, rather than the object, the class is the central concept of OOP. A class is a kind of abstract data type equipped with a possibly partial implementation, through which classes establish the necessary link with software construction — design and implementation. Much of the power of the object-oriented method, such as inheritance, encapsulation, and polymorphism, derives from this concept.

2.1.2 Inheritance

In OOP, inheritance is a strong conceptual relation that can hold between classes: a class may be an extension, specialization or combination of others. It is common that new software expands on previous developments, but problems of repetition and variation are largely ignored by traditional design methods. Writing the same code over and over again means not only waste of time, but also the risk of incoming inconsistencies and errors. With the support of inheritance, one class (subclass) can introduce itself by capturing the striking commonalities that exist within one or more mature classes (superclass) and adding the many differences that characterize individual cases.

The advantages of inheritance also cover a faster modifying mechanism: by inheritance, any modification in high level can affect the other related parts of system immediately. In addition, inheritance divides a system into different abstract levels, where developers can focus on them one by one without the bother of trivial details at the very beginning.

When a class inherits its members from more than one ancestor class, this is called *multiple inheritance*, which is a debatable feature. Generally speaking, multiple inheritance make the ancestor relation complex, so any change made in a certain ancestor may result in some unexpected effects on its successors and a compiler has to face the problem of processing those identical members. Nevertheless, there are still some object-oriented programming languages, such as Eiffel, C++, and perl, that support multiple inheritance with different ways. Eiffel will automatically join the members with same name and implementation together if they are not renamed by the programmer explicitly. C++ requires the programmer to state the inheritance path in detail. Perl uses the list of classes to inherit from as an ordered list. These contrast with Java and C#, which allow classes to inherit from multiple interfaces rather than multiple inheritance; this results in no ambiguity.

2.1.3 Assertion

For software programmers, reliability refers to the correctness and robustness of software. In order to improve reliability, assertions, boolean expressions, usually written as annotations are employed to specify what a system behavior is supposed to do rather than how it does. The use of assertion dates back to Hoare's 1969 paper on formal verification [22]. A correctness formula (also called Hoare triple) is an expression of the form

$$P \quad \{Q\} \quad R \tag{2.1}$$

where Q denotes a program; P and R, the properties of Q, are called precondition and postcondition respectively. However, it is only a mathematical notation used to constrain the properties of programs.

Design by Contract (DBC) proposed by Meyer as a trademark of Eiffel is a formal technique for dynamically checking specification violation during runtime [31]. The main idea behind DBC is that a client and a supplier, the elements of a software system, collaborate with each other according to a "contract". For example, suppose method M of class C provides a certain functionality needed by class D. Then, class D, the client, must fulfil a certain requirement, the precondition of method M, before invoking method M and as a result, class C, the provider, must ensure a certain property on the exit of method M, its postcondition. That is:

- precondition constrains the client, so it is an obligation for the client and a benefit for the supplier.

- postcondition constrains the supplier, so it is an obligation for the supplier and a benefit for the client.

Only precondition and postcondition are not enough, because they are used to describe the properties of individual methods. For the global states of a class, Eiffel supports the concept of class invariant. An assertion I is a correct class invariant for class C if and only if it meets the following two conditions:

- every constructor of C, when applied to arguments satisfying its precondition in a state where the attributes have their default values, yields a state satisfying I

- every exported method of C, when applied to arguments satisfying its precondition and a state satisfying I, yields a state satisfying I

Assertions used in preconditions, postcondition, and invariant express the semantic constraints on a class, so it is possible for us to define formally what it means for the class to be correct.

2.1.4 Garbage Collection

Garbage collection (GC) as one of automatic memory management techniques is used in most good object-oriented environments. The main idea is that a garbage collector, a facility included in the runtime system for a programming language, takes care of both detecting and reclaiming unreachable objects. With this technique, software developers do not need to worry about memory wasted on useless objects created by their software any more, so the reliability and timeliness of software products will surely benefit from it. GC was invented by John McCarthy around 1959 to solve the problems of manual memory management in his Lisp programming language [30]. The basic principle of how a mark and sweep garbage collector works is:

- mark phase, staring from the origins, follows references recursively to traverse the active part of the structure, marking as reachable all the objects it encounters
- sweep phase traverses the whole memory structure, reclaiming unmarked elements and erasing all the marks

Classical garbage collectors are inactive as long as there is enough memory available for the application. Its advantage is that it causes no overhead before the collector is triggered and a serious potential drawback is that a complete mark-sweep cycle may take a long time — especially in a virtual memory environment. Therefore, GC is rarely used on embedded or real-time systems.

To pursue better performance, some techniques were employed. First of all, endow developers with some control over the activation and deactivation of collector cycles. If a system contains a time-critical section, which mush not be subject to any unpredictable delay, the developer may put a "stop sign" at the beginning of the section and show a "green light" at the end; and at any point where the application is known to be idle, the developer may ask collector to work immediately. In addition, ones also use *generation scavenging*, the philosophy behind which is that the more garbage collection cycles an object has survived, the better chance it has of surviving many more cycles or even remaining forever reachable. Although this technique helps through lessen the frequency of collector cycles on "old" objects, there remains a need to perform full collections occasionally. Parallel collection, one of the practical solutions for GC, requires two separate threads: the application and the collector. During the execution of an object-oriented system, the application creates as many new objects as it needs; the collector free them continuously according to the principle mentioned above.

2.1.5 Object-Oriented Approaches

In contrast with structured approaches, which focus on functional decomposition from the perspective of "input-process-output", many object-oriented approaches have been derived from these exceedingly popular object-oriented techniques discussed above. Each of them has introduced a set of new modelings or notations. The rest of this section presents summarily five popular ones: Responsibility-Driven Design (RDD) [46], Object Modeling Technique (OMT) [39], Business Object Notation (BON) [43], Catalysis [13], and Vienna Development Method (VDM++) [16].

RDD

RDD, conceived by Rebecca Wirfs-Brock in 1990, is a shift from thinking about objects as data plus algorithms to thinking about objects as roles plus responsibilities. In a responsibility-based model, objects play specific roles and occupy well-known positions in the application architecture, which is a smoothly-running community of objects. Each object is accountable for a specific portion of the work and all objects collaborate in clearly-defined ways, contacting with each other to fulfill the larger goals of the application. By creating such a "community of objects," and assigning specific responsibilities to each, developers build a collaborative and flexible model of their application. On the other hand, developing consistent control styles for similar parts of the application may reduce the maintenance costs and incorporating flexibility into the design at specific "flex points" allows for planned extensions. Therefore, responsibilities are a good way to think about the behaviors of complex software systems. RDD consists of the following steps:

- create a CRC (Class, Responsibility, Collaborator) card for each candidate class, which usually is a noun with a small well-defined set of responsibilities
- identify and assign the responsibilities to candidate classes by asking what this class's objects need to know and what steps towards accomplishing each goal these objects should be responsible for
- find collaborations for candidate classes by asking what other objects need those result
- build inheritance hierarchies for all classes if necessary.
- identify subsystems by drawing the collaborations graph and then looking for strongly coupled classes

- construct protocols for each class
- implementing design

OMT

OMT, developed circa 1991 by Rumbaugh, Blaha, Premerlani, Eddy and Lorensen, is one of popular object-oriented development methods today. It is primarily used by system and software developers supporting full life-cycle development, targeting object-oriented implementations. Because of its simple core notation, OMT has proven easy to understand, to draw, and to use. So it continues to be successful in various application domains, such as telecommunication, transportation, and compilers. OMT consists of the following phases:

- analysis phase: understand and model the application and the domain within which it operates by formal models: the object model specifies what it happens to, the dynamic model specifies when it happens, and functional model specifies what happens.
  - object model: capture the static structure of a system by showing the objects in the system, relationships between these objects, and the attributes and operations that characterize each class of objects
  - dynamic model: describe the control flow, interactions, and operating sequences of the system and consist of multiple state diagrams
  - functional model: describe computations within a system
- system design phase: determine the overall architecture of the system
  - organize the system into subsystems
  - identify concurrency
  - allocate subsystems to processors and tasks
  - handle the boundary conditions and the system resources, especially the permanent data.
  - choose software control implementation

- object design phase: determine the full definitions of the classes and associations used in the implementation, as well as the interfaces and algorithms of the methods used to implement operations.

- implementation phase: discuss the specific details for implementing a system using programming languages and database management systems.

BON

BON, developed in the early 1990s by Jean-Marc Nerson and Kim Walden, is a means of extending the higher-level concepts of Eiffel into the realm of analysis and design aided by a set of graphical notations. These graphical notations do not include the associations, multiplicities, and state-charts that can be found in nearly all object-oriented analysis and design notations today. BON consists of informal charts, static architecture, class interfaces, dynamic scenarios and nine standard tasks are grouped into three parts:

- gather analysis information
  - delineate system borderline
  - list candidate classes
  - select classes and group into clusters
- describe the gathered structure
  - define classes
  - sketch system behavior
  - define public features
- design a computational model
  - refine system
  - generalize
  - complete and review system

Catalysis

Catalysis coined by Desmond D'Souza and Alan Cameron Wills is a method for component-based and object-oriented software development. It provides a strongly coherent set of techniques for business analysis and system development using Unified Modeling Language UML) and is characterized by following:

- **Traceability** from business models to code
- **Precision**, with clear unambiguous models and documents
- **Component** Based Development
- **Reuse** of designs, specifications, problem domain models, and even architectures
- **Scalability** from small to large teams and projects
- **Process** that is flexible yet repeatable, with multiple "routes"

Catalysis believes that these is no single process that fits every project: each one has different starting points, goals, and constraints. Therefore, it provides a list of process patterns that help developers plan a project appropriately to their situation [13].

VDM++

VDM++ is extended by Nico Plat, Paul Mukherjee and, later, Marcel Verhoef from VDM. It employs a formal notation to complement and enhance object-oriented class models and its development process consists of the following ten steps:

1. determine the purpose of the model.
2. read the requirements.
3. analyze the functional behavior from the requirements.
4. extract a list of possible classes or data types and operations. Create a dictionary.

5. sketch out representations for the classes using UML class diagrams. This includes the attributes and the associations between classes. Transfer this model to VDM++ and check its internal consistency.

6. sketch out signatures for the operations. Again, check the model's consistency in VDM++. The development is continued by adding operation signatures (the formal parameters and the result) at the class diagram level.

7. complete the class or data type definitions by determining potential invariant properties from the requirements and formalizing them. To make the model more comprehensive, it is a good idea to review the model to check coverage of the requirement. Document important properties or constraints as invariants. Before being able to validate the model created so far it is also necessary to consider how to construct instances of the different classes. In VDM++, constructors are simply written as operations with the same name as the class in which they are defined.

8. complete the operation definition by determining pre- and postcondition and operation body, modifying the type definition if necessary.

9. validate the specification using systematic testing and rapid prototyping. Three methods are used here:

    (a) integrity properties are formal descriptions of system properties that can be generated automatically by VDMTools.

    (b) VDMTools supports validation using conventional testing techniques, including features to enable test coverage documentation directly at the VDM++ level.

    (c) validation can be made executing models together with other code, e.g., a graphical front end.

10. implement the model manually or using automatic code generators that produce directly compilable code in C++ or Java.

2.1.6 Critiques

We have introduced some object-oriented techniques and five object-oriented approaches. However, it is not necessarily followed by a reusable, robust, modifiable, and maintainable software applications. The rest of this section explains several other issues that contribute to a satisfying software applications.

Design Patterns

The work of designing a good object-oriented software is easy to say, but difficult to do [18, 19]. Although design patterns may introduce some more classes through delegation and inheritance, they do provide partial solutions to some common problems, including analysis [17, 28], system design [6], middleware [34], process modeling [3], dependency management [15], and configuration management [4]. Let us take the *strategy design pattern* for instance. Suppose that there is a requirement of a duck pond simulation game, which can show a large variety of duck species swimming and making quacking sounds. Basing the standard object-oriented techniques and approaches discussed above, developers may naturally define one Duck superclass from which all other duck types inherit as shown in Figure 2.1. Since all ducks quack and swim, the superclass takes care of their implementations, while every subclass has to be responsible for implementing its own display function.

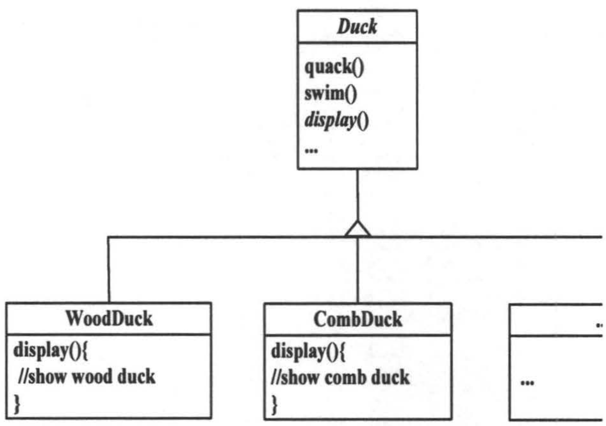


Figure 2.1: The initial structure of the duck game

Unfortunately, the extendibility of such a design structure is not satisfying. What will happen if here come two new requirements: let all existed ducks fly and add some rubber ducks, which can neither quack nor fly? Apparently, inheritance and overriding can not be the answer, because the specification will keep changing and developers will be forced to keep an eye on and possibly override fly method and quack method for every new subclass; trying to declare some interfaces, such as Flyable and Quackable, for the changes must lead to a mass of duplicate code.

The strategy design pattern seems to be a key to such a problem. The main idea of it is to decouple a policy-deciding class from a set of mechanisms so that different mechanisms can be changed transparently from a client. In other words, all the "problematic" behaviors, such as quack and fly, should be taken out of the superclass and then assigned to the specific duck according to concrete circumstances (see Figure 2.2). In this way, all the concrete strategies like FlyWithWings, FlyNoWay, Quark, and Mute can be substituted at runtime and new behaviors also can be added without modifying the other parts.

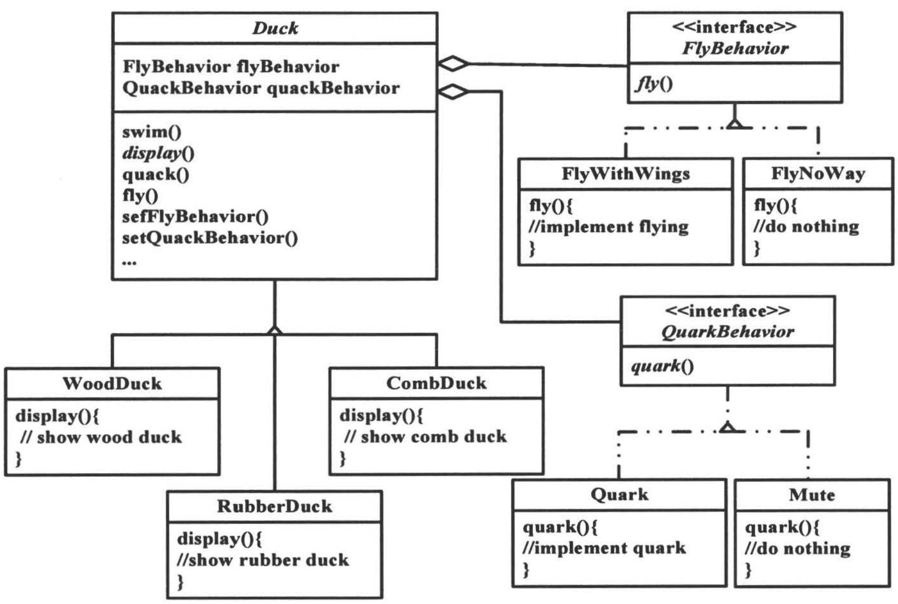


Figure 2.2: The refined structure using strategy design pattern

Through the example above, we can say that knowing the object-oriented basics, abstraction, encapsulation, polymorphism, and inheritance, as well as some popular approaches do not make one a good object-oriented designer. As practical object-oriented experience, design patterns show designers how to build systems with better qualities: reusability, extensibility, and maintainability, not concrete program code. More and more good patterns are going to be discovered by the following principles:

- Encapsulate what varies.
- Favor composition over inheritance.
- Program to interfaces, not implementations.
- Strive for loosely coupled designs between objects that interact.
- Classes should be open for extension but closed for modification.
- Depend on abstractions rather than concrete classes.
- A class should have only one reason to change.

Software Documentation

Software documentation is written text that accompanies and explains computer software. Its absence, insufficiency, or inconsistency means the loss of the partial or even total previous effort, because the program will undergo modifications due to errors or changes of requirements and reuses in other software applications. All large software development projects, irrespective of application, generate a large amount of associated documentation, mainly including the project plan, quality plan, requirements specification architecture description, design documentation, technical documentation, user manuals and test plan [42].

OOP did achieve a major improvement in the analysis and design of software, but it also suffers the problems coming from the consistency and readability of software documentation. The reason of that comes mainly from the documentation mechanism itself. Like traditional programming paradigms, OOP separates most documentation, such as design documentation and architecture description, from code, so it is hard to keep all this pivotal documentation up-to-date and synchronized. Especially for large projects and plus the time pressure, the inconsistency of documentation gets worse.

On the other hand, although technical documentation, which is used to explain class as well as its members, data structures and algorithms, is embedded within the source code as comments and may be supported by automatic documentation tools, such as doxygen [21], Javadoc, and TwinText [37], it is always short and organized in an order suitable for compilers rather than human beings.

In addition, software documentation without graphical notations is neither expressive nor appealing. In fact, most popular programming approaches today have their own set of graphical notations used to create an abstract model for their target software systems, which, especially the complex ones, enhance the importance of graphical notion. Usually, these notions are constructed by developers manually and then included into the corresponding software documentation. So incomplete changes may lead to inconsistency, which is the reason for other readers' misunderstanding.

2.2 Literate Programming

Introduced by Donald Knuth in the early 80's, literate programming is an approach that combines a programming language with a formatting language, thereby making programs more robust, more portable, and more easily to maintain than programs written only in one high-level language [9, 24, 38]. Its main idea is to treat a program as a *work of literature*, which is used to explain to human beings what it let a computer do rather than to instruct a computer what to do. The program is also viewed as a hypertext document, rather like the World Wide Web. By contrast with other programming paradigms, the program source code is embedded into documentation rather than the other way and the practitioner of literate programming needs to manipulate two kinds of languages simultaneously, neither of which can provide significantly better documentation of programs by itself.

The first published literate programming environment is WEB [23], which uses Pascal as its underlying programming language and TEX [26] for typesetting of the documentation. Pascal makes it possible to specify the algorithms formally and unambiguously, while TEX provides typographic tools to explain the local structure of such parts. The structure of WEB program may be thought of as a "web" that is made up of many interconnected *modules*, which may contain the actual program source code, abbreviations for the code, and description of the code. All the modules should be subdivided until their functionality is easily understandable. In WEB, the "bilinguist" writes such a program that serves as the source language for two different system routines as shown in Figure 2.3. Besides WEB, other implementations of this concept are CWEB [25], FWEB [27], noweb [38], FunnelWeb [1], and OCAMAWEB [29]. Some of them are different versions of WEB for documenting specific programming languages, such as C++ and Fortan, while others are documentation formatting language independent, such as noweb, and FWEB.

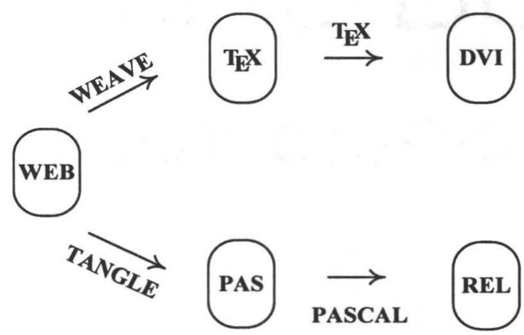


Figure 2.3: Dual usage of a WEB file (adapted from [24])

2.2.1 Advantages

As an efficient way to combine source code and its documentation, literate programming enhances the quality of programs. Its programs are characterized by flexible programming order, lossless factoring, better readability and better maintainability.

Flexible Programming Order

In literate programming, a program consists of some modules, which can be organized in arbitrary order without the constrain coming from compilers. So a programmer can choose the order best suited to explaining to human beings what he or she want a computer to do. In other words, this principle encourages the author of a literate program to take the time to consider each fragment of the program in its proper position. The reordering is especially useful for encapsulating tasks such as input validation, debugging, and printing output fit for humans.

Factoring

Traditionally, a function is the smallest ordered list of computer instructions and the compiler requires the full text of its algorithm to be held together continuously. This is the reason that overweighed code chunks can be found everywhere. To improve this situation, Knuth introduced a decomposition facility into the meta-language. With this technique, the definition can be broken into constituent parts without the extra cost for defining new functions. Therefore, every part of any algorithm can be discussed in detail sufficiently.

Readability

Knuth believes that a program should be regarded as a work of literature. By such a literary style of writing, programmers enjoy the freedom to discuss the design decisions as well as constraints that may lead to certain intricacies in their implementation. A program presented in book form is certainly characterized by better readability.

Maintainability

Since factoring and literary style endow programmers with the ability to describe their algorithms as well as the trade-offs in detail fully, every reader, including the author, can understand the program totally at any time. When an alteration is required, it should be fairly obvious which part of the "book" needs to be modified. Similarly, the description concerning such a alteration will be used as a reference for other maintenance or development later.

In a word, maximized factoring, detailed description and literate sequence bring literate programs better readability, which in turn makes the programs easer to be maintained.

2.2.2 Disadvantages

We have introduced Knuth's literate programming as well as its advantages. It turns out to be a good approach to produce better documentation and to improve the quality of software, but literate programming has not become a mainstream technique in software development yet. The reasons for this reside in the fact that writing literate programs requires additional time in comparison to writing "illiterate" programs and the limitation of language-dependence.

Time Overhead

There are following several issues that contribute to the time overhead. Literate programmers need longer time to learn before staring to work than traditional programmers do. Besides specific programming language and compare to Javadoc and TwinText, which are Source Code Documentation Tools rather than literate programming tools, they have to learn to install and configure the set of applications that support literate programming. Additionally, the harder part is learning how to properly write literate programs. So the longer learning curve of literate programming is a challenge to the beginners' patience.

Literate programming forces programmers to develop software applications using a completely different perspective, where the developers should first make their thoughts clear to everybody. In order to fulfil this requirement, only the ability of programming is obviously insufficient. It is because there are too many choices of expression way, order, and factoring extend to choose that literate programmers have to sacrifice time for the best.

In addition to programming errors, two new types of errors are introduced by this technique: WEB structural errors and formatting errors. The former are those caused by the incorrect use of the WEB's own language required to define the structure of a program. Since both Weave and Tangle routines use such structure as an input, this kind of error can the be propagated into programming and formatting language errors. Formatting errors are those caused by the misuse of formatting language. Similarly, these errors could affect other parts of a program.

For literate programmers, there is only one way to obtain the executable program. They have to run the Tangle routine over the WEB file first and then compile the output. If there exist any programming errors, they can not be found until executable program is built. In order to correct them, developers have to go back to the WEB file, make changes, then run the Tangle, and compile the output again.

Language-dependence

The first published literate programming environment is WEB [24], introduced by Donald Knuth in 1981; this system uses TEX as the document formatting language and PASCAL as the programming language. It is true that as long as a person knows both of the underlying languages, there is no trick at all to learning WEB, but what does it mean for those who do not know these two languages or for the circumstance that the underling languages do not suit for the programming of the target project?

In the section "The WEB System" of his Computer Journal article, Knuth addressed that the same principles would apply equally well if other languages were substituted: instead of TEX one could use a language like Scribe or Troff; instead of PASCAL, one could use ADA, ALGOL, LISP, COBOL, FORTRAN, APL, C, or even assembly language. However, all the literate programming systems derived from WEB depend on one or both underlying languages. CWEB is created by Donald Knuth and Silvio Levy as a follow up to Knuth's WEB, using the C programming language instead of PASCAL. OCAMAWEB is a CWEB like literate programming tool, which is a combination of the MATLAB [8] language and LATEX. Although noweb, FunnelWeb, and FWEB, can work with multiple programming languages, they still depends on their document formatting language respectively.

2.3 Summary

As a software application development approach in mainstream, object-oriented programming improve the quality of program, which includes reliability, modularity, compatibility, portability, and efficiency. The reason for this resides in the object-oriented techniques, such as class, object, inheritance, polymorphism, and abstraction. Its improvement on documentation and design approach as well as supporting tool is not satisfying. On the other hand, literate programming could produce significantly better documentation and improve the quality of software. However, its inevitable time overhead and language-dependence prevent literate programming from being a mainstream technique in software development.

Chapter 3

Object-Oriented Literate Programming

The previous chapters explored the the goals of OOLP and related research. In this chapter, we take a closer look at OOLP from the perspectives of its key aspects and supporting tools.

3.1 Overview

Nothing concerning OOLP is intrinsically new; what we have done is combined a number of ideas that have been in the field for a time. All of these techniques have their own advantages. By applying them systematically in a slightly new way, we propose a new programming paradigm — OOLP, which is anatomized in the following sections.

The practitioner of OOLP can be regarded as an essayist whose main tasks are to break the whole program into little pieces and to order or reorder them for pursuing the best suited to explaining what this software is doing. Every algorithm, even a single variable, is discussed properly in its natural place. In this way, the program and its documentation, including diagrams, are always consistent with each other. On the other hand, it still can be viewed as a collection of loosely connected objects, each of which is responsible for a certain specific task, which is a natural way for human beings to cognise the world. Therefore, this kind of *works of literature* is characterized by readability, reusability, flexibility, and maintainability.

Class diagrams are used in nearly all object-oriented analysis and design methods today. They can present readers a clear and intuitive view of the system structure. All existing literate programming tools would require developers to draw them by hand. In OOLP, Spark allows them (see Figure 3.1) to be generated automatically and inserted around the corresponding code part. Such automatic feature of Spark not only lightens developers' workload, but also ensures the consistence of class diagrams with the program code.

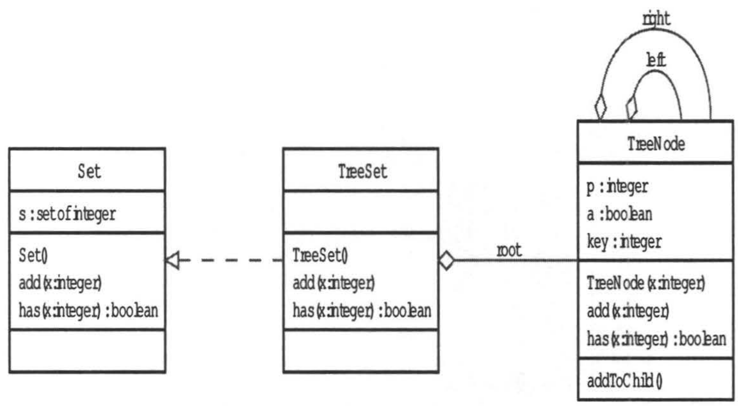


Figure 3.1: An example of automatic class diagram.

Like WEB, the programming environment of OOLP itself is chiefly a combination of two other languages: (1) a documentation formatting language and (2) a programming language. The difference is that programmers can choose their favorite or most suitable combination of these two kinds of languages. The main point is to let the practitioners of OOLP enjoy the power of the inherently bilingual tool, and get rid of the limitation of language-dependence.

In OOLP, the documentation formatting language provides tools to explain the local structure of documentation parts and to build the documentation that describes the program clearly and that facilitates program maintenance, while the programming language makes it possible to specify the algorithms formally and to obtain a machine-executable program. In addition, the supporting tool, Spark, is responsible

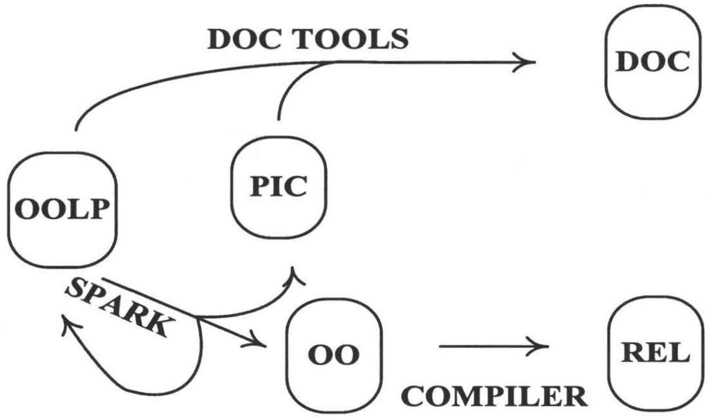


Figure 3.2: Overview of OOLP environment

for reconstructing compiler-acceptable code files, inserting continuous program code back to documentation, and building graphical notations.

3.2 Document Formatting Languages

Since OOLP dose not fix on any specific document formatting language, a programmer can choose anyone from the popular text based document generation systems, such as AsciiDoc [2], LaTeX, and DocBook [45]. The main point is that the target source file can be edited by arbitrary text editor platform-independently and that Spark can parse and process the target source file.

Let us look at this process in slightly more detail. Suppose AsciiDoc is used as the document formatting language and we have written an OOLP program and put it into a computer text file called EXAMPLE.TXT. The concrete syntax of AsciiDoc can be found in Appendix E. To generate hardcopy documentation for the program, we can run *asciidoc.py*, which is an executable program that takes the file EXAMPLE.TXT as input and produces another file as output. By setting different command line parameters, we can ask AsciiDoc to produce several predefined back end outputs, including DocBook, HTML, LinuxDoc, and LaTeX. Take the LaTeX output for example, after running the following command, we can have a file EXAMPLE.TEX as output.

asciidoc.py –unsafe –backend=latex EXAMPLE.TXT

Then we run the LaTeX processor, which takes EXAMPLE.TEX as input and produces EXAMPLE.PDF as output.

pdflatex EXAMPLE.TEX

By default, AsciiDoc produces plain HTML 4.01 file. We can simply run the following command,

asciidoc.py EXAMPLE.TXT

Then, we will get a file named EXAMPLE.HTML.

To use AsciiDoc, we need to setup the environment first (see Appendix A).

The process is the same for other document formatting languages, but the features of OOLP are denoted a little differently. For example, a piece of program code is listed in AsciiDoc as following:

```
-------------------------------
feature {ANY}
        STAFF...match (id: INTEGER): BOOLEAN is
            require
                id >= 0
            do
                Result := id = number
            end
-------------------------------
```

In DocBook, the same code goes as following:

```
<programlisting>
feature {ANY}
        STAFF...match (id: INTEGER): BOOLEAN is
            require
                id >= 0
            do
                Result := id = number
            end
</programlisting>
```

In LaTeX, it is listed as following:

```
\begin{verbatim}
feature {ANY}
        STAFF...match (id: INTEGER): BOOLEAN is
            require
                id >= 0
            do
                Result := id = number
            end
\end {verbatim}
```

Graphic notations are included in different ways too. For example, in AsciiDoc, a picture is included as following:

```
image::hpta_transit_info.jpg[Object Model]
//$ HPTA_TRANSIT_INFO DATABASE FILE_DATABASE @VERTICAL
```

"hpta_transit_info.jpg" is the picture's name and followed by its attribute, "Object Model". "//" denotes a comment line, which will be omited by AsciiDoc compiler, but Spark considers it as a setting of the diagram: HPTA_TRANSIT_INFO, DATABASE and FILE_DATABASE are explained as the classes included in this diagram; @VERTICAL means that the diagram must be drawn vertically. Other settings are discussed in Charter 5.

In DocBook, the same picture is included as following:

```
<figure><title>Object Model</title>
<graphic fileref="hpta_transit_info.jpg"></graphic>
<!--$ HPTA_TRANSIT_INFO DATABASE FILE_DATABASE @VERTICAL -->
</figure>
```

In LaTeX, it is included as following:

```
\includegraphics[width=100mm, height=65mm]{hpta_transit_info.jpg}
%$ HPTA_TRANSIT_INFO DATABASE FILE_DATABASE @VERTICAL
```

3.3 Programming Languages

In OOLP, programmers can also choose their programming language from multiple popular candidates, such as Java, C#, Eiffel, Lime, and C++. In principle, any programming language, like PASCAL, Basic, and even assembly language, is eligible for being such a candidate, but in this paper, we only focus on object-oriented programming languages.

Since in literate programming, the continuous program written in certain programming language has been broken into sections and ordered best for explaining to human beings, the traditional process of "compile, load, and go" has been slightly lengthened to "reassemble, compile, load, and go".

3.4 Spark

Spark consists of two parts: front end and back end. The front end takes an OOLP program as input and produces a number of program source code files as well as one graphic notation script file; the back end takes the graphic notation script file as input and produces a number of DOT files, which are used to feed GraphViz. GraphViz generates all the diagram files upon the request (see Figure 3.3). This structure decouples the programming language parsing from the algorithm of diagram layout so that different mechanisms can be changed transparently from each other.

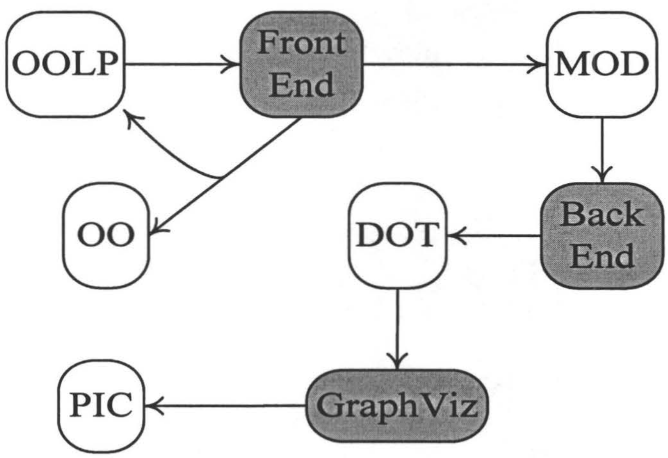


Figure 3.3: The Workflow of Spark

Two issues contribute to the fact that Spark gets rid of the limitation of the language-dependence in all existing literate programming tools. First, Spark focuses only on the code blocks and picture blocks and considers the other parts in OOLP programs as comments. In other words, as long as the document formatting language can work well by itself, Spark can accept it. Second, by providing different front ends, Spark can be easily extent to adapt to various programming languages (see Appendix D).

It is because Spark parses program code partially that it can help to debug the program. Spark generates graphic notation automatically. This not only lightens the workload of developers, but also ensures the consistence of diagrams with code. The usage of Spark is simple, i.e. the front end followed by the OOLP program file. For example, the front end for Eiffel is chose and the program is still EXAMPLE.TXT, then the command is:

```
perl sparkf-eiffel.pl EXAMPLE.TXT
```

3.5 Editors

All the supporting tools are independent software and can be either embedded into any extensible edit platform as plug-ins or called under OS shells, So there is no specific requirement for its editor. In this paper, as an example we choose VIM, which is a highly configurable text editor built to enable efficient text editing platform-independently. In what it follows we will show how to build a integrated development environment (IDE) by using the supporting tools as well as VIM.

```
vim EXAMPLE.TXT
```

To make the work easier, we can define a new command for VIM as following:

```
:command Spark :!perl sparkf.pl %
```

Then as long as finishing editing the OOLP program in VIM, we can switch to the command mode and input the new command set above as following:

```
:Spark
```

Whenever the command, Spark, is invoked, it begins to parse the current source file, and then both program code files and graphic notation files have been generated immediately if no errors. In this way, we can call the compiler to compile the source code and use other document formatting language tools to produce the consistent document.

3.6 Reference Developing Process

Basing the existing object-oriented approaches such as RDD, OMT, BON, Catalysis and VDM++ (see Section 2.1.5), we propose a reference developing process, of which each step is discussed in details in the rest of this section.

3.6.1 Analysis

In object-oriented software development, this phase takes the input of a fuzzy, minimal, possibly inconsistent target specification and produces the output of a understanding, complete, consistent description of essential characteristics and behavior. The final product, object, distinguishes object-oriented analysis from other approaches, such as structured analysis and Jackson's method [12].

Creating a Dictionary

The correctness of understanding the main terms used in the requirements is the key to get the correct model of the target system, so the dictionary must be as detailed and rigorous as possible. The potential classes and types identified in the dictionary could then form the basis of a class diagram, whereas the potential operations might be described as use cases. This idea comes from VDM++.

Identifying Classes

Object-oriented software consists of classes, which describes a group of objects with similar properties, common behavior, common relationships to other objects. So the main task of this step is to find out all classes from the dictionary constructed and keep the number of entities in the initial model as small as possible at the same time. The principles are listed as following:

- Omit those nouns, if they are irrelevant with the purpose of the system.
- Model those nouns as attributes, if they have only trivial functionality.
- Create an overall class to represent the entire system so that the precise relationships between the different classes and their associations can be expressed there.

- Whenever an association is introduced consider its multiplicity and give it a role name.
- Try to keep encapsulation by the modifiers such as private and protected.
- Document important properties or constraints as invariants.

Sketching Operations

An operation is a function or transformation that may be applied to or by objects in a class. The aim of this step is to try to describe all the operations listed in the dictionary with signature (parameters and result) and formal specification (pre- and postcondition). Then, assign them to the classes identified respectively. This idea comes from BON, Catalysis, and VDM++.

Constructing Initial Model

An object model captures the static structure of a system by showing the objects in the system, relationships between the objects and the attributes and operations that characterize each class of objects. This model provides an intuitive graphic representation of a system and is valuable for communicating with customers. This idea comes from OMT and VDM++; Spark supports the automatic generation of this model.

3.6.2 Design

During analysis, the focus is on what needs to be done. During design, decisions are made about how the problem will be solved better. This goal can be approached more efficiently by employing the successful experience such as existing business models and design patterns.

Consulting Existing Business Models

A business model is the object model that has been employed successfully in a certain actual project. Such well-found business model can bring us not only the speed of development, but also the quality of software product, after all it have passed the enough arguments and testing. So as long as a business model can meeting the requirements of the target system entirely or partially, we should replace the initial model generated in analysis with it entirely or partially. This idea comes from Catalysis.

Applying Design Patterns

We can not expect to find out everything in our library; in more cases, we need construct a new one. Then, what kind of design is good design? This question is ignored by most existing object-oriented approaches. We recommend design patterns, which can provide the answer (see Section 2.1.6). In this step, many new classes may be introduced into the current model, but this is we have to pay for the design with better flexibility, extensibility and reliability.

3.6.3 Implementing

The goal of Analysis and design is to bridge the gap between the real world and computer domain; the goal of this portion is discuss the specific details for implementing a system using programming languages. By the aid of Spark and literate tools, one can debug the program and view the document freely.

3.6.4 Testing

Testing is the process used to measure the quality of developed computer software. Since software is developed by human beings, it is inevitable that there exist some errors. Therefore, testing must be conducted for every computer software.

In order to cover the correctness, completeness, security, reliability, usability and portability of software, one needs to perform the following tests:

- **Usability testing**, which tries to find faults in the user interface design of the system.
- **Unit testing**, which tries to find faults in participating objects.
- **Integration testing**, which is the activity of finding faults when testing the individually tested components together.

- **System testing**, which tests the entire system.
  - **Functional testing**, which tests the requirements.
  - **Performance testing**, which checks the design goals.
  - **Acceptance testing**, which check the system against the project agreement and is done by the customer.

3.7 Summary

This chapter has introduced the following major features of OOLP and Spark:

- Including program code and graphic notations in various document formatting languages.
- Setting graphic notations
- Setting up the developing environment.
- Using Spark to generate graphic notation files and program code.
- Constructing OOLP IDE with VIM.
- A reference developing process

Chapter 4

Transit Information System Case Study

In this chapter, by an example, Transit Information System, we demonstrate how to use the technique of OOLP in developing software. The source code is listed in Appendix B. The rest of this chapter that follow is the actual output of an OOLP program file.

4.1 Transit Information System

4.1.1 Requirements

In this project, we are asked to develop an information system for a local train and bus service. Our customer, HPTA (Happy Passenger Transit Authority), has no clear picture what it should do, except to increase customer satisfaction and make traveling more attractive. All the information we have goes as follows:

- It will be used by passengers as well as by HPTA staff.
- Selected staff members would be allowed to update the information.
- Passengers should be able to enter their start and destination, a desired time, and get a bunch of possible connections.
- Connections can be direct or with changing busses or trains.

- For each bus and train station, the information like opening hours and accessibility is maintained.
- Users can browse a list of all bus and train routes or check the details of a certain route..
- Some bus stops and train stops are conjoint, but some not.
- Trains have two-digit numbers and busses have three-digit numbers.
- Connections between trains and busses must have at least five minutes for the change.

For simplicity, we assume that detours and delays do not occur, stops are never skipped.

4.1.2 An Overview

The following picture (Figure 4.1) is the object model of transit information system. As the root class, HPTA_TRANSIT_INFO controls the whole system from the beginning to the end. Class DATABASE is a deferred class, whose subclasses, such as class FILE_DATABASE, are responsible for maintaining system data. Class CONNECTION_FINDER is also a deferred class, whose subclasses, such as class PRIME_FINDER, are responsible for finding the possible connections.

The purpose of the application is to maintain the system information, including local train or bus service and the status of staffs, and provide users current public transit service information, including possible connections and routes.

4.1.3 Dictionary

To understand the main terms used in the requirements, we create a dictionary as following:

- *passenger*: a person, who want to get his or her destination by bus or train.
- *staff*: a person, who works for HPTA.

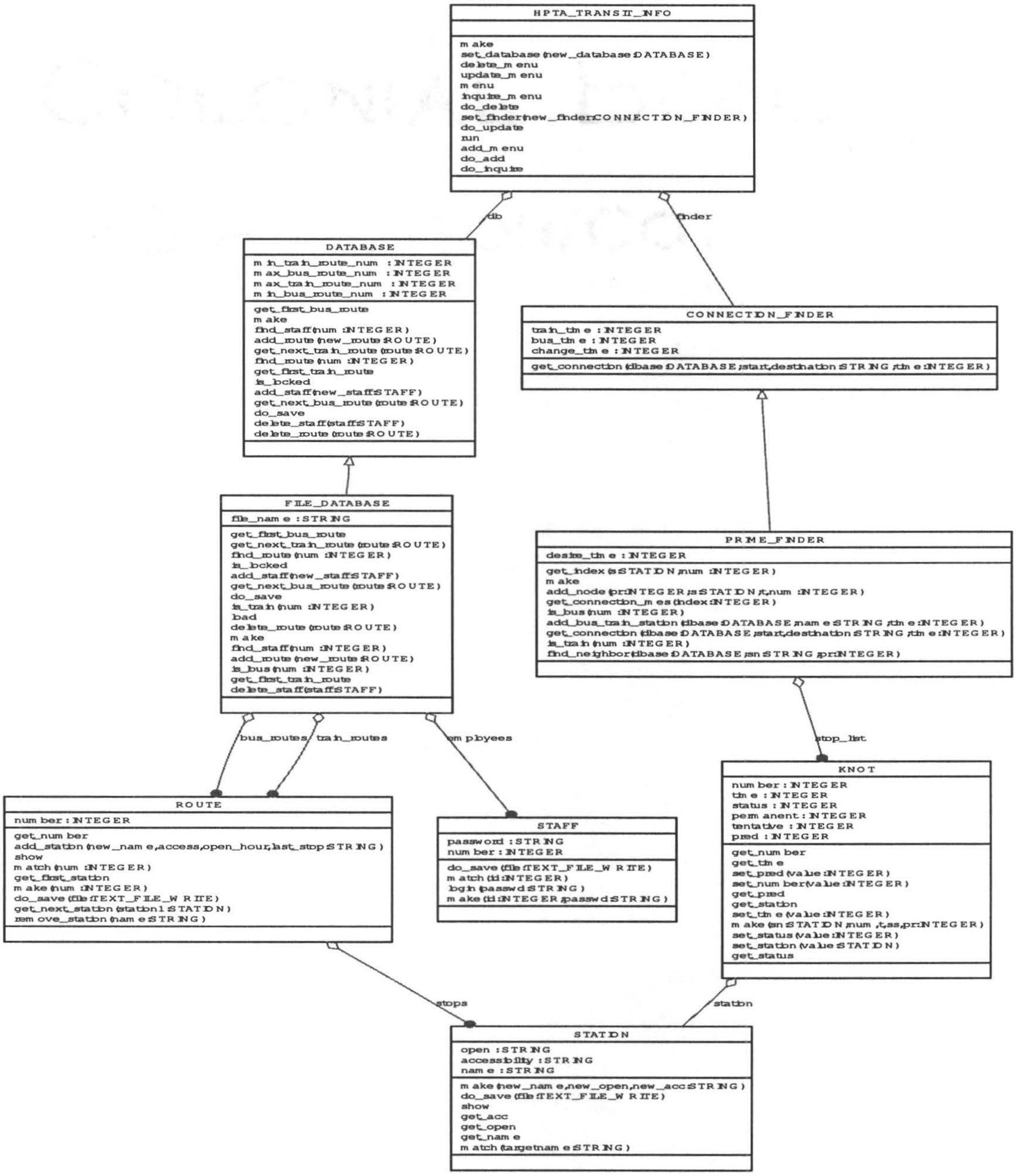


Figure 4.1: Object model of transit information system.

- *start*: a station, where a passenger begin his or her journey.
- *destination*: a station to which a passenger is going or directed.
- *desired time*: an interval, within which one want get to the destination from the start.
- *connection*: a sequence of stations.
- *bus*: a long motor vehicle for carrying passengers, usually along a fixed route.
- *train*: a series of connected railroad cars pulled or pushed by one or more locomotives.
- *route*: a course for buses or trains to travel from one station to another.
- *opening hour*: a time, at which the first vehicle departs.
- *accessibility*: a description of the running status of a station.
- *update*: a change of system information.
- *browse*: a display of the information of all routes.
- *check*: a detail show of a certain route information.

4.1.4 Identifying Class

The following classes are identified from the requirements.

```
class HPTA_TRANSIT_INFO
end
```

Class HPTA_TRANSIT_INFO is identified as a class of the entire system.

```
class STAFF
feature {NONE}
	number: INTEGER
	password: STRING
end
```

STAFF is a class with attributes employee number and password. The requirements state that selected staff members would be allowed to update the system.

```
class STATION
feature {NONE}
        name: STRING
        open: STRING
        accessibility: STRING
end
```

STATION is a class with attributes name, opening hour, and accessibility.

```
class ROUTE
feature {NONE}
      number: INTEGER
      stops: LINKED_LIST [STATION]
end
```

ROUTE is a class with attributes station list and route number.

4.1.5 Identifying Operations

All three operations listed in the dictionary belong naturally in the class HPTA_TRANSIT_INFO, because they are dependent on the interface of the system.

- login should belong in class STAFF, because it keep the secret of a certain staff.

4.1.6 Consulting The Library of Model

There is no suitable business model in our existing library, so we have to build this system from the beginning.

4.1.7 Applying Design Patterns

According to the requirements, our application needs to keep all system information and to calculate possible connections. There exist so many different methods for these two tasks. Hence, we apply the strategy design pattern. We declare two deferred classes

```
deferred class DATABASE
end
```

and

```
deferred class CONNECTION_FINDER
end
```

Then, we define two private members in class HPTA_TRANSIT_INFO denoted by the class name followed by three dots as following:

```
feature {NONE}
        HPTA_TRANSIT_INFO... db: DATABASE
        finder: CONNECTION_FINDER
```

i.e.

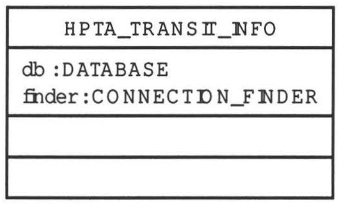


Figure 4.2: The attributes of class HPTA_TRANSIT_INFO

In this way, we can add new algorithms easily and even change mechanisms at runtime with the following private methods:

```
feature {NONE}
        HPTA_TRANSIT_INFO... set_finder(new_finder: CONNECTION_FINDER) is
            require
                new_finder /= Void
            do
                finder := new_finder
            ensure
                finder = new_finder
            end
```

and

```
feature {NONE}
        HPTA_TRANSIT_INFO... set_database(new_database: DATABASE) is
            require
                new_database /= Void
            do
                db := new_database
            ensure
                db = new_database
            end
```

Their preconditions require that the new comers are not invalid and their postconditions ensure that the private member db and finder are set correctly.

Class CONNECTION_FINDER describes the interface that is common to all concrete mechanisms as following:

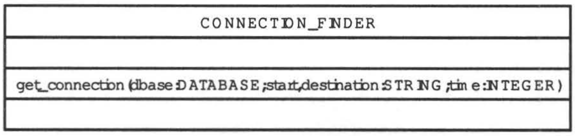


Figure 4.3: Class CONNECTION_FINDER

```
feature {HPTA_TRANSIT_INFO}
    CONNECTION_FINDER...
    get_connection(dbase: DATABASE; start, destination: STRING; time: INTEGER): STRING is
            require
                start /= Void
                destination /= Void
                time >= 0
                dbase /= Void
            deferred
            end
```

Class DATABASE describes the interface that is common to all concrete data maintain mechanisms as following:

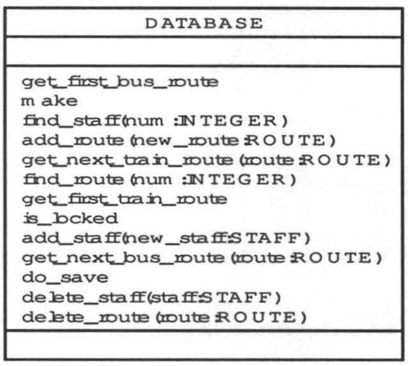


Figure 4.4: Class database

```
feature {HPTA_TRANSIT_INFO, CONNECTION_FINDER}
        DATABASE... get_first_bus_route: ROUTE is
                deferred
                end
```

This method can return the first bus route object and is used by class HPTA_TRANSIT_INFO and class CONNECTION_FINDER only. Together with the following method, its clients can browse all bus routes one by one.

```
feature {HPTA_TRANSIT_INFO, CONNECTION_FINDER}
        DATABASE... get_next_bus_route(route: ROUTE): ROUTE is
            deferred
            end
```

Similarly, we can browse all train routes by the following two methods:

```
feature {HPTA_TRANSIT_INFO, CONNECTION_FINDER}
        DATABASE... get_first_train_route: ROUTE is
            deferred
            end
```

and

```
feature {HPTA_TRANSIT_INFO, CONNECTION_FINDER}
        DATABASE... get_next_train_route(route: ROUTE): ROUTE is
            deferred
            end
```

Browsing all staff information is not necessary, but we need to find given staff object by the following method.

```
feature {HPTA_TRANSIT_INFO, CONNECTION_FINDER}
        DATABASE... find_staff (num: INTEGER): STAFF is
            require
                num > 0
            deferred
            end
```

This method can return an STAFF object, whose employee number equals to the parameter *num*. It is because all employee number start from 1 that the precondition is added.

For convenience, we also provide a route finding method as follows:

```
feature {HPTA_TRANSIT_INFO, CONNECTION_FINDER}
        DATABASE... find_route (num: INTEGER): ROUTE is
            require
                num >= min_train_route_num
                num <= max_bus_route_num
            deferred
            end
```

The following method is the creation of class DATABASE and invoked by class HPTA_TRANSIT_INFO only.

```
feature {HPTA_TRANSIT_INFO}
        DATABASE... make is
            deferred
            end
```

In order to update system information, class DATABASE also requires the interfaces of adding and deleting as following:

```
feature {HPTA_TRANSIT_INFO}
        DATABASE... add_route(new_route: ROUTE) is
            require
                new_route /= Void
            deferred
            end
```

and

```
feature {HPTA_TRANSIT_INFO}
        DATABASE... delete_route (route: ROUTE) is
            require
                route /= Void
            deferred
            end
```

These two methods can add or delete a certain route to or from this system respectively and is called by class HPTA_TRANSIT_INFO only.

Similarly, class HPTA_TRANSIT_INFO also can add or delete a certain staff by the following two methods:

```
feature {HPTA_TRANSIT_INFO}
        DATABASE... add_staff(new_staff: STAFF) is
            require
                new_staff /= Void
            deferred
            end
```

and

```
feature {HPTA_TRANSIT_INFO}
        DATABASE... delete_staff (staff: STAFF) is
            require
                staff /= Void
            deferred
            end
```

As long as some system information is updated, DATABASE object must be informed to save the change by the following method.

```
feature {HPTA_TRANSIT_INFO}
        DATABASE... do_save is
            deferred
            end
```

According to the requirements, only selected members can update the system. We define that when the database is locked, only the user, who knows both employee number and password, can conduct an update.

```
feature {HPTA_TRANSIT_INFO}
        DATABASE... is_locked: BOOLEAN is
            deferred
            end
```

The subclasses of these two deferred classes implement each concrete behavior mentioned above.

The following four constants are used to point out the bound of route number

```
feature {NONE}
        DATABASE... max_bus_route_num: INTEGER is 999
        min_bus_route_num: INTEGER is 100
        max_train_route_num: INTEGER is 99
        min_train_route_num: INTEGER is 10
```

4.1.8 Algorithms Design

File Database

For simplicity, we save the system information in a file named "sys_info.txt". So we define a subclass of class DATABASE, FILE_DATABASE as following:

```
class FILE_DATABASE
inherit DATABASE
feature {NONE}
     file_name: STRING is "sys_info.txt"
end
```

i.e.

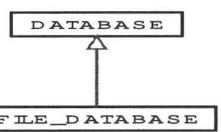


Figure 4.5: The hirarchy of databases

class FILE_DATABASE keep bus routes, train routes and staffs with LINKED_LIST as following:

```
feature {NONE}
        FILE_DATABASE... train_routes: LINKED_LIST [ROUTE]
        bus_routes:     LINKED_LIST [ROUTE]
        employees:      LINKED_LIST [STAFF]
```

now, class FILE_DATABASE becomes:

The creation of FILE_DATABASE is method make

```
create FILE_DATABASE... make
```

The main task of make is to initialize this three list

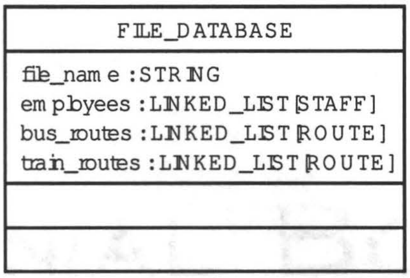


Figure 4.6: The attributes of FILE_DATABASE

```
feature {HPTA_TRANSIT_INFO}
        FILE_DATABASE... make is
            do
                create employees.make
                create bus_routes.make
                create train_routes.make
                load
            ensure
                employees /= Void
                bus_routes /= Void
                train_routes /= Void
            end
```

and to load the system information for that file:

```
feature {NONE}
   FILE_DATABASE... load is
        local
            input_string : STRING
            text_file_read: TEXT_FILE_READ
            text_file_write: TEXT_FILE_WRITE
            split: ARRAY[STRING]
            new_staff: STAFF
            route: ROUTE
        do
            create text_file_read.connect_to(file_name)
            if text_file_read.is_connected then
                from text_file_read.read_line
                until text_file_read.end_of_input
                loop
                    if text_file_read.last_string.upper = 1 then
                        inspect text_file_read.last_string.first.to_upper
                        when 'S' then
                            text_file_read.read_line
                            input_string := text_file_read.last_string.twin
                            split := input_string.split
                            create new_staff.make (split.first.to_integer, split.last)
                            employees.add_last(new_staff)
                        when 'B', 'T' then
                            text_file_read.read_line
                            input_string := text_file_read.last_string.twin
                            split := input_string.split
                            route := find_route(split.item(4).to_integer)
                            if route = Void then
                                create route.make(split.item(4).to_integer)
```

```
                    route.add_station(split.first, split.item(2), split.item(3), split.last)
                    if split.item(4).to_integer > max_train_route_num then
                        bus_routes.add_last(route)
                    else
                        train_routes.add_last(route)
                    end
                else
                    route.add_station(split.first, split.item(2), split.item(3), split.last)
                end
            else
            end
        end
        text_file_read.read_line
      end
      text_file_read.disconnect
    else
      create text_file_write.connect_to(file_name)
      if text_file_write.is_connected then
          text_file_write.disconnect
      end
    end
  end
```

By the following method, one can get the specific route object.

```
feature {HPTA_TRANSIT_INFO, CONNECTION_FINDER}
        FILE_DATABASE... find_route (num: INTEGER): ROUTE is
            local
                i: INTEGER
                route: ROUTE
            do
                if num > max_train_route_num then
                    from i := bus_routes.lower
                    until i > bus_routes.upper or else bus_routes.item(i).match(num)
                    loop
                        i := i+1
                    end
                    if i <= bus_routes.upper then
                        route := bus_routes.item(i)
                    end
                else
                    from i := train_routes.lower
                    until i > train_routes.upper or else train_routes.item(i).match(num)
                    loop
                        i := i+1
                    end
                    if i <= train_routes.upper then
                        route := train_routes.item(i)
                    end
                end
                Result := route
            end
```

Similarly, using the following method, one can get the staff with such employee number:

```
feature {HPTA_TRANSIT_INFO, CONNECTION_FINDER}
        FILE_DATABASE... find_staff (num: INTEGER): STAFF is
            local
                i: INTEGER
                staff: STAFF
            do
                from i := employees.lower
                until i > employees.upper or else employees.item(i).match(num)
                loop
```

```
            i := i+1
        end
        if i <= employees.upper then
            staff := employees.item(i)
        end
        Result := staff
    end
```

By the following four methods, one can browse all train routes and bus routes:

```
feature {HPTA_TRANSIT_INFO, CONNECTION_FINDER}
    FILE_DATABASE... get_first_bus_route: ROUTE is
        local
            route: ROUTE
        do
            if not bus_routes.is_empty then
                route := bus_routes.first
            end
            Result := route
        end
    get_next_bus_route(route: ROUTE): ROUTE is
        require
            bus_routes.index_of(route) > 0
        local
            next_route: ROUTE
        do
            if bus_routes.index_of(route) < bus_routes.upper then
                next_route := bus_routes.item(bus_routes.index_of(route)+1)
            end
            Result := next_route
        end
    get_first_train_route: ROUTE is
        local
            route: ROUTE
        do
            if not train_routes.is_empty then
                route := train_routes.first
            end
            Result := route
        end
    get_next_train_route(route: ROUTE): ROUTE is
        require
            train_routes.index_of(route) > 0
        local
            next_route: ROUTE
        do
            if train_routes.index_of(route) < train_routes.upper then
                next_route := train_routes.item(train_routes.index_of(route)+1)
            end
            Result := next_route
        end
```

By the following method, HPTA_TRANSIT_INFO object can add an arbitrary route to this database

```
feature {HPTA_TRANSIT_INFO}
    FILE_DATABASE... add_route(new_route: ROUTE) is
        do
            if is_bus(new_route.get_number) then
                bus_routes.add_last(new_route)
            elseif is_train(new_route.get_number) then
                train_routes.add_last(new_route)
            end
        end
```

By the following method, HPTA_TRANSIT_INFO object can add a staff to this database

```
feature {HPTA_TRANSIT_INFO}
	FILE_DATABASE... add_staff(new_staff: STAFF) is
		do
			employees.add_last(new_staff)
		end
```

By the following method, HPTA_TRANSIT_INFO object can remove an arbitrary route from this database

```
feature {HPTA_TRANSIT_INFO}
	FILE_DATABASE... delete_route (route: ROUTE) is
		do
			if is_bus(route.get_number) then
				bus_routes.remove(bus_routes.index_of(route))
			elseif is_train(route.get_number) then
				train_routes.remove(train_routes.index_of(route))
			end
		end
```

By the following method, HPTA_TRANSIT_INFO object can remove a staff from this database

```
feature {HPTA_TRANSIT_INFO}
	FILE_DATABASE... delete_staff (staff: STAFF) is
		do
			employees.remove(employees.index_of(staff))
		end
```

In FILE_DATABASE, as long as employees is not empty, this database is locked, which means you have to log in before updating.

```
feature {HPTA_TRANSIT_INFO}
	FILE_DATABASE... is_locked: BOOLEAN is
		do
			Result := not employees.is_empty
		end
```

Whenever the database is changed, it have to save the new data to the specific file by the following method:

```
feature {HPTA_TRANSIT_INFO}
	FILE_DATABASE... do_save is
		local
			file_2_write: TEXT_FILE_WRITE
			i: INTEGER
		do
			create file_2_write.connect_to(file_name)
			if file_2_write.is_connected then
				from i := employees.lower
				until i > employees.upper
				loop
					employees.item(i) .do_save(file_2_write)
					i := i + 1
				end
				from i := bus_routes.lower
```

```
            until i > bus_routes.upper
            loop
                bus_routes.item(i) .do_save(file_2_write)
                i := i + 1
            end
            from i := train_routes.lower
            until i > train_routes.upper
            loop
                train_routes.item(i) .do_save(file_2_write)
                i := i + 1
            end
            file_2_write.disconnect
        else
            io.put_string("Update_database_failed!%N")
        end
    end
```

For convenience, we define the following two methods to tell if the current route is bus or train route:

```
feature {NONE}
    FILE_DATABASE... is_train(num: INTEGER): BOOLEAN is
        do
            Result := num >= min_train_route_num and num <= max_train_route_num
        end
    is_bus(num: INTEGER): BOOLEAN is
        do
            Result := num >= min_bus_route_num and num <= max_bus_route_num
        end
```

Conditional Shortest Path

According to the requirement that connections between trains and busses must have at least five minutes for the change, we have to consider bus station and train station as two different stations even they share the same name. In addition, we define a constant change_time in class CONNECTION_FINDER, whose subclasses need it.

```
feature {NONE}
    CONNECTION_FINDER... change_time: INTEGER is 5
```

For convenience, we assume that a bus needs 2 minutes to get to the second stop and a train needs only 1 minute. So we also define the following two members in class CONNECTION_FINDER.

```
feature {NONE}
    CONNECTION_FINDER... train_time: INTEGER is 1
    bus_time: INTEGER is 2
```

PRIME_FINDER is one of the subclasses of CONNECTION_FINDER

```
inherit
    PRIME_FINDER... CONNECTION_FINDER
```

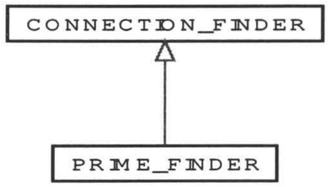


Figure 4.7: The hierarchy of class CONNECTION_FINDER

i.e.

Our first algorithm, PRIME_FINDER, is that starting from the start stations, including both bus station and train station, we search for all direct neighbors one after another and calculate their time respectively. In this way, as long as we found the destination as the next neighbor or no more new neighbors before get to the destination, our searching work is done.

To implement this algorithm, we declare list in class PRIME_FINDER

```
feature {NONE}
    PRIME_FINDER... stop_list: LINKED_LIST [KNOT]
```

Every node of this list record the following information:

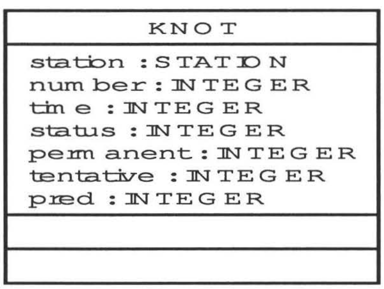


Figure 4.8: The attributes of class KNOT

```
feature {NONE}
    KNOT... station: STATION
```

Form the start down to the destination, as long as the station is found as a valid neighbor, it will be set in a KNOT object by the following method.

```
feature {PRIME_FINDER}
    KNOT... set_station(value: STATION) is
        do

            station := value

        end
```

Of course, class KNOT requires PRIME_FINDER object give a non Void value.

```
feature {PRIME_FINDER}
    KNOT... get_station: STATION is
        do
            Result := station
        end
```

After searching, PRIME_FINDER object can get the record of station by the above method.

```
feature {NONE}
    KNOT... number: INTEGER
```

The number of KNOT object keeps the route number of the station and is set by the following method:

```
feature {PRIME_FINDER}
    KNOT... set_number(value: INTEGER) is
        require
            value >= 0
            value <= 999
        do
            number := value
        end
```

According to the requirement that train route number is a two-digit number and bus route number is a three-digit number, we set a precondition like that for this method.

```
feature {PRIME_FINDER}
    KNOT... get_number: INTEGER is
        do
            Result := number
        end
```

The above method can tell PRIME_FINDER object the route, to which this station belongs.

```
feature {NONE}
    KNOT... time: INTEGER
```

Member time records the total time needed from start and is set by the following method

```
feature {PRIME_FINDER}
    KNOT... set_time(value: INTEGER) is
        require
            value >= 0
        do
            time := value
        end
```

The time of start node is 0 and the time of destination is desire time plus one, so here KNOT object requires a nonnegative number.

```
feature {PRIME_FINDER}
    KNOT... get_time: INTEGER is
        do
            Result := time
        end
```

The above method is used to provide time for PRIME_FINDE object.

```
feature {NONE}
    KNOT... pred: INTEGER
```

This member is used to record the index of last stop in this list. The *pred* of start is -1. That the *pred* of two destination are all -1 means that there is no possible connection between the start and the destination.

PRIME_FINDER object set this member by the following method:

```
feature {PRIME_FINDER}
    KNOT... set_pred(value: INTEGER) is
        do
            pred := value
        end
```

and get the value of this member by the following method:

```
feature {PRIME_FINDER}
    KNOT... get_pred: INTEGER is
        do
            Result := pred
        end
```

Then, how can we judge if this node should be check for new neighbors? we define the member status in class KNOT.

```
feature {NONE}
    KNOT... status: INTEGER
```

If there is no more new neighbors can be found for the current station, this member should be set as permanent, which is a constant of class KNOT;

```
feature {PRIME_FINDER}
    KNOT... permanent: INTEGER is 1
```

otherwise, member status should be set as tentative, which is another constant of class KNOT.

```
feature {PRIME_FINDER}
    KNOT... tentative: INTEGER is 0
```

This member can be set by the following method

```
feature {PRIME_FINDER}
    KNOT... set_status(value: INTEGER) is
        require
            value >= tentative
            value <= permanent
        do
            status := value
        end
```

and get by the following method

```
feature {PRIME_FINDER}
    KNOT... get_status: INTEGER is
        do
            Result := status
        end
```

Method make is the creation of class KNOT

```
creation {PRIME_FINDER}
    KNOT... make
```

and its main task is to initialize this object with the given parameters as following:

```
feature {PRIME_FINDER}
    KNOT... make(sn: STATION; num, t, ss, pr: INTEGER) is
        do
            set_station(sn)
            set_number(num)
            set_time(t)
            set_status(ss)
            set_pred(pr)
        end
```

Every node is added into the list by the following method:

```
feature {NONE}
    PRIME_FINDER... add_node (pr: INTEGER; s: STATION; t, num: INTEGER) is
        require
            t >= 0
        local
            node: KNOT
        do
            create node.make(s, num, t, node.tentative, pr)
            if s = Void then
              node.set_status(node.permanent)
            end
            stop_list.add_last(node)
        end
```

If the station is Void, then the new node will be considered as dead.

The logic of possible connection finding is implemented mainly in the following method.

```
feature {HPTA_TRANSIT_INFO}
    PRIME_FINDER...
    get_connection(dbase: DATABASE; start, destination: STRING; time: INTEGER): STRING is
        require else
            stop_list.upper = 0
        local
            connection, cur_station: STRING
            node: KNOT
            i, monitor: INTEGER
            is_end, break: BOOLEAN
        do
            connection := ""
            desire_time := time
            add_bus_train_station(dbase, destination, desire_time+1)
            add_bus_train_station(dbase, start, 0)

            i := 3
            cur_station := start.twin

            from
            until is_end or else cur_station = Void
            loop
                monitor := stop_list.upper
                find_neighbor(dbase, cur_station, i)
                if monitor = stop_list.upper then
                    if stop_list.item(i) /= Void then
                        stop_list.item(i) .set_status(node.permanent)
                    end
                end
                is_end := True

                from
                until break or else i > stop_list.upper
                loop
                    if stop_list.item(i) /= Void then
                        node := stop_list.item(i)
                        if node.get_status = node.tentative and node.get_station /= Void then
                            cur_station := node.get_station.get_name
                            is_end := False
                            break := True
                        end
                    end
                    if not break then
                        i := i + 1
                    end
                end

                if break then
                    break := False
                end
            end

            connection := get_connection_mes(1)
            connection := connection + get_connection_mes(2)

            if connection.same_as("") then
                connection := "There_is_no_connection_from_your_start"
                        + "_to_your_destination_in_such_time."
            end

            Result := connection
        ensure
            Result /= Void
        end
```

The first parameter provides the source of data; the second and third parameters are the names of start station and destination station respectively; the last parameter is the desire time, which will be used to set the private member desire_time:

```
feature {NONE}
    PRIME_FINDER... desire_time: INTEGER
```

At the beginning of searching, we initialize the stop_list of a PRIME_FINDER object with four nodes, i.e. bus and train stations of destination followed by bus and train stations of start, using the following method:

```
feature {NONE}
    PRIME_FINDER... add_bus_train_station(dbase: DATABASE; name: STRING; time: INTEGER) is
        require
            name /= Void
            time >= 0
        local
            route: ROUTE
            station: STATION
            is_end: BOOLEAN
            num: INTEGER
        do
            route := dbase.get_first_bus_route
            from
            until is_end or route = Void
            loop
                station := route.get_first_station
                from
                until is_end or station = Void
                loop
                    if name.same_as(station.get_name) then
                        is_end := True
                    end
                    if not is_end then
                        station := route.get_next_station(station)
                    end
                end
                if not is_end then
                    route := dbase.get_next_bus_route(route)
                end
            end
            if not is_end then
              station := Void
            end
            if route /= Void then
                num := route.get_number
            else
                num := 0
            end
            add_node(-1, station, time, num)

            station := Void
            is_end := False
            route := dbase.get_first_train_route
            from
            until is_end or route = Void
            loop
                station := route.get_first_station
                from
                until is_end or station = Void
```

```
            loop
               if name.same_as(station.get_name) then
                  is_end := True
               end
               if not is_end then
                  station := route.get_next_station(station)
               end
            end
            if not is_end then
               route := dbase.get_next_train_route(route)
            end
         end
         if not is_end then
            station := Void
         end
         if route /= Void then
            num := route.get_number
         else
            num := 0
         end
         add_node(-1, station, time, num)
   end
```

Then from the bus station of start, we try to find its direct neighbor by the following method:

```
feature {NONE}
   PRIME_FINDER... find_neighbor(dbase: DATABASE; sn: STRING; pr: INTEGER) is
         require
            sn /= Void
         local
            cost, index, switch: INTEGER
            p_node, node: KNOT
            route: ROUTE
            station, last: STATION
            name: STRING
            break: BOOLEAN
         do
            from switch := 0
            until switch > 1
            loop
               if switch = 0 then
                  cost := bus_time
               else
                  cost := train_time
               end
               if pr >= stop_list.lower and pr <= stop_list.upper then
                  p_node := stop_list.item(pr)
               end
               if p_node /= Void then
                  if p_node.get_station /= Void then
                     if switch = 0 then
                        if is_train(p_node.get_number) then
                           cost := change_time + cost
                        end
                        route := dbase.get_first_bus_route
                     else
                        if is_bus(p_node.get_number) then
                           cost := change_time + cost
                        end
                        route := dbase.get_first_train_route
                     end
                     from
                     until route = Void
                     loop
                        station := route.get_first_station
```

```
            last := station
            from
            until station = Void or break
            loop
                name := station.get_name.twin
                if name /= Void and name.is_equal(sn) then
                    if not last.get_name.is_equal(name) then
                        index := get_index(last , route.get_number)
                        if index >= 0 then
                            node := stop_list.item(index)
                            if node.get_station /= Void then
                                if is_train(node.get_number) then
                                    if node.get_time > p_node.get_time + cost then
                                        node.set_pred(pr)
                                        node.set_time(p_node.get_time + cost)
                                        node.set_number(route.get_number)
                                    end
                                end
                            end
                        else
                            add_node(pr, last , p_node.get_time+cost , route.get_number)
                        end
                    end
                    last := route.get_next_station(station)
                    if last /= Void then
                        index := get_index(last , route.get_number)
                        if index >= 0 then
                            node := stop_list.item(index)
                            if node.get_station /= Void then
                                if is_train(node.get_number) then
                                    if node.get_time > p_node.get_time + cost then
                                        node.set_pred(pr)
                                        node.set_time(p_node.get_time + cost)
                                        node.set_number(route.get_number)
                                    end
                                end
                            end
                        else
                            add_node(pr, last , p_node.get_time+cost , route.get_number)
                        end
                    end
                    break := True
                else
                    last := station;
                    station := route.get_next_station(station)
                end
            end
            break := False
            if switch = 0 then
                route := dbase.get_next_bus_route(route)
            else
                route := dbase.get_next_train_route(route)
            end
          end
        end
      end
      switch := switch + 1
    end
  end
```

For convenience, we define the following two methods to tell if the current route is train or bus:

```
feature {NONE}
    PRIME_FINDER... is_train(num: INTEGER): BOOLEAN is
        do
```

```
        Result := num >= 10 and num <= 99
    end
```

and

```
feature {NONE}
    PRIME_FINDER... is_bus(num: INTEGER): BOOLEAN is
        do
            Result := num >= 100 and num <= 999
        end
```

The following method is used to get the index of a certain station in the list; if the target station is not in the list, -1 will be return.

```
feature {NONE}
    PRIME_FINDER... get_index(s: STATION; num: INTEGER): INTEGER is
        require
            s /= Void

        local
            ind, i: INTEGER
            node: KNOT
            name: STRING
        do
            ind := -1
            from  i := stop_list.lower
            until i > stop_list.upper
            loop
                node := stop_list.item(i)
                if node.get_station /= Void then
                    name := node.get_station.get_name
                    if name.is_equal(s.get_name) then
                      if is_bus(num) and is_bus(node.get_number) then
                            ind := i
                      elseif is_train(num) and is_train(node.get_number) then
                            ind := i
                      end
                    end
                end
                i := i + 1
            end
            Result := ind
        end
```

When the searching is done, we can get the information of possible connections by the following method:

```
feature {NONE}
    PRIME_FINDER... get_connection_mes(index: INTEGER): STRING is
        require
            index >= 0
        local
            node: KNOT
            mes: STRING
        do
            mes := ""
            node := stop_list.item(index)
            if node /= Void then
                if node.get_station /= Void then
                    if node.get_pred /= -1 and node.get_time <= desire_time then
                        mes := "_No." + node.get_number.to_string + "->"
                                + node.get_station.get_name + "_in_"
                                + node.get_time.to_string + "_minutes%N"
```

```
                node := stop_list.item(node.get_pred)
                from
                until node = Void or else node.get_station = Void or else node.get_pred = -1
                loop
                    mes := "-No." + node.get_number.to_string + "->"
                          + node.get_station.get_name + mes
                    node := stop_list.item(node.get_pred)
                end
                if node /= Void then
                   if node.get_station /= Void then
                      mes := "%N" + node.get_station.get_name + mes
                   end
                else
                   mes := ""
                end
             end
          end
       end
       Result := mes
    end
```

The creation of PRIME_FINDER is method make

```
creation {ANY} PRIME_FINDER...make
```

it is defined as following:

```
feature {HPTA_TRANSIT_INFO}
      PRIME_FINDER...make is
         do
            create stop_list.make
         ensure
            stop_list /= Void
         end
```

Now, let us talk about the root class HPTA_TRANSIT_INFO.

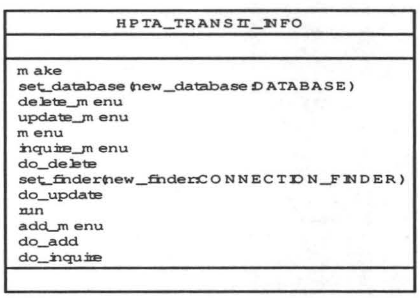


Figure 4.9: The methods of class HPTA_TRANSIT_INFO

The creation of class HPTA_TRANSIT_INFO is make

```
create HPTA_TRANSIT_INFO...make
```

Its main task is to initialize the database and connection finder, and then run the whole system:

```
feature {ANY}
        HPTA_TRANSIT_INFO...make is
            local
                prime_finder: PRIME_FINDER
                file_database: FILE_DATABASE
            do
                create file_database.make
                set_database(file_database)
                create prime_finder.make
                set_finder(prime_finder)
                run
            end
```

In order to increase customer satisfaction, we run the system by a series of menus

```
feature {NONE}
        HPTA_TRANSIT_INFO...run is
                do
                    from
                    until io.last_character.to_upper = 'Q'
                    loop
                        menu
                        io.read_character
                        io.put_new_line
                        inspect io.last_character.to_upper
                        when 'U' then do_update
                        when 'I' then do_inquire
                        else
                        end
                    end
                end
```

In order to use OS command, we let class HPTA_TRANSIT_INFO be a subclass of class SYSTEM, which is a predefined class in Eiffel.

```
inherit
     HPTA_TRANSIT_INFO...SYSTEM
```

Method menu is the main menu of the interface of this system and

```
feature {NONE}
        HPTA_TRANSIT_INFO...menu is
                do
                        execute_command_line("cls")
                        io.put_string("[
                              **************************************
                                          Welcome to HPTA
                              **************************************
                                  U Update System Information
                                  I Inquire about Transit Information
                                  Q Quit

                              Enter menu choice:
                              ]")
                end
```

This is the main menu and there are two items in it, through which users can either update or inquire system information. The first line of the method body is used to clear the screen.

If users chose the first menu item, they are going to enter the following menu, i.e. update_menu:

```
feature {NONE}
        HPTA_TRANSIT_INFO ... update_menu is
                do
                        execute_command_line("cls")
                        io.put_string("[
______________________________***************************************
__________________________________________Welcome_to_HPTA
______________________________***************************************
_________________________________A_Add
_________________________________D_Delete
_________________________________G_Go_back

______________________________Enter_menu_choice:
______________________________]")
                end
```

In this menu, users can add new information, such as staffs and stations, as follow:

```
feature {NONE}
        HPTA_TRANSIT_INFO ... add_menu is
                do
                        execute_command_line("cls")
                        io.put_string("[
______________________________***************************************
__________________________________________Welcome_to_HPTA
______________________________***************************************
_________________________________S_Add_a_station
_________________________________E_Add_a_staff
_________________________________G_Go_back

______________________________Enter_menu_choice:
______________________________]")
                end
```

Follows the logic of method do_add:

```
feature {NONE}
        HPTA_TRANSIT_INFO ... do_add is
            local
                employee: STAFF
                id: INTEGER
                input, name, password, open, access, last: STRING
                is_end: BOOLEAN
                route: ROUTE
            do
                from
                until is_end
                loop
                    add_menu
                    io.read_line
                    input := io.last_string.twin
                    io.put_new_line
                    if not input.is_empty then
                        inspect input.first.to_upper
                        when 'G' then is_end := True
                        when 'S' then
                              io.put_string("%NEnter_station_name:_")
                              io.read_line
                              name := io.last_string.twin
                              io.put_string("%NEnter_open_hour:_")
```

```
                        io.read_line
                        open := io.last_string.twin
                        io.put_string("%NEnter_its_accessibility:_")
                        io.read_line
                        access := io.last_string.twin
                        io.put_string("%NEnter_route_number:_")
                        io.read_line
                        id := io.last_string.to_integer
                        io.put_string("%NEnter_the_name_of_its_last_station:_")
                        io.read_line
                        last := io.last_string.twin
                        route := db.find_route(id)
                        if route = Void then
                            create route.make(id)
                            route.add_station(name, access, open, last)
                            db.add_route(route)
                        else
                            route.add_station(name, access, open, last)
                        end
                    when 'E' then
                        io.put_string("%NEnter_your_ID:_")
                        io.read_line
                        id := io.last_string.to_integer
                        io.put_string("%NEnter_your_password:_")
                        io.read_line
                        password := io.last_string.twin
                        create employee.make(id, password)
                        db.add_staff(employee)
                    else
                    end
                end
            end
        end
```

they can also delete those information as follow:

```
feature {NONE}
    HPTA_TRANSIT_INFO... delete_menu is
            do
                execute_command_line("cls")
                io.put_string("[
______________________________***************************************
_____________________________________________Welcome_to_HPTA
______________________________***************************************
_________________________________S_Delete_a_station
_________________________________E_Delete_a_staff
_________________________________R_Delete_a_route
_________________________________G_Go_back

______________________________Enter_menu_choice:
______________________________]")
            end
```

Follows the logic of method do_delete:

```
feature {NONE}
    HPTA_TRANSIT_INFO... do_delete is
        local
            is_end: BOOLEAN
            num: INTEGER
            staff: STAFF
            route: ROUTE
            input, name: STRING
        do
            from
            until is_end
```

```
loop
    delete_menu
    io.read_line
    input := io.last_string.twin
    io.put_new_line
    if not input.is_empty then
        inspect input.first.to_upper
        when 'G' then is_end := True
        when 'S' then
                io.put_string("%NEnter_route_number:_")
                io.read_line
                num := io.last_string.to_integer
                io.put_string("%NEnter_station_name:_")
                io.read_line
                name := io.last_string.twin
                route := db.find_route(num)
                if route /= Void then
                    route.remove_station(name)
                else
                    io.put_string ("%NNo_such_a_station%N")
                    io.read_line
                end
        when 'R' then
                io.put_string("%NEnter_route_number:_")
                io.read_line
                num := io.last_string.to_integer
                route := db.find_route(num)
                if route /= Void then
                    db.delete_route(route)
                else
                    io.put_string ("%NNo_such_a_station%N")
                    io.read_line
                end
        when 'E' then
                io.put_string("%NEnter_ID:_")
                io.read_line
                num := io.last_string.to_integer
                staff := db.find_staff(num)
                if staff /= Void then
                    db.delete_staff(staff)
                else
                    io.put_string ("%NNo_such_a_staff%N")
                    io.read_line
                end
        else
        end
    end
end
end
```

According to the requirement, only authorized staffs can do such things, so this system will ask the user to log in the system before he or she enter the update menu. The following method do_update has the logic to require the user to enter his or her employee number and password first.

```
feature {NONE}
        HPTA_TRANSIT_INFO ... do_update is
            local
                id: INTEGER
                passed, is_end: BOOLEAN
                password, input: STRING
                staff: STAFF
            do
                io.read_line
```

```
            if db.is_locked then
                io.put_string ("%NEnter_employee_ID:_")
                io.read_line
                id := io.last_string.to_integer
                staff := db.find_staff(id)
                if staff /= Void then
                    io.put_string ("%NEnter_password:_")
                    io.read_line
                    password := io.last_string.twin
                    passed := staff.login(password)
                end
            else
                io.put_string ("[
___________________________The_list_of_authorized_staff_is_not_empty,
___________________________so_please_set_authorization_as_soon_as_possible...
___________________________]")
                passed := True
                io.read_line
            end
            if passed then
                from
                until is_end
                loop
                    update_menu
                    io.read_line
                    input := io.last_string.twin
                    io.put_new_line
                    if not input.is_empty then
                        inspect input.first.to_upper
                        when 'A' then do_add
                        when 'D' then do_delete
                        when 'G' then is_end := True
                        else
                        end
                    end
                end
                db.do_save
            else
                io.put_string ("%NLogin_failed!%N")
                io.read_line
            end
        end
```

The actual logging responsibility is assigned to class STAFF as public feature to class HPTA_TRANSIT_INFO:

```
feature {HPTA_TRANSIT_INFO}
        STAFF... login(passwd: STRING): BOOLEAN is
            require
                passwd /= Void
            do
                Result := password.is_equal(passwd)
            end
```

If the result is True, the user can continue his or her update, otherwise, this system will remain on the main menu.

If users chose the second menu item of the main menu, they will enter the following query menu without any bother, because the requirement says that any one can have access to the transit information.

```
feature {NONE}
        HPTA_TRANSIT_INFO... inquire_menu is
                do
                        execute_command_line("cls")
                        io.put_string("[
                                    ***************************************
                                                Welcome_to_HPTA
                                    ***************************************
                                         F_Find_a_possible_connection
                                         S_Show_a_route
                                         B_Browse_all_routes
                                         G_Go_back

                                    Enter_menu_choice:
                                    ]")
                end
```

The first item of this menu is used for users to find a possible connection. Following the logic of method do_inquire, users are required to enter their start, destination, as well as their desire time.

```
feature {NONE}
        HPTA_TRANSIT_INFO... do_inquire is
                local
                    input, start, dest: STRING
                    is_end: BOOLEAN
                    num, time: INTEGER
                    route: ROUTE
                do
                    from
                    until is_end
                    loop
                        inquire_menu
                        io.read_line
                        input := io.last_string.twin
                        io.put_new_line
                        if not input.is_empty then
                            inspect input.first.to_upper
                            when 'B' then
                                from route := db.get_first_bus_route
                                until route = Void
                                loop
                                    route.show
                                    route := db.get_next_bus_route(route)
                                end
                                from route := db.get_first_train_route
                                until route = Void
                                loop
                                    route.show
                                    route := db.get_next_train_route(route)
                                end
                                io.put_string ("%N%NStrike_any_key_to_continue...")
                                io.read_line
                            when 'F' then
                                io.put_string("%NEnter_the_station_name_of_your_start:_")
                                io.read_line
                                start := io.last_string.twin
                                io.put_string("%NEnter_the_station_name_of_your_destination:_")
                                io.read_line
                                dest := io.last_string.twin
                                io.put_string("%NEnter_your_desire_time(in_minutes):_")
                                io.read_line
                                time := io.last_string.to_integer
                                io.put_string(finder.get_connection(db, start, dest, time))
```

```
                io.put_string ("%N%NStrike_any_key_to_continue...")
                io.read_line
            when 'S' then
                io.put_string ("Input_the_route_number_(10_-_999):_")
                io.read_line
                num := io.last_string.to_integer
                route := db.find_route(num)
                if route /= Void then
                    route.show
                else
                    io.put_string ("Sorry_there_is_no_such_a_route")
                end
                io.put_string ("%N%NStrike_any_key_to_continue...")
                io.read_line
            when 'G' then is_end := True
            else
            end
        end
    end
end
```

Now, it is time to implement the methods of class ROUTE

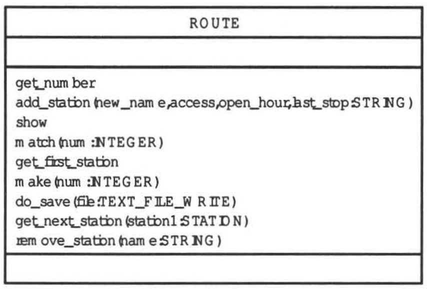


Figure 4.10: The methods of class ROUTE

The creation of ROUTE is make, which can be invoke by class HPTA_TRANSIT_INFO

```
creation ROUTE...make
```

The main task of make is initialize the route number and station list

```
feature {HPTA_TRANSIT_INFO}
    ROUTE...make (num: INTEGER) is
        require
            num > 9
            num < 1000
        do
            number := num
            create stops.make
        ensure
            number = num
            stops /= Void
        end
```

According the requirement, route number must be two- or three-digit number, so we define the following invariant for class ROUTE.

```
invariant
        ROUTE... number > 9
        number < 1000
end
```

At any time, its client get route number by the following method:

```
feature {ANY}
     ROUTE... get_number : INTEGER is
         do
            Result := number
         end
```

also, by the following method to tell if the current route is which we want:

```
feature {ANY}
      ROUTE... match (num: INTEGER): BOOLEAN is
           do
              Result := num = number
           end
```

By the following method, its client adds new stations for this ROUTE object and at the same time set the name, the accessibility, the opening hour, and last station for this new station.

```
feature {HPTA_TRANSIT_INFO, DATABASE}
      ROUTE... add_station(new_name, access, open_hour, last_stop: STRING) is
           local
              new_station: STATION
              i: INTEGER
              last: STRING
           do
              from i := stops.lower
              until i > stops.upper or else stops.item(i) .match(new_name)
              loop
                  i := i + 1
              end
              if i > stops.upper then
                    create new_station.make(new_name, access, open_hour)
                    last := last_stop.twin
                    last.to_upper
                    if last.same_as("NONE") then
                        stops.add_first (new_station)
                    else
                        from i := stops.lower
                        until i > stops.upper or else stops.item(i) .match(last_stop)
                        loop
                            i := i+1
                        end
                        if i <= stops.upper then
                            stops.add (new_station, i+1)
                        else
                            create new_station.make (last_stop, access, open_hour)
                            stops.add_last (new_station)
                            stops.add_last (new_station)
                        end
                    end
              end
           end
```

HPTA_TRANSIT_INFO object removes a certain station by the following method, whose only parameter is the name of the target station.

```
feature {HPTA_TRANSIT_INFO}
    ROUTE... remove_station(name: STRING) is
        local
            i: INTEGER
        do
            from i := stops.lower
            until i > stops.upper or else stops.item(i) .match(name)
            loop
                i := i + 1
            end
            if i <= stops.upper then
                stops.remove(i)
            end
        end
```

The subclasses of CONNECTION_FINDER use the following two methods to visit all stations in this route

```
feature {CONNECTION_FINDER}
    ROUTE... get_first_station: STATION is
        local
            station: STATION
        do
            if stops.upper > 0 then
                station := stops.first
            end
            Result := station
        end
    get_next_station(station1: STATION): STATION is
        require
            station1 /= Void
        local
            station: STATION
        do
            if stops.index_of(station1) < stops.upper then
                station := stops.item(stops.index_of(station1)+1)
            end
            Result := station
        end
```

Class ROUTE keep the secret of saving itself, so DATABASE object can call this method to fulfill the task. Actually, such assignment is worth to discuss. Maybe should move to the subclasses of DATABASE, because only they know exactly how to save those data.

```
feature {DATABASE}
    ROUTE... do_save(file: TEXT_FILE_WRITE) is
        require
            file.is_connected
        local
            i: INTEGER
            tag, last: STRING
        do
            if number > 99 then
                tag := "b"
            else
                tag := "t"
            end
```

```
            last := "None"
            from i := stops.lower
            until i > stops.upper
            loop
                file.put_string(tag+"%N")
                stops.item(i) .do_save(file)
                file.put_string("_" + number.to_string +"_" + last +"%N")
                last := stops.item(i) .get_name.twin
                i := i + 1
            end
        end
```

Similarly, the following method is responsible for showing the details of this route, but only class HPTA_TRANSIT_INFO know exactly how to display with interface, so this method should be move to class HPTA_TRANSIT_INFO.

```
feature {HPTA_TRANSIT_INFO}
    ROUTE...show is
        local i: INTEGER
        do
            if number > 99 then
                io.put_string ("%NBus_route_No.")
            else
                io.put_string ("%NTrain_route_No.")
            end
            io.put_integer (number)
            io.put_string (":_")
            from i := stops.lower
            until i > stops.upper
            loop
                stops.item(i) .show
                if i < stops.upper then
                    io.put_string ("->")
                end
                i := i+1;
            end
            io.put_new_line
        end
```

Same problem can be found on the method show of class STATION

```
feature {ROUTE}
        STATION...show is
            do
                io.put_string (name)
            end
```

Now, let us look at the class STATION, whose creation is method make too,

```
create STATION...make
```

and defined as following:

```
feature {ROUTE}
        STATION...make (new_name, new_open, new_acc: STRING) is
            require
                new_name /= Void
                new_open /= Void
                new_acc /= Void
            do
                name := new_name.twin
                open := new_open.twin
```

```
            accessibility := new_acc.twin
        end
```

The main task of it is to initial these three features of class STATION. At any time, its client can visit these three features by the following methods:

```
feature {ROUTE, CONNECTION_FINDER}
        STATION...get_name: STRING is
            do
                Result := name.twin
            end
        get_acc: STRING is
            do
                Result := accessibility.twin
            end
        get_open: STRING is
            do
                Result := open.twin
            end
```

Similar with the method do_save of class ROUTE, this method should be moved into the subclasses of DATABASE.

```
feature {ROUTE}
        STATION...do_save(file: TEXT_FILE_WRITE) is
            require
                file.is_connected
            do
                file.put_string(name + "_" + accessibility + "_" + open)
            end
```

The same problem can be found on class STAFF

```
feature {DATABASE}
        STAFF...do_save(file: TEXT_FILE_WRITE) is
            require
                file.is_connected
            do
                file.put_string("s%N" + number.to_string + "_" + password + "%N")
            end
```

We identify station with name only, i.e. if two stations share the same name, we assume they are the same station. Here case is insensitive.

```
feature {ROUTE}
        STATION...match (targetname: STRING): BOOLEAN is
            require
                targetname /= Void
            do
                Result := name.same_as (targetname)
            end
```

Now, let us talk about the implementation of class STAFF.

The creation of class STAFF is make

```
creation{ANY} STAFF...make
```

it is defined as following:

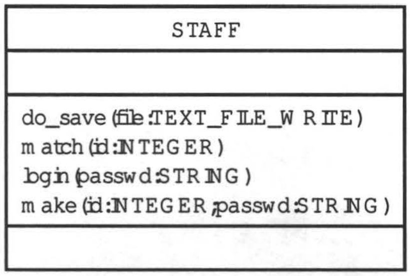


Figure 4.11: The methods of class STAFF

```
feature {ANY}
        STAFF...make (id: INTEGER; passwd: STRING) is
            require
                id >= 0
                passwd /=Void
            do
                number := id;
                password := passwd.twin
            ensure
                number >= 0
                password = passwd
            end
```

its main task is initialize staff's id and password.

Method match is used to identify a certain staff and is defined as following:

```
feature {ANY}
        STAFF...match (id: INTEGER): BOOLEAN is
            do
                Result := id = number
            end
```

Any staff has an unique employee number, which is generated from 0, and a password, which must not be Void:

```
invariant
        STAFF...number_positive: number >= 0
        password_not_void: password /= Void
end
```

4.1.9 Automatic Code Listing

So far, we have implement the system. In order to give an integrated view for ones who are used to read code, Spark inserts all program code here automatically.

Automatical code is listed in Appendix C.

4.1.10 Testing

Updating system

When no staff is authorized, we try to update system information. The result is

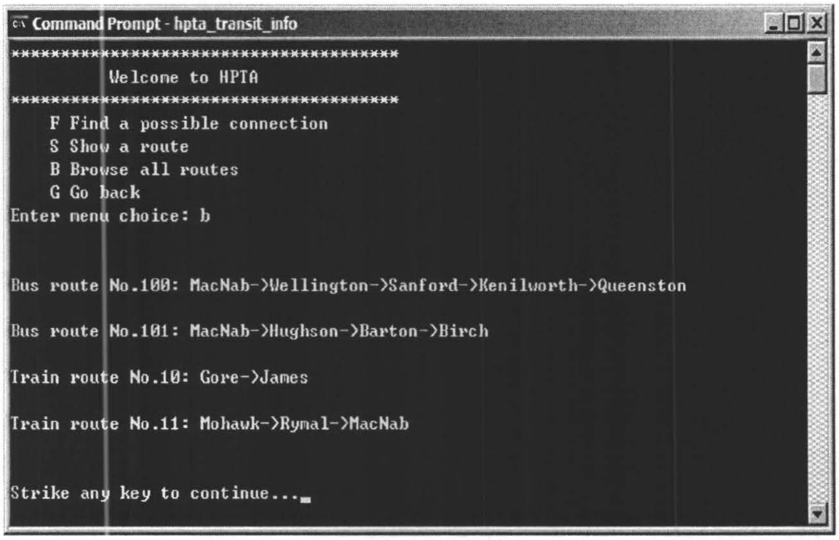
Otherwise, we try to update system information. The system requires ID and password for logging in as following:

These results satisfy the design requirements.

Browsing all routes

We try to browse the information of all routes as following:

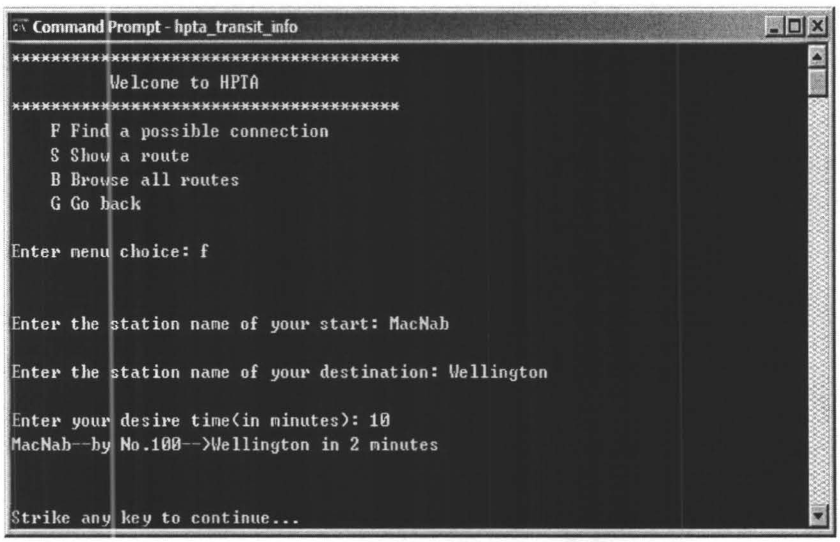
These result satisfies the design requirements.



```
Command Prompt - hpta_transit_info
****************************************
          Welcome to HPTA
****************************************
    F Find a possible connection
    S Show a route
    B Browse all routes
    G Go back
Enter menu choice: b


Bus route No.100: MacNab->Wellington->Sanford->Kenilworth->Queenston

Bus route No.101: MacNab->Hughson->Barton->Birch

Train route No.10: Gore->James

Train route No.11: Mohawk->Rymal->MacNab


Strike any key to continue...
```

Finding connection

We try to find a connection between two stations as following:



```
Command Prompt - hpta_transit_info
****************************************
          Welcome to HPTA
****************************************
    F Find a possible connection
    S Show a route
    B Browse all routes
    G Go back

Enter menu choice: f


Enter the station name of your start: MacNab

Enter the station name of your destination: Wellington

Enter your desire time(in minutes): 10
MacNab--by No.100-->Wellington in 2 minutes


Strike any key to continue...
```

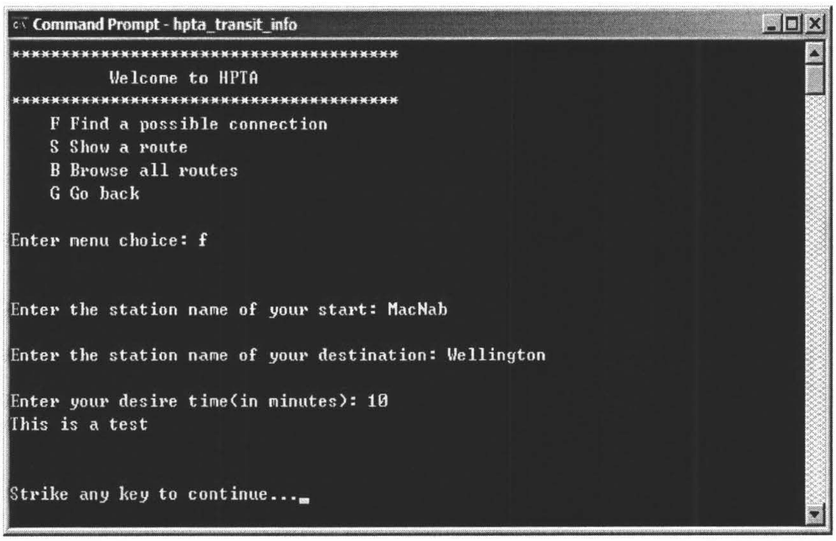
These result satisfies the design requirements.

Strategy pattern

We construct a sample connection finder class and change the algorithm at run-time.

```
class FAKE_FINDER
inherit
        CONNECTION_FINDER
creation {ANY}
        make
feature {HPTA_TRANSIT_INFO}
        make is
            do
            end
        get_connection(start, destination: STRING; time: INTEGER): STRING is
            do
                Result := "This is a test%N";
            end
end
```

The result is



These result satisfies the design requirements.

Chapter 5

Implementation

Chapter 3 has showed us the key features of OOLP and some supporting tools. In this chapter, we present the implementation of Spark in a literate way using Spark itself. The rest of this chapter that follow is the actual output of Spark source file.

5.1 Introduction

The two reasons why we present the implementation of Spark in a literate way are that we want to show the universality of Spark, i.e. it can work with not only object-oriented programming languages, but also structured programming languages, and that since the main task of Spark is to parse the syntax of a certain programming language, which is a fairly stable structure, structured programming makes the program clear, simple and efficient. This choice is also followed by one drawback that we have to draw the diagrams by hand.

Spark is implemented entirely in perl. We choose perl mainly because it is good at text manipulation, which is the main task of Spark, and perl is a stable, cross platform programming language, which leads to Spark being inherently platform-independent.

In order to gain more flexibility, we separate Spark into two parts, i.e., front end and back end (see Figure 5.1). The front end is responsible for explaining graphical notation settings and parsing program code chunks; the back end takes care of producing graphical notation files. So far, we have developed three front ends, which are used for Eiffel, Lime, and perl itself respectively.

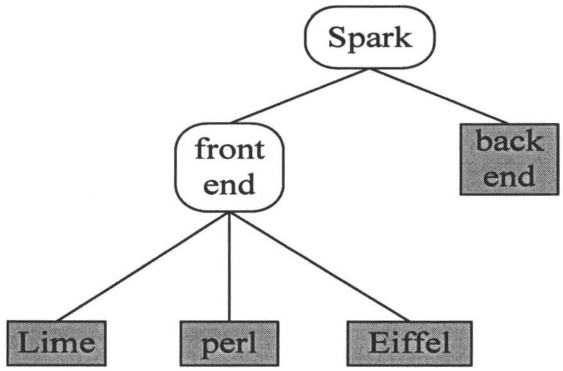


Figure 5.1: Top-level structure for Spark

5.2 Graphic Notation describer

The graphic notation describer is a interim file used to describe all the graphic notations included in the software documentation and the only input file of the back end of Spark. So the changes coming from either document formatting language or programming language do not impact the back end. In addition, except several important setting tags, programmers do not know anything about it, because this file is going to be deleted by the back end before it finishes its work.

In the following table of graphic notation describer structure, terminals are shown in bold font and nonterminals in italics, parentheses '(' and ')' indicate grouping when needed, square brackets '[' and ']' enclose optional items, curly parentheses '{' and '}' show the (zero or more) repeatable items, and vertical bars '|' separate alternatives.

modFile	::=	{*classDiagram*} {*class*}
classDiagram	::=	**@CLASSDIAGRAM** *nameSequence* [**@VERTICAL**] [*format*]
format	::=	**@BRIEF** | **@CONCISE** | **@METHOD** | **@ATTRIBUTE** | **@ACTION**
class	::=	**@CLASS** *name* {*statementSequence*}
		| **@INTERFACE** *name* { *statementSequence* }
statementSequence	::=	*statement statementSequence*
statement	::=	*extendStmt*
		| *inheritStmt*
		| *implementStmt*
		| *fieldStmt*
		| *cotrStmt*

| *methodStmt*
| *actionStmt*
| *dependentStmt*
extendStmt ::= **@EXTEND** *nameSequence*
inheritStmt ::= **@INHERIT** *nameSequence*
implementStmt ::= **@IMPLEMENT** *nameSequence*
fieldStmt ::= **@VAR** *name* { *modifier* }
cotrStmt ::= **@INIT** *name* { *modifier* }
methodStmt ::= **@METHOD** *name* { *modifier* }
actionStmt ::= **@ACTION** *name* { *modifier* }
nameSequence ::= *name nameSequence*
modifier ::= *visibility* | *type* | ‘(’*paraSequence*‘)’
visibility ::= **@PRIVATE** | **@PUBLIC** | **@PROTECTED**
paraSequence ::= *parameter*; *paraSequence*
parameter ::= *name* ‘:’ *type*
type ::= *name*
name ::= *letter* { *letter* | *digital* }
letter ::= **a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t |**
u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | D | L |
M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
digital ::= **1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0**

Table 5.1: The block structure of graphic notation describer.

A graphic notation describer consists of multiple *classDiagrams* followed by multiple *class*es. *classDiagram* begin with the key word “@CLASSDIAGRAM”. *nameSequence* is a list of class names included in this diagram. “@VERTICAL” is used to set the direction of the specified graphic notation, i.e. if “@VERTICAL” is set, the diagram will be drawn vertically, otherwise horizontally. “@HEAD”, “@BRIEF” and “@CONCISE” are used to control the format of the specified graphic notation. If “@CONCISE” is set, the class diagram will hide all the information about the method's parameters of the involved class. If “@BRIEF” is set, the class diagram will be shown without parameters and types. If “@HEAD” is set, the class diagram will be shown with class name only. Ones also can use “@METHOD”, “@ATTRIBUTE”, and “@ACTION” to control the display areas. For example, if “@METHOD” is set, all class methods will be shown in the diagram and if “@ATTRIBUTE” is set, only attributes of class can be saw in the diagram. All the setting tags are included in source file as specifical comments, so they are transparent for everything except the front end. *class* begin with the key words "@CLASS" or "@INTERFACE". class name followed by a list of statements in the form of one statement each line. "*inheritStmt*", "*extendStmt*" and "*implementStmt*" begin respectively with the key words "@INHERIT", "@EXTEND", and "@IMPLEMENT", which are followed by a list of superclass names. "fieldStmt", "cotrStmt", "methodStmt", and "actionStmt" are the member statements of this class and begin with their key word respectively.

5.3 Front End

The front end of Spark takes an OOLP program file as input and produces machine-readable program code files and one graphic notation script file as output. In addition, it can insert the copy of machine-readable program code list back into the OOLP program file upon the request. The only entrance of front end is *main*, which depends on three modules, i.e. *parseSource*, *doOutput*, and *callBackEnd* (see Figure 5.2). The rest of this section discusses the implementation of Spark front end for Lime in details.

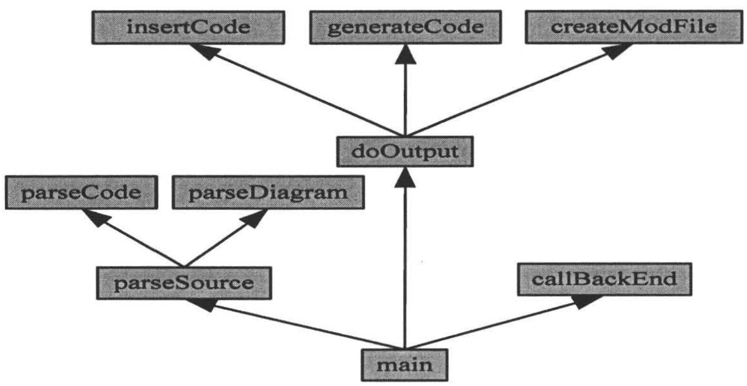


Figure 5.2: Top-level structure for Spark front end

The entrance of Spark front end for lime,

```
Sparkf-lime...&main;
```

is defined as following:

```
Sparkf-lime...sub main{
    $#ARGV == 0 || die "Usage: perl sparkf.pl filename\n";
```

```
    open(SOURCE, "< $old") || die "Cannot open $old!\n";
    close SOURCE || die "Can't close $old: $!";
    &parseSource;
    &doOutput;
    &callBackEnd;
}
```

The first process in main is parsing the OOLP program, whose name is hold by a local variable $old.

```
Sparkf-lime...my $old = $ARGV[0];
```

Module *parseSource* keeps reading the content of the program line by line:

```
Sparkf-lime...sub parseSource{
    while(1){
        &newLine;
        last if($done);
        if($currentLine =~ /image::/){
            &parseDiagram;
        }elsif($currentLine =~ /^-{5,}/){
            &parseCode;
        }
    }
}
```

using the following function:

```
Sparkf-lime...sub newLine{
    my @array = @currentWords;
    push(@lastWords, \@array);
    if($#lineBuffer < 0){
        unless($done){
            $done = 1;
            while(<>){
                chop;
                $currentLine = $_;
                @currentWords = split;
                $pointer = 0;
                $done = 0;
                last if($#currentWords >= 0);
                $done = 1;
            }
        }
    }else{
        my $refArray = pop(@lineBuffer);
        @currentWords = @$refArray;
        $pointer = 0;
    }
}
```

where local variable @currentWords defined as

```
Sparkf-lime...my @currentWords = ();
```

always keeps the words of the current line in array form and local variable @lastWords defined as

```
Sparkf-lime...my @lastWords = ();
```

keeps all the addresses of old @currentWords in order and local variable $currentLine defined as

```
Sparkf-lime...my $currentLine = "";
```

always keeps the words of the current line in string form and local variable $done defined as

```
Sparkf-lime...my $done = 0;
```

will be set as 1 after the last line is read and local variable @lineBuffer defined as

```
Sparkf-lime...my @lineBuffer = ();
```

is used to hold the current line temporarily in the case that front end need read again the last word, which is in the last line, and the current line is still needed. The local variable $pointer defined as

```
Sparkf-lime...my $pointer = 0;
```

is used to point out the current word the front end is reading and increases by one after the execution of function *nextWord* defined as

```
Sparkf-lime...sub nextWord{
  if($pointer >= $#currentWords){
    &newLine;
  }else{
    $pointer++;
  }
  $currentWords[$pointer];
}
```

and decreases by one after the execution of function *lastWord* defined as

```
Sparkf-lime...sub lastWord{
  if($pointer == 0){
     if($#lastWords >= 0){
        my @array = @currentWords;
        push(@lineBuffer, \@array);
        my $refArray = pop(@lastWords);
        @currentWords = @$refArray;
        $pointer = $#currentWords;
     }
  }else{
     $pointer--;
  }
  $currentWords[$pointer];
}
```

If a diagram tag such as "image::" is encountered, the front end will enter the status of parsing diagrams:

```
Sparkf-lime...sub parseDiagram{
  /\w+::(\w+)\.(\w+)/;
  my $picture = $1.".".$2;
```

```
  my $token = &nextWord;
  if($token =~ /\/\/\$/){
    for(my $i = 1; $i <= $#currentWords; $i++){
       $picture .= " ".$currentWords[$i];
    }
  }else{
    &lastWord;
  }
  push(@diagramList, $picture);
}
```

Front end keeps all the information concerning the current diagram in a local variable, $picture, and then pushes it into the diagram list:

```
Sparkf-lime...my @diagramList = ();
```

If a code tag such as a serial of "-" is encountered, the front end will enter the status of parsing code, which is the main difference between different front ends:

```
Sparkf-lime...sub parseCode{
  my $token = &nextWord;
  until($token =~ /^-{5,}/){
    if($token eq "class"){
      parseClass(0);
    }elsif($token eq "final"){
      &nextWord;
      parseClass(1);
    }elsif($memberList{$token}){
      parseMembers($token);
    }
    $token = &nextWord;
  }
}
```

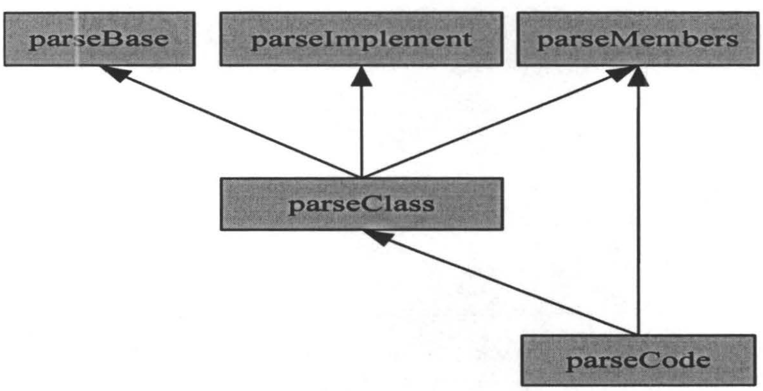


Figure 5.3: The structure of Module parseCode

The front end for lime parses various code units such as a class, a class member, and a statement, according to the syntax one by one as following:

If the first token of a code block is "final" or "class", then this block is a class block. For class block, front end parse it according to the syntax that
class ::= ["final"] "class" identifier base implement members "end"
as following:

```
Sparkf-lime...sub parseClass{
  $currentClass = &nextWord;
  my $classBody = hashAdd(\%classList, $currentClass);
  $classBody->{"final"} = $_[0];
  my $token = &nextWord;
  if($token eq "inherit"){
     parseBase("inherit");
     $token = &nextWord;
  }
  if($token eq "extend"){
     parseBase("extend");
     $token = &nextWord;
  }
  while($token eq "implement"){
     $token = &nextWord;
     parseImplement($token);
     $token = &nextWord;
  }
  while($memberList{$token}){
     parseMembers($token);
     $token = &nextWord;
  }
  die "parsing class failed![$token]\n" if($token ne "end");
}
```

where the local variable $currentClass keeps the name of current class and is defined as:

```
Sparkf-lime...my $currentClass = "";
```

and supporting function *hashAdd* is used to add item to a hash table without duplicate and defined as:

```
Sparkf-lime...sub hashAdd{
  my($hash, $item) = @_;
  if(not $hash->{$item}){
     my %newHash = ();
     $hash->{$item} = \%newHash;
  }
  $hash->{$item};
}
```

If the first token of a code block is one of the members of

```
Sparkf-lime...my %memberList = ("public", 1, "action",      1, "template",1, "const",  1,
                                "initialization",            1, "method",  1, "var",    1);
```

then the block is a class member block. There are six kinds of legal members, i.e. constant, variable, method, action, initialization, and label:

```
Sparkf-lime...sub parseMembers{
  my ($token, $public) = ($_[0], 0);
```

```
  if($token eq "public"){
     $public = 1;
     $token = &nextWord;
  }
  if($token eq "const"){
     parseConst(&nextWord, $public);
  }elsif($token eq "var"){
     parseVariable($public, &nextWord);
  }elsif(($token eq "method")||($token eq "template")){
     parseMethod($token, $public);
  }elsif($token eq "action"){
     parseAction(&nextWord);
  }elsif($token eq "initialization"){
     parseInitialization(&nextWord);
  }elsif($token =~ /^'\w+'/){
     parseLabel($token);
  }
}
```

For constants, the syntax is
constant ::= "const" identifier [":" type] "=" expression:

```
Sparkf-lime...sub parseConst{
  my ($token, $public) = @_;
  my $identifier = "";
  my $const = "$public.";
  if($token =~ /^(\w+)$/){
     $identifier = $token;
     $token = &nextWord;
  }elsif($token =~ /^(\w+)(\S+)/){
     $identifier = $1;
     $token = $2;
  }
  if($token eq ":"){
     $const .= " : ".parseType(&nextWord);
     $token = $currentWords[$pointer];
  }elsif($token =~ /^:(\w+\S*)/){
     $const .= " : ".parseType($1);
     $token = $currentWords[$pointer];
  }
  if($token =~ /=(\S+)/){
     $const .= " = ".parseExpression($1);
  }elsif($token =~ /=$/){
     $const .= " = ".parseExpression(&nextWord);
  }else{
     $token = &nextWord;
     if($token =~ /=(\S+)/){
        $const .= " = ".parseExpression($1);
     }elsif($token =~ /=$/){
        $const .= " = ".parseExpression(&nextWord);
     }else{
        die "parse const statement failed!\n";
     }
  }
  my $classBody = hashAdd(\%classList, $currentClass);
  my $constFields = hashAdd($classBody, "const");
  $constFields->{$identifier} = $const;
}
```

The supporting module *parseExpression* is defined according to its syntax
expression ::= conjunction "or" conjunction
as following:

```
Sparkf-lime...sub parseExpression{
  my $token = $_[0];
  my $expression = "";
  $token = $1 if($token =~ /(.+);/);
  if($token =~ /\Wor\W/){
    my @wordBuffer = split(/\s+/, $token);
    $token = shift(@wordBuffer);
  }else{
    $expression = parseConjunction($token);
    while($expression ne ""){
       $token = &nextWord;
       if($token eq "or"){
          $expression .= " or ".parseConjunction(&nextWord);
       }else{
          &lastWord;
          last;
       }
    }
  }
  $expression;
}
```

The syntax of conjunction is
conjunction ::= relational "and" relational :

```
Sparkf-lime...sub parseConjunction{
  my $conjunction = parseRelational($_[0]);
  while($conjunction ne ""){
    my $token = &nextWord;
    if($token eq "and"){
      $conjunction .= " and ".parseRelational(&nextWord);
    }else{
      &lastWord;
      last;
    }
  }
  $conjunction;
}
```

The syntax of relational is
relational ::= additive [("<" | ">" | "≤" | "≥" | "=" | "!=") additive]:

```
Sparkf-lime...sub parseRelational{
  my $relational = "";
  my $token = $_[0];
  if($token =~ /(.+)(<|>|!=|=|<=|>=|  |  )(.+)/){
     $relational = parseAdditive($1)." ".$2." ".parseAdditive($3);
  }elsif($token =~ /(.+)(<|>|!=|=|<=|>=|  |  )$/){
     $relational = parseAdditive($1)." ".$2." ".parseAdditive(&nextWord);
  }else{
     $relational = parseAdditive($token);
     $token = &nextWord;
     if($token =~ /^(<|>|!=|=|<=|>=|  |  )$/){
        $relational .= " ".$1." ".parseAdditive(&nextWord);
     }elsif($token =~ /^(<|>|<=|>=|!=|=|  |  )(.+)/){
        $relational .= " ".$1." ".parseAdditive($2);
     }else{
        &lastWord;
     }
  }
  $relational;
}
```

The syntax of additive is
additive ::= multiplicative ("+" | "-") multiplicative

```
Sparkf-lime...sub parseAdditive{
  my $additive = parseMultiplicative($_[0]);
  my $token = $currentWords[$pointer];
  while($additive ne ""){
     if($token =~ /(\+|-)$/){
        $additive .= " $1 ".parseMultiplicative(&nextWord);
     }elsif($token =~ /(\+|-)(\S+)/){
        $additive .= " $1 ".parseMultiplicative($2);
     }else{
        $token = &nextWord;
        if($token =~ /^(\+|-)$/){
           $additive .= " $1 ".parseMultiplicative(&nextWord);
        }elsif($token =~ /(\+|-)(\S+)/){
           $additive .= " $1 ".parseMultiplicative($2);
        }else{
           $token = $currentWords[$pointer];
           &lastWord;
           $token = $currentWords[$pointer];
           last;
        }
     }
  }
  $additive;
}
```

The syntax of multiplicative is
multiplicative ::= unary("*" | "/" | "div" | "mod") unary

```
Sparkf-lime...sub parseMultiplicative{
  my $multiplicative = parseUnary($_[0]);
  my $token = $currentWords[$pointer];
  while($multiplicative ne ""){
     if($token =~ /(\*|\/)(\S+)/){
        $token = $2;
        $multiplicative .= " $1 ".parseUnary($token);
     }elsif($token =~ /(\+|-)$/){
        $token = &nextWord;
        $multiplicative .= " $1 ".parseUnary($token);
     }else{
        $token = &nextWord;
        if($token =~ /^(\*|\/)(\S+)/){
           $token = $2;
           $multiplicative .= " $1 ".parseUnary($token);
        }elsif($token =~ /^(\+|-)$/){
           $token = &nextWord;
           $multiplicative .= " $1 ".parseUnary($token);
        }elsif($token eq "div"){
           $token = &nextWord;
           $multiplicative .= " div ".parseUnary($token);
        }elsif($token eq "mod"){
           $token = &nextWord;
           $multiplicative .= " mod ".parseUnary($token);
        }else{
           &lastWord;
           last;
        }
     }
  }
  $multiplicative;
}
```

The syntax of unary is
unary ::= ("-" | "not" | "+")unary | primary
and in turn the syntax of primary is
primary ::= integer | "nil" | "true" | "false" | designator | "new" name[actuals]:

```
Sparkf-lime...sub parseUnary{
  my ($unary, $token) = ("", $_[0]);
  if($token =~ /^(-|\+)(\S+)/){
     $unary = "$1".parseUnary($2);
  }elsif($token =~ /^(-|\+)$/){
     $unary = "$1".parseUnary(&nextWord);
  }elsif($token eq "not"){
     $unary = "not ".parseUnary(&nextWord);
  }elsif($token =~ /not\s+(.+)/){
     $unary = "not ".parseUnary($1);
  }elsif($token =~ /(\d+)/){
     $unary = "$1";
  }elsif(($token =~ /^(nil)\W*/) ||
         ($token =~ /^(true)\W*/) ||
         ($token =~ /^(false)\W*/)){
     $unary = $1;
  }elsif($token eq "new"){
     $unary = "new ".&nextWord;
     $token = &nextWord;
     if($token =~ /^\(/){
        $unary .= " ".parseActuals($token);
     }else{
        &lastWord;
     }
  }elsif($token =~ /new\s+(\w+)\s*(\(.+\))/){
     $unary = "new ".$1.parseActuals($2);
  }elsif($token =~ /(\w+)/){
     $unary = parseDesignator($token) unless($keyWordsList{$1});
  }
  $unary;
}
```

The syntax of designator is
designator ::= identifier "." identifier | actuals :

```
Sparkf-lime...sub parseDesignator{
  my ($designator, $token) = ("", $_[0]);
  if($token =~ /^(\w+)$/){
     $designator = $token;
     $token = &nextWord;
  }elsif(($token =~ /^(\w+)(\.\S+)/)||($token =~ /^(\w+)(\(\S+)/)){
     $designator = $1;
     $token = $2;
  }else{
     die "parse designator failed![$token]\n";
  }
  while(1){
     if($keyWordsList{$designator}){
        $designator = "";
        &lastWord;
        last;
     }
     if($token =~ /^\./){
        $token = ".".&nextWord if($token =~ /^\.$/);
        if($token =~ /^\.(\w+)$/){
           $designator .= ".".$1;
           $token = &nextWord;
        }elsif($token =~ /^\.(\w+)(\S+)/){
```

```
            $designator .= ".".$1;
            $token = $2;
        }else{
            die "parse designator failed![3]\n";
        }
    }elsif($token =~ /^\(/){
        $designator .= parseActuals($token);
        $token = $currentWords[$pointer];
        if(($token =~ /\)(\.\S*)/)||($token =~ /\)(\(\S*)/)){
            $token = $2;
        }elsif($token =~ /\)$/){
            $token = &nextWord;
        }else{last;}
    }else{
        &lastWord;
        last;
    }
  }
  $designator;
}
```

For variables, the syntax is variable ::= "var" idList ":" type:

```
Sparkf-lime...sub parseVariable{
  my @vars = parseIdList($_[1]);
  my ($type, $public) = ("", $_[0]);
  my $token = $currentWords[$pointer];
  if($token =~ /\w:(\w+\S*)/){
      $type = parseType($1);
  }elsif($token =~ /\w+:$/){
      $type = parseType(&nextWord);
  }else{
      $token = &nextWord;
      if($token =~ /^:(\w+\S*)/){
          $type = parseType($1);
      }elsif($token eq ":"){
          $type = parseType(&nextWord);
      }else{
          die "parse variable failed!\n";
      }
  }
  foreach my $v(@vars){
      if($v =~ /(\w+)\.(\w+)/){
          $currentClass = $1;
          $v = $2;
      }
      my $classBody = hashAdd(\%classList, $currentClass);
      my $varFields = hashAdd($classBody, "var");
      $varFields->{$v} = $type;
  }
}
```

For methods, the syntax method ::= "method" identifier formals[":" type] statement:

```
Sparkf-lime...sub parseMethod{
  my @vars = ();
  my ($token, $public) = @_;
  my ($identifier, $type) = ("", "");
  my ($formals, $statement) = ("", "");
  if($token eq "method"){
      $token = &nextWord;
      if($token =~ /^(\w+\.\w+)$/){
          $identifier = $1;
          $token = &nextWord;
```

```
    }elsif($token =~ /^(\w+\.\w+)(\S+)/){
        $identifier = $1;
        $token = $2;
    }elsif($token =~ /^(\w+)$/){
        $identifier = $1;
        $token = &nextWord;;
    }elsif($token =~ /^(\w+)(\S+)/){
        $identifier = $1;
        $token = $2;
    }else{
        die "parse method failed!\n";
    }
    $formals = parseFormal($token);
    $token = $currentWords[$pointer];
    if(($token =~ /\):$/)||
       (($formals eq "")&&($token =~ /:$/))){
        $token = &nextWord;
        $type = parseType($token);
    }elsif(($token =~ /\):(\w+.*)/)||
           (($token =~ /:(\w+.*)/)&&($formals eq ""))){
        $type = parseType($1);
    }else{
        $token = &nextWord;
        if($token =~ /:$/){
            $token = &nextWord;
            $type = parseType($token);
        }elsif($token =~ /:(\w+.*)/){
            $type = parseType($1);
        }else{
            &lastWord;
        }
    }
    my $newWord = 0;
    $token = $currentWords[$pointer];
    if(($type ne "")||(($type eq "")&&($formals eq ""))){
        if($token =~ /(\'.*)/){
            $token = $1;
        }else{
            $newWord = 1;
            $token = &nextWord;
        }
    }elsif($formals ne ""){
        if($token =~ /\)(.+)/){
            $token = $1;
        }else{
            $newWord = 1;
            $token = &nextWord;
        }
    }
    $statement = parseStatementList($token);
    &lastWord if(($statement eq "") && $newWord);
    if($identifier =~ /(\w+)\.(\w+)/){
        $currentClass = $1;
        $identifier = $2;
    }
    $identifier .= ".".$formals;
    my $classBody = hashAdd(\%classList, $currentClass);
    my $varFields = hashAdd($classBody, "method");
    $varFields->{$identifier}={TYPE=>$type,STATEMENT=>$statement}
    unless($varFields->{$identifier});
    $varFields->{$identifier}->{STATEMENT} = $statement
    if(($varFields->{$identifier}->{STATEMENT}eq"")&&($statement ne ""));
    $varFields->{$identifier}->{TYPE} = $type
    if(($varFields->{$identifier}->{TYPE}eq "")&&($type ne ""));
  }
}
```

where the supporting module *parseFormal* is defined as following according to the syntax (formals ::= ["(" idList ":" type "," idList ":" type ")"]):

```
Sparkf-lime...sub parseFormal{
  my ($formals, $token) = ("", $_[0]);
  if($token =~ /\(/){
     $formals = "(";
     if($token =~ /\($/){
        $token = &nextWord;
     }elsif($token =~ /\((\S+)/){
        $token = $1;
     }
     while(1){
        if($token =~ /(.*\))\S*/){
           $formals .= $1;
           last;
        }else{
           $formals .= $token;
           $token = &nextWord;
        }
     }
  }
  $formals;
}
```

where the supporting module *parseType* is defined as following according to the syntax (type ::= ["shared"] "array"[expression"," expression]"of"(name | "integer" | "boolean")):

```
Sparkf-lime...sub parseType{
  my $type = "";
  my $token = $_[0];
  if($token eq "shared"){
     $type = "shared ";
     $token = &nextWord;
  }
  while(1){
     if($token eq "array"){
        $type .= "array ";
        while($token ne "of"){
           $token = &nextWord;
           $type .= "$token ";
        }
        $type .= "of ";
        $token = &nextWord;
     }elsif($token eq "set"){
        $type .= "set ";
        $type .= &nextWord." ";
        $token = &nextWord;
     }else{
        last;
     }
  }
  if(($token =~ /^(integer)\W*/)||($token =~ /^(boolean)\W*/)){
     $type .= $1;
  }elsif($token =~ /^\w+/){
     $type .= parseName($token);
  }else{
     die "parse type failed![2]\n";
  }
  $type;
}
```

the supporting module *parseStatementList* is defined as following according to the syntax (statementList ::= statement ";" statement):

```
Sparkf-lime...sub parseStatementList{
  my $statementList = parseStatement($_[0]);
  $statementList = "" if($statementList =~ /^\s+$/);
  while($statementList ne ""){
    my $token = $currentWords[$pointer];
    my @array = split(/;/, $token);
    for(my $i = 1; $i <= $#array; $i++){
        $statementList .=";\n".parseStatement($array[$i]);
    }
    if($token =~ /;$/){
      $statementList .=";\n".parseStatement(&nextWord);
    }else{
      $token = &nextWord;
      if($token =~ /^;$/){
        $statementList .=";\n".parseStatement(&nextWord);
      }elsif($token =~/^;(\S+)/){
        $statementList .=";\n".parseStatement($1);
      }else{
        &lastWord;
        $statementList .="\n";
        last;
      }
    }
  }
  $statementList;
}
```

Statement is the most complex one in Lime. Its syntax is

statement ::= designator [":="expression] |
designatorList ":=" expressionList |
"begin" statement ";" statement "end" |
"when" expression "do" statement |
"if" expression "then" statement ["else" statement] |
"while" expression "do" statement |
"repeat" statement ";" statement "until" expression |
variable statement |
constant statement |
"return" [expression] |
label

```
Sparkf-lime...sub parseStatement{
  my $space = "     ";
  $layers++;
  for(my $i = 0; $i < $layers; $i++){
     $space .="   ";
  }
  my ($statement, $token) = ($space, $_[0]);
  if($token eq "begin"){
```

```
    $statement .= "begin\n".parseStatementList(&nextWord);
    $token = &nextWord;
    die "pasrse begin statement failed![$token]" if($token !~ /^end\W*/);
    $statement .= $space."end";
  }elsif($token eq "when"){
    $statement .= "when ".parseExpression(&nextWord);
    $statement .= " ".&nextWord."\n".parseStatement(&nextWord);
  }elsif($token eq "if"){
    $statement .= "if ".parseExpression(&nextWord);
    $statement .= " ".&nextWord."\n".parseStatement(&nextWord);
    $token = $currentWords[$pointer];
    if(($token =~ /\'.+\'else(\'.+)/)||($token =~ /\)else(\'.+)/)){
      $statement .= "\n".$space."else\n".parseStatement($1);
    }elsif(($token =~ /\'(.+)\'else$/)||($token =~ /\)else$/)){
      $statement .= "\n".$space."else\n".parseStatement(&nextWord);
    }else{
      $token = &nextWord;
      if($token eq "else"){
        $statement .= "\n".$space."else\n".parseStatement(&nextWord);
      }elsif($token =~ /^else(\'.+)/){
        $statement .= "\n".$space."else\n".parseStatement($1);
      }else{
        &lastWord;
      }
    }
  }elsif($token eq "while"){
    $statement .= "while ".parseExpression(&nextWord);
    $statement .= " ".&nextWord."\n".parseStatement(&nextWord);
  }elsif($token eq "repeat"){
    $statement .= "repeat\n".parseStatementList(&nextWord);
    $statement .= $space.&nextWord." ".parseExpression(&nextWord);
  }elsif($token eq "var"){
    my @vars = parseIdList(&nextWord);
    $token = $currentWords[$pointer];
    my $type = "";
    if($token =~ /\w:(\w+\S*)/){
      $type = parseType($1);
    }elsif($token =~ /\w:$/){
      $type = parseType(&nextWord);
    }else{
      $token = &nextWord;
      if($token =~ /^:(\w+\S*)/){
        $type = parseType($1);
      }elsif($token eq ":"){
        $type = parseType(&nextWord);
      }
    }
    $statement .= "var ";
    while($#vars >=0){
      if($#vars > 0){
        $statement .= shift(@vars).", ";
      }else{
        $statement .= shift(@vars)." : ".$type;
      }
    }
  }elsif($token eq "const"){
    $token = &nextWord;
    $statement .= "const ".$1 if($token =~ /^(\w+)/);
    $token = $currentWords[$pointer];
    if($token =~ /:$/){
      $statement .= " : ".parseType(&nextWord);
    }elsif($token =~ /:(\w+\S*)/){
      $statement .= " : ".parseType($1);
    }else{
      $token = &nextWord;
      if($token =~ /:$/){
        $statement .= " : ".parseType(&nextWord);
      }elsif($token =~ /:(\w+\S*)/){
```

```
        $statement .= " : ".parseType($1);
      }else{
        &lastWord;
      }
      $token = $currentWords[$pointer];
    }
    if($token =~ /=$/){
      $statement .= " = ".parseExpression(&nextWord);
    }elsif($token =~ /=(\S+)/){
      $statement .= " = ".parseExpression($1);
    }else{
      $token = &nextWord;
      if($token =~ /=$/){
        $statement .= " = ".parseExpression(&nextWord);
      }elsif($token =~ /=(\S+)/){
        $statement .= " = ".parseExpression($1);
      }else{
        die "parse const statement failed!\n";
      }
    }
  }elsif($token eq "return"){
    my $t = parseExpression(&nextWord);
    &lastWord if($t eq "");
    $statement .= "return ".$t;
  }elsif($token =~ /^'\w+'/){
    $statement .= $token;
  }elsif($token =~ /^\w+/){
    $statement .= parseAssignment($token) unless($keyWordsList{$token});
  }
  $layers--;
  $statement;
}
```

where local variable %keyWordsList is used to identify the key words of Lime and defined as:

```
Sparkf-lime...my %keyWordsList = ("abort",  1, "action",  1, "and",       1,
                                  "array",  1, "char",    1, "do",        1,
                                  "begin",  1, "boolean", 1, "case",      1,
                                  "class",  1, "const",   1, "div",       1,
                                  "downto", 1, "else",    1, "end",       1,
                                  "export", 1, "integer", 1, "map",       1,
                                  "false",  1, "final",   1, "if",        1,
                                  "import", 1, "initialization",          1,
                                  "method", 1, "mod",     1, "new",       1,
                                  "nil",    1, "package", 1, "real",      1,
                                  "not",    1, "of",      1, "or",        1,
                                  "private",1, "program", 1, "procedure",1,
                                  "repeat", 1, "return",  1, "set",       1,
                                  "this",   1, "sequence",1, "until",     1,
                                  "skip",   1, "super",   1, "then",      1,
                                  "to",     1, "true",    1, "type",      1,
                                  "var",    1, "when",    1, "while",     1);
```

and local variable $layer is used to count the nest of statement and defined as:

```
Sparkf-lime...my $layers = -1;
```

For actions, the syntax is action ::= “action” identifier statement:

```
Sparkf-lime...sub parseAction{
  my $identifier = $_[0];
  die "parse action failed!\n" if($identifier !~ /^\w+$/);
  my $statement = parseStatementList(&nextWord);
```

```
  my $classBody = hashAdd(\%classList , $currentClass);
  my $actionFields = hashAdd($classBody , "action");
  $actionFields->{$identifier} = $statement
  unless($actionFields->{$identifier});
}
```

For initializations, the syntax is initialization ::= "initialization" formals statement:

```
Sparkf-lime...sub parseInitialization{
  my $token = $_[0];
  my $statement = "";
  my $formals = parseFormal($token);
  if($formals ne ""){
      $token = $currentWords[$pointer];
      if($token =~ /\)(\S+)/){
          $statement = parseStatementList($1);
      }else{
          $statement = parseStatementList(&nextWord);
      }
  }else{
      $formals = "init";
      $statement = parseStatementList($token);
  }
  my $classBody = hashAdd(\%classList , $currentClass);
  my $actionFields = hashAdd($classBody , "initialization");
  $actionFields->{$formals} = $statement
  unless($actionFields->{$formals});
}
```

For labels, the syntax is label ::= 'char':

```
Sparkf-lime...sub parseLabel{
  my @array = ($_[0]);
  my $classBody = hashAdd(\%classList , $currentClass);
  if($classBody->{"label"}){
      my $old = $classBody->{"label"};
      my $find = 0;
      foreach my $v(@$old){
          if($v eq $_[0]){
              $find = 1;
              last;
          }
      }
      push(@$old, $_[0]) unless($find);
  }else{
      $classBody->{"label"} = ($_[0]);
  }
}
```

Lime also support multiple assignment as following:

```
Sparkf-lime...sub parseAssignment{
  my (@designatorList , @expressionList) = ((), ());
  my ($finish , $token , $assignment) = (0, $_[0], "");
  my @statementList = split(/;/, $token);
  $token = $statementList[0];
  $finish = 1 if(($token =~ /\)else$/)||($token =~ /\)else\'/));
  until($finish){
     last if($#statementList > 0);
     my $temp = &nextWord;
     if($temp =~ /^(\w+)/){
       if(($1 eq "end")||($1 eq "else")||
          ($1 eq "until")||($memberList{$1})){
```

```
        &lastWord;
        $finish = 1;
      }else{
        if($token =~ /\w+$/){
          $token.=" ".$temp;
        }else{
          $token.=$temp;
        }
      }
    }else{
      if($token !~ /\w+$/){
        $token.=" ".$temp;
      }else{
        $token.=$temp;
      }
    }
    @statementList = split(/;/, $token);
  }
  $token = $statementList[0];
  if($token =~ /(.+):=(.+)/){
    @designatorList = split(/,/, $1);
    @expressionList = split(/,/, $2);
    for(my $i=0; $i<$#designatorList; $i++){
      $assignment .= parseDesignator($designatorList[$i]).", ";
    }
    $assignment.=parseDesignator($designatorList[$#designatorList])." := ";
    for(my $i=0; $i<$#expressionList; $i++){
      $assignment .= parseExpression($expressionList[$i]).", ";
    }
    $assignment .= parseExpression($expressionList[$#expressionList]);
  }else{
    $assignment .= parseDesignator($token);
  }
  $assignment;
}
```

For actuals, the syntax is actuals ::= "("expression"," expression")":

```
Sparkf-lime...sub parseActuals{
  my $token = $_[0];
  my $actuals = "(";
  if($token =~ /\((.+)/){
    $token = $1;
  }else{
    die "parse actuals statement failed![1]\n";
  }
  while(1){
    if($token =~ /(.+),(\S+)/){
      $actuals .= parseExpression($1).", ";
      $token = $2;
    }elsif($token =~ /(.+),$/){
      $actuals .= parseExpression($1).", ";
      $token = &nextWord;
    }elsif($token =~ /(.+)\)/){
      $actuals .= parseExpression($1);
      last;
    }else{
      die "parse actuals statement failed![2]\n";
    }
  }
  $actuals .= ")";
  $actuals;
}
```

The syntax of name is name ::= identifer "." identifier :

```
Sparkf-lime...sub parseName{
  my ($name, $token) = ("", $_[0]);
  if($token =~ /^\w+$/){
     $name = $token;
     $token = &nextWord;
  }elsif($token =~ /^(\w+)(\S+)/){
     $name = $1;
     $token = $2;
  }else{
     die "parse name failed!\n";
  }
  while(1){
     if($token =~ /^\./){
        if($token =~ /\.(\w+)$/){
           $token = &nextWord;
           $name .= ".".$1;
        }elsif($token =~ /\.(\w+)(\S+)/){
           $token = $2;
           $name .= ".".$1;
        }else{
           die "parse name failed!\n";
        }
     }else{
        &lastWord;
        last;
     }
  }
  $name;
}
```

The syntax of idList is idList ::= identifier "," identifier:

```
Sparkf-lime...sub parseIdList{
  my @idList = ();
  my $token = $_[0];
  my $finish = 0;
  if($token =~ /^(\w+)$/){
     push(@idList, $1);
     $token = &nextWord;
  }elsif($token =~ /^(\w+)(,\S*)/){
     push(@idList, $1);
     $token = $2;
  }elsif($token =~ /^(\w+)/){
     push(@idList, $1);
     $finish = 1;
  }else{
     die "parse idList failed![1]\n";
  }
  until($finish){
     if($token =~ /^,(\w+)$/){
        $token = &nextWord;
        push(@idList, $1);
     }elsif($token =~ /^,$/){
        $token = ",".&nextWord;
     }elsif($token =~ /^,(\w+)(,\S*)/){
        $token = $2;
        push(@idList, $1);
     }elsif($token =~ /^,(\w+)(.+)/){
        $finish = 1;
        push(@idList, $1);
     }else{
        &lastWord;
        $finish = 1;
     }
  }
  @idList;
```

```
}
```

For base, the syntax is base ::= ["inherit" name | "extend" name]:

```
Sparkf-lime...sub parseBase{
  my $classBody = hashAdd(\%classList, $currentClass);
  $classBody->{$_[0]} = &nextWord;
}
```

For implement, the syntax is implement ::= "implement" name:

```
Sparkf-lime...sub parseImplement{
  my @array = ();
  my $token = $_[0];
  while(1){
    while($token =~ /(\w+),(\S+)/){
      push(@array, $1);
      $token = $2;
    }
    if($token =~ /(\w+),/){
      push(@array, $1);
      $token = &nextWord;
    }else{
      push(@array, $token);
      $token = &nextWord;
      if($token =~ /^,(\S+)/){
        $token = $1;
      }elsif($token =~ /^,/){
        $token = &nextWord;
      }else{
        &lastWord;
        last;
      }
    }
  }
  die "parse implement failed!\n[2]" if($#array < 0);
  my $classBody = hashAdd(\%classList, $currentClass);
  if($classBody->{"implement"}){
    my $old = $classBody->{"implement"};
    my $find = 0;
    foreach my $v(@array){
      foreach my $w(@$old){
        if($v eq $w){
          $find = 1;
          last;
        }
      }
      $find ? $find = 0 : push(@$old, $v);
    }
  }else{
    $classBody->{"implement"} = \@array;
  }
}
```

All the parsing result of code blocks will be inserted into the class list:

```
Sparkf-lime...my %classList = ();
```

The second process in main is outputting the parsing result.

```
Sparkf-lime...sub doOutput{
   generateCode("");
   &insertCode;
   &createModFile;
```

```
}
```

Module *generateCode* is responsible for generating the actual program code files for compiler using the information in the variable %classList and defined as:

```
Sparkf-lime...sub generateCode{
  my $outFile;
  my @keyList = keys(%classList);
  foreach my $v(@keyList){
    my $classBody = $classList{$v};
    if($_[0] eq ""){
      open $outFile, "> $v.lime" || die "Create file failed!\n";
    }else{
      $outFile = $_[0];
    }
    print $outFile "final " if($classBody->{"final"});
    print $outFile "class $v ";
    if($classBody->{extend}){
      print $outFile "extend $classBody->{extend} ";
    }elsif($classBody->{inherit}){
      print $outFile "inherit $classBody->{inherit} ";
    }
    if($classBody->{implement}){
      print $outFile "implement ";
      my $first = 1;
      my $memberBody = $classBody->{implement};
      foreach my $w(@$memberBody){
        if($first){
          print $outFile "$w";
          $first = 0;
        }else{
          print $outFile ", $w";
        }
      }
    }
    print $outFile "\n";
    my @memberList = keys(%$classBody);
    foreach my $u(@memberList){
      my $memberBody = $classBody->{$u};
      unless(($u eq "implement")||($u eq "inherit")||
             ($u eq "extend")||($u eq "final")){
        my @fieldList = keys(%$memberBody);
        foreach my $f(@fieldList){
          if($u eq "var"){
            print $outFile "  var $f : $memberBody->{$f}\n\n";
          }elsif($u eq "const"){
            if($memberBody->{$f} =~ /(\d+)\.(.+)/){
              print $outFile "  ";
              print $outFile "public " if($1);
              print $outFile "const $f$2\n";
            }
          }elsif($u eq "initialization"){
            print $outFile "  initialization ";
            print $outFile "$f" if($f ne "init");
            print $outFile "\n";
            print $outFile "$memberBody->{$f}\n"
            if($memberBody->{$f} ne "");
          }elsif($u eq "action"){
            print $outFile "  action $f\n";
            print $outFile "$memberBody->{$f}\n"
            if($memberBody->{$f} ne "");
          }elsif($u eq "method"){
            if($f =~ /(\w+)\./){
              print $outFile "  method $1";
              print $outFile "$1" if($f =~ /\w+\.(.+)/);
              print $outFile " : $memberBody->{$f}->{TYPE}"
```

```
                    if($memberBody->{$f}->{TYPE} ne "");
                    print $outFile "\n";
                    print $outFile "$memberBody->{$f}->{STATEMENT}"
                    if($memberBody->{$f}->{STATEMENT} ne "");
                    print $outFile "\n";
                }
            }
        }#end of foreach
    }#end of unless
  }
  print $outFile "end\n\n";
  close $outFile if($_[0] eq "");
 }#end of foreach
}
```

After generating code files, front end will insert a copy of code to the origin file, if it find the special tag-pair, "//CODE LIST BEGIN" and "//CODE LIST END", by module *insertCode*, which is defined as:

```
Sparkf-lime...sub insertCode{
  my $switch = 0;
  open my $oldFile, "< $old" || die "Can't open $old: $!";
  open my $newFile, "> $new" || die "Can't open $new: $!";
  while(<$oldFile>){
    if($_ =~ /^\/\/CODE LIST BEGIN/){
      print $newFile $_;
      print $newFile "----\n";
      $switch = 1;
      generateCode($newFile);
    }elsif($_ =~ /^\/\/CODE LIST END/){
      print $newFile "----\n";
      print $newFile $_;
      $switch = 0;
    }else{
      print $newFile $_ unless($switch);
    }
  }
  close $oldFile || die "Can't close $old: $!";
  close $newFile || die "Can't close $old: $!";
  unlink($old);
  rename($new, $old) || die "Can't rename $old: $!";
}
```

In fact, this module creates a file named by the variable $new, which is defined as:

```
Sparkf-lime...my $new = "temp";
```

and then copy the content other than the part between the code list tags into this new file, and after that insert the content of %classList into this new file, and finally deletes the old file and renames the new file with the old name.

Module *createModFile* is responsible for creating a scripts file according to the content of variable @diagramList. If it is empty, nothing will happen; otherwise, front end will create a file with the name defined by variable $filename, which is defined as:

```
Sparkf-lime...my $filename = "oolp.mod";
```

to describe the diagrams used in this program.

```
Sparkf-lime...sub createModFile{
  open FILE, "> $filename"||die "Open file failed!";
  foreach my $v(@diagramList){
    print FILE "\@CLASSDIAGRAM $v\n";
  }
  print FILE "\n";
  my @keyList = keys(%classList);
  foreach my $v(@keyList){
     my $classBody = $classList{$v};
     print FILE "\@CLASS $v\n";
     print FILE "\@FINAL\n" if($classBody->{"final"});
     if($classBody->{extend}){
        print FILE "\@EXTEND $classBody->{extend}\n";
     }elsif($classBody->{inherit}){
        print FILE "\@INHERIT $classBody->{inherit}\n";
     }
     if($classBody->{implement}){
        print FILE "\@IMPLEMENT ";
        my $sep = "";
        my $memberBody = $classBody->{implement};
        foreach my $w(@$memberBody){
           print FILE "$sep$w";
           $sep = "," if($sep eq "");
        }
        print FILE "\n";
     }
     my @memberList = keys(%$classBody);
     foreach my $u(@memberList){
        my $memberBody = $classBody->{$u};
        unless(($u eq "implement")||($u eq "inherit")||
              ($u eq "extend")||($u eq "final")){
           my @fieldList = keys(%$memberBody);
           foreach my $f(@fieldList){
              if($u eq "var"){
                 print FILE "\@VAR $f $memberBody->{$f}\n";
              }elsif($u eq "const"){
                 if($memberBody->{$f} =~ /(\d+)\.(.+)/){
                    print FILE "\@CONST ";
                    print FILE "\@PUBLIC " if($1);
                    print FILE "$f$2\n";
                 }
              }elsif($u eq "initialization"){
                 print FILE "\@INIT ";
                 print FILE "$f" if($f ne "init");
                 print FILE "\n";
              }elsif($u eq "action"){
                 print FILE "\@ACTION $f\n";
              }elsif($u eq "method"){
                 if($f =~ /(\w+)\./){
                    print FILE "\@METHOD $1 ";
                    print FILE "$1 " if($f =~ /\w+\.(.+)/);
                    print FILE "$memberBody->{$f}->{TYPE}"
                    if($memberBody->{$f}->{TYPE} ne "");
                    print FILE "\n";
                 }
              }
           }#end of foreach
        }#end of unless
     }#end of foreach
     print FILE "\n";
  }
  close FILE || die "Close $filename failed!";
```

```
}
```

The third process in main is to call the back end of Spark. It is simply defined as:

```
Sparkf-lime...sub callBackEnd{
   system "perl sparke.pl";
}
```

5.4 Back End

The back end of Spark takes the graphic notation describer mentioned above as input and produce all the graphic notation files as output. However, layout algorithm is out of the range of this thesis, so we choose an automatic diagram layout tool, Graphviz, to fulfill this task. Graphviz is a package of open source tools initiated by AT&T Research Labs for drawing graphs specified in dot language scripts. Now, let us look at how the back end works.

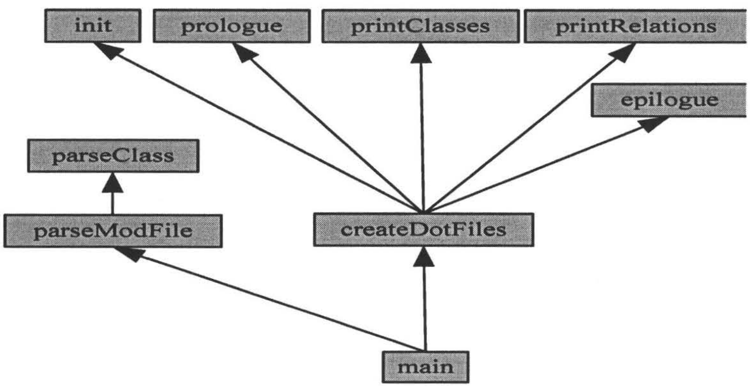


Figure 5.4: Top-level structure for Spark back end

The entrance of Spark back end for lime,

```
Sparkb...&main;
```

is defined as following:

```
Sparkb...sub main{
   &parseModFile;
   &createDotFiles;
}
```

The first process in main is to parse the script file created by front end and defined as:

```
Sparkb...sub parseModFile{
   open FILE, "< $filename" || die "Open file failed!\n";
   while(<FILE>){
      if($_ =~ /^\@CLASSDIAGRAM/){
         push(@diagramList, $_);
      }elsif($_ =~ /^\@CLASS/){
         parseClass($_);
      }
   }
   close FILE || die "Close file failed!\n";
   unlink($filename);
}
```

where local variable $filename hold the name of script file and defined as:

```
Sparkb...my $filename = "oolp.mod";
```

and variable @diagramList keeps all the information about diagrams to be created and defined as:

```
Sparkb...my @diagramList = ();
```

Module *parseClass* is used to retrieve all the class information by reconstructing the class list %classList, which is defined as:

```
Sparkb...my %classList = ();
```

and *parseClass* is defined as:

```
Sparkb...sub parseClass{
  chop($_[0]);
  my @words = split(/\s+/, $_[0]);
  my $name = $words[1];
  my %newHash = ();
  while(<FILE>){
     last if($_ eq "\n");
     chop;
     @words = split;
     if($words[0] eq "\@VAR"){
        unless($newHash{var}){
           my %hash = ();
           $newHash{var} = \%hash;
        }
        my $type = $words[2];
        for(my $i=3; $i<=$#words; $i++){
           $type .= " $words[$i]";
        }
        $newHash{var}->{$words[1]} = $type;
     }elsif($words[0] eq "\@ACTION"){
        unless($newHash{action}){
           my @array = ();
           $newHash{action} = \@array;
        }
        my $body = $newHash{action};
        push(@$body, $words[1]);
     }elsif($words[0] eq "\@METHOD"){
        unless($newHash{method}){
           my %hash = ();
```

```
            $newHash{method} = \%hash;
        }
        if($#words == 3){
            $newHash{method}->{$words[1]."".$words[2]} = $words[3];
        }elsif($#words == 1){
            $newHash{method}->{$words[1]} = "";
        }elsif($#words == 2){
            if($words[2] =~ /\(.+\)/){
                $newHash{method}->{$words[1]."".$words[2]} = "";
            }else{
                $newHash{method}->{$words[1]} = $words[2];
            }
        }
    }elsif($words[0] eq "\@INIT"){
        unless($newHash{init}){
            my @array = ();
            $newHash{init} = \@array;
        }
        my $body = $newHash{init};
        if($#words == 1){
            push(@$body, $name."".$words[1]);
        }elsif($#words == 0){
            push(@$body, $name);
        }
    }elsif($words[0] eq "\@IMPLEMENT"){
        unless($newHash{implement}){
            my @array = ();
            $newHash{implement} = \@array;
        }
        my @temp = split(/,/, $words[1]);
        my $body = $newHash{implement};
        while($#temp >= 0){
            push(@$body, shift(@temp));
        }
    }elsif($words[0] eq "\@EXTEND"){
        $newHash{extend} = $words[1];
    }elsif($words[0] eq "\@INHERIT"){
        unless($newHash{inherit}){
            my @array = ();
            $newHash{inherit} = \@array;
        }
        my @temp = split(/,/, $words[1]);
        my $body = $newHash{inherit};
        while($#temp >= 0){
            push(@$body, shift(@temp));
        }
    }elsif($words[0] eq "\@CONST"){}
  }
  $classList{$name} = \%newHash;
}
```

The second process in main is to *createDotFiles*, which is defined as:

```
Sparkb...sub createDotFiles{
  foreach my $v(@diagramList){
    my @diagram = split(/\s+/, $v);
    if($diagram[1] =~ /(\w+)\.(\w+)/){
      &init;
      for(my $i=2; $i<=$#diagram; $i++){
        if($diagram[$i] =~ /\@VERTICAL/){
          $isHorizontal = 0;
        }elsif($diagram[$i] =~ /\@BRIEF/){
          $isBrief = 1;
        }elsif($diagram[$i] =~ /\@CONCISE/){
          $isConcise = 1;
        }elsif($diagram[$i] =~ /\@HEAD/){
          $isHead = 1;
```

```
        }elsif($diagram[$i] =~ /\@METHOD/){
            $isMethodOnly = 1;
        }elsif($diagram[$i] =~ /\@ATTRIBUTE/){
            $isAttributeOnly = 1;
        }elsif($diagram[$i] =~ /(\w+)/){
            push(@classesAdded, $diagram[$i]) unless(&findClass($diagram[$i]));
        }
    }
    next if($#classesAdded <0);
    open my $out, "> $1.dot" || die "Create file failed!\n";
    prologue($out);
    printClasses($out);
    printRelations($out);
    epilogue($out);
    system "dot -T$2 -o$1.$2 $1.dot";
    unlink("$1.dot");
  }
 }
}
```

For each item in @diagramList, it will create one dot file for GraphViz and one dot file is corresponding to one picture. The supporting function *findClass* is used to determine if the class will appear in this diagram and defined as:

```
Sparkb...sub findClass{
  my $found = 0;
  foreach my $v(@classesAdded){
     if($v eq $_[0]){
         $found = 1;
         last;
     }
  }
  $found;
}
```

Every diagram is new, so the back end clears the environment first every time. The supporting function *init* is defined as:

```
Sparkb...sub init{
  (@classesAdded, @relationsAdded) = ((), ());
  ($isHorizontal, $isBrief, $isConcise, $isHead) = (1, 0, 0, 0);
  ($isMethodOnly, $isAttributeOnly, $isActionOnly) = (0, 0, 0);
}
```

where the variable @classesAdded is used to hold all the classes, which are going to appear in the current diagram and defined as:

```
Sparkb...my @classesAdded = ();
```

and the variable @relationsAdded is used to describe all the relations between these classes and defined as:

```
Sparkb...my @relationsAdded = ();
```

and the variable $isHorizontal is used to describe the direction of this diagram and its default value is 1, which means that the diagram will be drawn horizontally.

```
Sparkb...my $isHorizontal = 1;
```

and the variable $isBrief, $isConcise and $isHead are used to describe the form of this diagram and defined as:

```
Sparkb...my ($isBrief, $isConcise, $isHead) = (0, 0, 0);
```

and the variable $isMethodOnly, $isAttributeOnly, and $isActionOnly are used to indicate which part of the classes in the diagram will be shown and defined as:

```
Sparkb...my ($isMethodOnly, $isActionOnly, $isAttributeOnly) = (0, 0, 0);
```

After initialization and necessary settings, the back end begins to build the dot file basing on the obtained data.

First of all, it create a prologue as:

```
Sparkb...sub prologue{
  my $out = $_[0];
  print $out "# Class diagram\n# Generated by Spark version 1.0\n\n".
             "digraph G {\n\tedge [fontname=\"".$edgeFontName.
             "\", fontsize=10, labelfontname=\"".$edgeFontName.
             "\", labelfontsize=10];\n\tnode [fontname=\"".$nodeFontName.
             "\", fontsize=10, shape=plaintext];\n";
  print $out "\trankdir=LR;\n\tranksep=1;\n" if($isHorizontal);
  print $out "\tbgcolor=\"" + $bgColor + "\";\n" if($bgColor =~ /.+/);
}
```

where the variable $bgColor is defined as:

```
Sparkb...my $bgColor = "";
```

Then, back end begins to print classes listed in @classesAdded as following:

```
Sparkb...sub printClasses{
  my $out = $_[0];
  foreach my $v(@classesAdded){
     print $out "\t".$v." [label=";
     externalTableStart($out);
     #show head
     innerTableStart($out);
     tableLine($out, "CENTER", "interface") if($isInterface);
     tableLine($out, "CENTER", $v);
     innerTableEnd($out);
     #add reuse relations
     my $classBody = $classList{$v};
     if($classBody->{implement}){
        my $memberBody = $classBody->{implement};
        foreach my $w(@$memberBody){
           push(@relationsAdded, "$v.implement.$w\n");
        }
     }
     if($classBody->{inherit}){
        my $memberBody = $classBody->{inherit};
        foreach my $w(@$memberBody){
           push(@relationsAdded, "$v.inherit.$w\n") if(&findClass($w));
        }
     }
     if($classBody->{extend}){
```

```
        push(@relationsAdded, "$v.extend.$classBody->{extend}\n")
        if(&findClass($classBody->{extend}));
    }
    if($classBody->{var}){
        my $memberBody = $classBody->{var};
        my @varList = keys(%$memberBody);
        foreach my $w(@varList){
            my $type = $memberBody->{$w};
            if($type =~ /^(.+)\s+\@LIST\s+(\w+)/){
                push(@relationsAdded, "$v.haslist.$2.$w\n") if(&findClass($2));
            }else{
                push(@relationsAdded, "$v.has.$type.$w\n") if(&findClass($type));
            }
        }
    }
    unless($isHead){
        #show Attributes
        my $displayed = 0;
        innerTableStart($out);
        unless($isMethodOnly || $isActionOnly){
            if($classBody->{var}){
                my $memberBody = $classBody->{var};
                my @varList = keys(%$memberBody);
                foreach my $w(@varList){
                    $displayed |= attribute($out, $w, $memberBody->{$w});
                }
            }
        }
        tableLine($out, "LEFT", " ") unless($displayed);
        innerTableEnd($out);
        #show operation
        innerTableStart($out);
        $displayed = 0;
        unless($isAttributeOnly || $isActionOnly){
            if($classBody->{init}){
                $displayed = 1;
                my $memberBody = $classBody->{init};
                foreach my $w(@$memberBody){
                    if($w =~ /(.+)\.(.+)/){
                        operation($out, $1, $2, "");
                    }else{
                        operation($out, $w, "()", "");
                    }
                }
            }
            if($classBody->{method}){
                $displayed = 1;
                my $memberBody = $classBody->{method};
                my @methodList = keys(%$memberBody);
                foreach my $w(@methodList){
                    my $type = $memberBody->{$w};
                    my ($para, $name) = ("", "");
                    if($w =~ /(.+)\.(.+)/){
                        ($name, $para) = ($1, $2);
                    }else{
                        $name = $w;
                    }
                    operation($out, $name, $para, $type);
                }
            }
        }
        tableLine($out, "LEFT", " ") unless($displayed);
        innerTableEnd($out);
        #show action
        innerTableStart($out);
        $displayed = 0;
        unless($isAttributeOnly || $isMethodOnly){
            if($classBody->{action}){
```

```
                $displayed = 1;
                my $memberBody = $classBody->{action};
                foreach my $w(@$memberBody){
                    operation($out, $w, "()", "");
                }
            }
        }
        tableLine($out, "LEFT", " ") unless($displayed);
        innerTableEnd($out);
    }#end of unless($isHead)
    externalTableEnd($out);
    nodeProperties($out);
  }
}
```

where the variable $isInterface is defined as:

```
Sparkb...my $isInterface = 0;
```

The supporting function *externalTableStart* is used to draw the start border of class diagram and defined as:

```
Sparkb...sub externalTableStart{
  my ($out, $bgcolor) = ($_[0], "");
  $bgcolor = " bgcolor=\"".$nodeFillColor."\"" if($nodeFillColor ne "");
  print $out "<<table border=\"0\" cellborder=\"1\" cellspacing=\"0\" ".
            "cellpadding=\"2\" port=\"p\"".$bgcolor.">".$linePostfix;
}
```

where the variable $nodeFillColor is defined as:

```
Sparkb...my $nodeFillColor = "";
```

and function *innerTableStart* is used to draw the inner border of class diagram and defined as:

```
Sparkb...sub innerTableStart{
  my $out = $_[0];
  print $out $linePrefix.$linePrefix."<tr><td><table border=\"0\" ".
            "cellspacing=\"0\" cellpadding=\"1\">".$linePostfix;
}
```

where the variable $linePrefix and $linePostfix are defined as:

```
Sparkb...my ($linePrefix, $linePostfix) = ("\t", "\n");
```

and function *tableLine* is used to draw a common line of class diagram and defined as:

```
Sparkb...sub tableLine{
  my $out = $_[0];
  my ($topen, $tclose) = ("", "</td></tr>");
  my $prefix = $linePrefix.$linePrefix.$linePrefix;
  if($_[1] eq "CENTER"){
      $topen = $prefix."<tr><td> ";
  }elsif($_[1] eq "LEFT"){
      $topen = $prefix."<tr><td align=\"left\"> ";
  }elsif($_[1] eq "RIGHT"){
```

```
        $topen = $prefix."<tr><td align=\"right\"> ";
    }
    print $out $topen.$_[2].$tclose.$linePostfix;
}
```

where the variable $align is defined as:

```
Sparkb...my $align = "CENTER";
```

and function *innerTableEnd* is used to draw inner end of class diagram and defined as:

```
Sparkb...sub innerTableEnd{
    my $out = $_[0];
    print $out $linePrefix.$linePrefix."</table></td></tr>".$linePostfix;
}
```

and function *externalTableEnd* is used to close class diagram and defined as:

```
Sparkb...sub externalTableEnd{
    my $out = $_[0];
    print $out $linePrefix.$linePrefix."</table>>";
}
```

and function *attribute* is used to display the attributes of class in diagram and defined as:

```
Sparkb...sub attribute{
    my $display = 1;
    if(&findClass($_[2])){
        $display = 0;
    }else{
        if($_[2] =~ /^(.+)\s+\@LIST\s+(\w+)/){
            if(&findClass($2)){
                $display = 0;
            }else{
                my $att = $_[1];
                $att .= " : ".$1 unless($isConcise||$isBrief);
                tableLine($_[0], "LEFT", $att);
            }
        }else{
            my $att = $_[1];
            $att .= " : ".$_[2] unless($isConcise||$isBrief);
            tableLine($_[0], "LEFT", $att);
        }
    }
    $display;
}
```

and function *operation* is used to print operations of class in diagram and defined as:

```
Sparkb...sub operation{
    my ($out, $name, $para, $type) = @_;
    my $cs = $name;
    if($isBrief){
        $cs .="()";
    }else{
        $cs .= $para;
        $type = "" if($isConcise);
        $cs .= " : ".$type if($type ne "");
```

```
    }
    tableLine($out, "LEFT", $cs);
}
```

and function *nodeProperties* is used to print the common properties of the nodes of diagram and defined as:

```
Sparkb...sub nodeProperties{
  my $out = $_[0];
  print $out ", fontname=\"".$nodeFontName."\"".
              ", fontcolor=\"".$nodeFontColor."\"".
              ", fontsize=".$nodeFontSize."];\n";
}
```

where the variable $nodeFontName, $nodeFontColor, and $nodeFontSize are defined as:

```
Sparkb...my ($nodeFontName, $nodeFontColor, $nodeFontSize) = ("arial", "black", 10);
```

After that, back end begins to print the relations listed in @relationsAdded as following:

```
Sparkb...sub printRelations{
  my $out = $_[0];
  foreach my $r(@relationsAdded){
     my ($tailLabel, $headLabel) = ("", "");
     if($r =~ /(\w+)\.extend\.(\w+)/){
        print $out "\t//".$1." extend ".$2."\n"."\t".$2.":p -> ".$1.
           ":p [dir=back,arrowtail=empty,color=\"".$edgeColor."\"];\n";
     }elsif($r =~ /(\w+)\.implement\.(\w+)/){
        print $out "\t//".$1." implement ".$2."\n"."\t".$2.":p -> ".$1.
           ":p [dir=back,arrowtail=empty,style=dashed,color=\"".$edgeColor."\"];\n";
     }elsif($r =~ /(\w+)\.inherit\.(\w+)/){
        print $out "\t//".$1." inherit ".$2."\n"."\t".$2.":p -> ".$1.
           ":p [dir=back,arrowtail=empty,color=\"".$edgeColor."\"];\n";
     }elsif($r =~ /(\w+)\.haslist\.(\w+)\.(\w+)/){
        print $out "\t// ".$1." has a list of ".$2."\n"."\t".$1.":p -> ".$2.":p [".
              "taillabel=\"".$tailLabel."\", "."label=\"".$3."\", "."headlabel=\"".$headLabel.
              "\", "."fontname=\"".$edgeFontName."\", "."fontcolor=\"".$edgeFontColor."\", ".
              "fontsize=".$edgeFontSize.", "."color=\"".$edgeColor."\", ".$associationMap{"list"}."];\n";
     }else{
        $r =~ /(.+)\.(.+)\.(.+)\.(.+)/;
        print $out "\t// ".$1." ".$2." ".$3."\n"."\t".$1.":p -> ".$3.":p [". "taillabel=\"".$tailLabel.
              "\", "."label=\"".$4."\", "."headlabel=\"".$headLabel."\", "."fontname=\"".
              $edgeFontName."\", "."fontcolor=\"".$edgeFontColor."\", "."fontsize=".$edgeFontSize.", ".
              "color=\"".$edgeColor."\", ".$associationMap{$2}."];\n";
     }
  }
}
```

where the variable $edgeFontName, $edgeColor, $edgeFontSize, and $edgeFontColor are defined as:

```
Sparkb...my ($edgeFontName, $edgeColor) = ("arial", "red");
         my ($edgeFontSize, $edgeFontColor) = (10, "black");
```

and the variable %associationMap is defined as:

```
Sparkb...my %associationMap = ( "assoc",      "arrowhead=none",
                            "navassoc",   "arrowhead=open",
                            "has",        "arrowhead=none, arrowtail=ediamond",
                            "composed",   "arrowhead=none, arrowtail=diamond",
                            "list",       "arrowhead=dot, arrowtail=ediamond",
                            "depend",     "arrowhead=open, style=dashed");
```

The last job is to print epilogue as following:

```
Sparkb...sub epilogue{
  my $out = $_[0];
  print $out "}\n";
  close $out;
}
```

5.5 Testing

To verify the design and implementation of Spark, we performed testing following the strategy mentioned in Section 3.6.4.

5.5.1 Usability Testing

Correct Usage

We tried to use Spark with an actual OOLP program file name, leaf.txt, as following:

perl sparkf-lime.pl leaf.txt

The result is that there are three Lime files and four pictures generated by Spark.

No Parameters

We tried to use Spark without parameters as following:

perl sparkf-lime.pl

The result is

Usage: perl sparkf.pl filename

Wrong Parameters

We tried to use Spark with fake file name as following:

```
perl sparkf-lime.pl aaa
```

The result is

```
Cannot open aaa!
```

These three results show that Spark can handle both legal and illegal usages and satisfies the design requirements.

5.5.2 Unit Testing

Syntax Coverage

We composed a sample code program file that coverage all the syntax of Eiffel.

```
Testing of a declaration of a class.
------------------------------------
class STUDENT
end
------------------------------------
Testing of the inheritance relation of a class.
------------------------------------
inherit STUDENT...PEOPLE
------------------------------------
Testing of two features of a class.
------------------------------------
feature {NONE} PEOPLE...name: STRING
         age: INTEGER
------------------------------------
Testing of a deferred class.
------------------------------------
deferred class PEOPLE
end
------------------------------------
Testing of a operation with formal specification and various statements of a class
------------------------------------
feature {NONE}
        STUDENT...set_name(new_name: STRING) is
            local
               a: INTEGER
            do
               name := new_name
               create employees.make
               getup
               if text_file_read.is_connected then
                  split := input_string.split
               end
               if text_file_read.is_connected then
                  split := input_string.split
               else
                  split := input_string.split
               end
               inspect text_file_read.last_string.first.to_upper
```

```
            when 'S' then
                text_file_read.read_line
            when 'B', 'T' then
                text_file_read.read_line
            else
            end
                from  text_file_read.read_line
                until text_file_read.end_of_input
                loop
                    text_file_read.read_line
                end
        end
--------------------------------------------------------------
Testing of a deffered operation with formal specification of a class
--------------------------------------------------------------
feature
    PEOPLE...set_name(n: STRING) is
        require
            n /= Void
        deferred
        end
--------------------------------------------------------------
Testing of a constant of a class
--------------------------------------------------------------
feature {NONE} STUDENT...min_age: INTEGER is 5
--------------------------------------------------------------
Testing of an invariant of a class
--------------------------------------------------------------
invariant
        PEOPLE...age > 0
                 age < 200
end
--------------------------------------------------------------
Testing of an operation with result of a class.
--------------------------------------------------------------
feature {ANY}
      STUDENT...match (n: STRING): BOOLEAN is
        do
            Result := name = n
        end
--------------------------------------------------------------

//CODE LIST BEGIN
----
----
//CODE LIST END
```

After running Spark on this sample,

```
perl sparkf-eiffel.pl coverage.txt
```

we got:

```
Testing of a declaration of a class.
--------------------------------------------------------------
class STUDENT
end
--------------------------------------------------------------
Testing of the inheritance relation of a class.
--------------------------------------------------------------
inherit STUDENT...PEOPLE
--------------------------------------------------------------
Testing of two features of a class.
--------------------------------------------------------------
```

```
feature {NONE} PEOPLE...name: STRING
          age: INTEGER
```

Testing of a deferred class.

```
deferred class PEOPLE
end
```

Testing of a operation with formal specification and various statements of a class

```
feature {NONE}
      STUDENT...set_name(new_name: STRING) is
          local
             a: INTEGER
          do
             name := new_name
             create employees.make
             getup
             if text_file_read.is_connected then
                split := input_string.split
             end
             if text_file_read.is_connected then
                split := input_string.split
             else
                split := input_string.split
             end
             inspect text_file_read.last_string.first.to_upper
             when 'S' then
                 text_file_read.read_line
             when 'B', 'T' then
                 text_file_read.read_line
             else
             end
                 from  text_file_read.read_line
                 until text_file_read.end_of_input
                 loop
                    text_file_read.read_line
                 end
          end
```

Testing of a deffered operation with formal specification of a class

```
feature
    PEOPLE...set_name(n: STRING) is
          require
             n /= Void
          deferred
          end
```

Testing of a constant of a class

```
feature {NONE} STUDENT...min_age: INTEGER is 5
```

Testing of an invariant of a class

```
invariant
       PEOPLE...age > 0
                age < 200
end
```

Testing of an operation with result of a class.

```
feature {ANY}
     STUDENT...match (n: STRING): BOOLEAN is
          do
             Result := name = n
          end
```

```
//CODE LIST BEGIN
-----
class STUDENT
inherit
    PEOPLE
feature {NONE}
    set_name(new_name: STRING) is
        local
            a : INTEGER
        do
            name := new_name
            create employees.make
            getup
            if text_file_read.is_connected then
                split := input_string.split
            end
            if text_file_read.is_connected then
                split := input_string.split
            else
                split := input_string.split
            end
            inspect
                text_file_read.last_string.first.to_upper
            when 'S' then
                text_file_read.read_line
            when 'B', 'T' then
                text_file_read.read_line
            else
            end
            from
                text_file_read.read_line
            until text_file_read.end_of_input
            loop
                text_file_read.read_line
            end
        end
    min_age : INTEGER is 5
feature {ANY}
    match(n: STRING) : BOOLEAN is
        do
            Result := name = n
        end
end

deferred class PEOPLE
feature
    set_name(n: STRING) is
        require
            n /= Void
        deferred
        end
feature {NONE}
    name : STRING
    age : INTEGER
invariant
    age > 0
    age < 200
end

-----
//CODE LIST END
```

The result shows that Spark can parse syntax of Eiffel and generates files correctly.

Diagram Files Generating

We composed a mod file for testing of diagram generation.

```
@CLASSDIAGRAM student1.ps2 PEOPLE STUDENT ATHLETE @VERTICAL
@CLASSDIAGRAM student2.ps2 PEOPLE STUDENT @HEAD
@CLASSDIAGRAM student3.ps2 STUDENT @METHOD
@CLASSDIAGRAM student4.ps2 STUDENT @ATTRIBUTE
@CLASSDIAGRAM student5.ps2 STUDENT @CONCISE
@CLASSDIAGRAM student6.ps2 STUDENT @BRIEF

@CLASS PEOPLE
@METHOD set_name (n:STRING)
@VAR name STRING
@VAR num STRING

@CLASS STUDENT
@INHERIT PEOPLE
@IMPLEMENT ATHLETE
@METHOD match (n:STRING) BOOLEAN
@VAR num INTEGER
@METHOD run

@INTERFACE ATHLETE
@METHOD run
```

According to this mod file, Spark should generate 6 diagrams. Figure 5.5 is drawn vertically and includes all these three classes or interface.

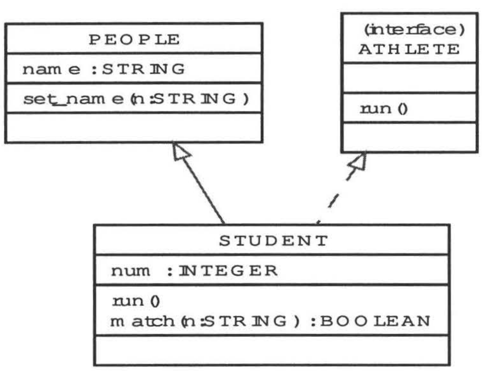


Figure 5.5: Vertical testing

Figure 5.6: Horizontal testing

Figure 5.6 is drawn horizontally and shows the class head only.

Figure 5.7 shows the methods of a class only and figure 5.8 shows the attributes of a class only.

Figure 5.9 and figure 5.10 show classes in concise form and brief form respectively.

The results shows that Spark satisfies the design requirements.

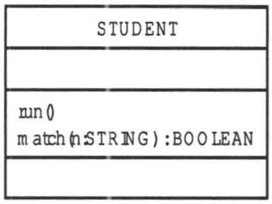


Figure 5.7: Method only testing

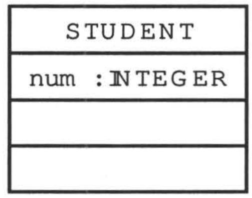


Figure 5.8: Attribute only testing

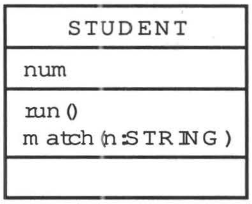


Figure 5.9: Concise form testing

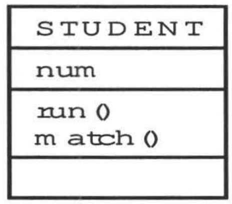


Figure 5.10: Brief form testing

5.5.3 Integration Testing

Our case study itself is perfect integration testing for Spark. The result that program files can be compiled successfully and the graphic notation files is successfully included in the documentation (see Chapter 4) shows that Spark satisfies the design requirements.

5.5.4 System Testing

We conducted all testing mentioned above on MS Windows and Macintosh. The same results show that Spark is platform-independent and Spark satisfies the design requirements.

Chapter 6

Conclusion and Future Work

In this thesis, we presented a new programming paradigm, object-oriented literate programming, which combines several existed significant ideas and is used to construct object-oriented programs in literature style. A set of software tools, Spark, is implemented to support this technique. So far the implementation of Spark altogether contains about hand-written 5000 lines and consists of the following four parts:

- The front end of Spark for Lime (1200 lines).
- The front end of Spark for Eiffel (2500 lines).
- The front end of Spark for perl (800 lines).
- The back end of Spark (500 lines).

Chapter 3 introduced OOLP and all features of Spark, from which we can see how programmers can enjoy the freedom of choosing the combination of languages to develop their software. Chapter 4 gave a case study, Transit Information System, implemented with this technique. It turns out that object-oriented software applications can be expressed in literate style well. In addition, programmers do not need to worry about the graphical notations as well as any extra cost spent on training and tools. Of course, more complex examples are still needed to validate this technique.

Spark itself needs further development to make it more complete and useful. One of the most important things is to build more front ends for Spark in order to adapt more object-oriented programming languages, such as Java and C#.

More graphical notations are need to be supported. So far, Spark can only generate class diagrams. In fact, further development could let Spark have the ability to produce dynamic diagrams, such as sequence diagram and statechart, which can make software documentation more expressive and more complete.

Appendix A

Installation

In order to run the study case presented in this thesis it is necessary to get hold of the following five separate tools. All of them have setup programs as well as installation instructions that can be found on the Web.

A.1 Perl

Perl is a dynamic programming language created by Larry Wall. As an open source software, every body can download its latest version for free from *www.perl.com/download.csp.*

Perl is necessary, because Spark is developed entirely in this language. The version 5.8.8.820 is employed in the testing of Spark.

A.2 Graphviz

Graphviz is a package of open source tools initiated by AT&T Research Labs for drawing graphs specified in DOT language scripts. Since it is free software licensed under the Common Public License, every one can download it for free from *www.ryandesign.com/graphviz* (for Mac OS) and from *www.graphviz.org/Download_windows.php* (for MS Windows)

Graphviz is necessary, because the back end of Spark depends on it. The version 2.12 is employed in the testing of Spark.

A.3 AsciiDoc

AsciiDoc is a text document format for writing short documents, articles, books and UNIX man pages. Its files can be translated to HTML and DocBook markups easily. Free use of AsciiDoc is granted under the terms of the GNU General Public License, so every one can download the latest version for free from *www.methods.co.nz/asciidoc/downloads.html.*

AsciiDoc is necessary, because it is used as the document formatting language in the study case. The version 8.2.1 is employed in the testing of Spark.

A.4 Python

Python is a dynamic object-oriented programming language. As an OSI certified open source software, every body can download its latest version for free from *www.python.org/download.*

Python is necessary, because AsciiDoc depends on it. The version 2.5.1 is employed in the testing of Spark.

A.5 SmartEiffel

SmartEiffel is a free Eiffel compiler. It has been developed at the Lorraine Laboratory of Research in Information Technology and its Applications, an institute affiliated to the French National Institute for Research in Computer Science and Control. SmartEiffel can be downloaded for free from *smarteiffel.loria.fr.*

SmartEiffel is necessary, because Eiffel is chose as the programming language in the study case. The version 2.2 is employed in the testing of Spark.

Appendix B

Source Code of Case Study

```
== The Requirement of Transit Information System
In this project, we are asked to develop an information system for a
local train and bus service. Our customer, HPTA (Happy Passenger
Transit Authority), has no clear picture what it should do, except
to increase customer satisfaction and make traveling more
attractive. All the information we have goes as follows:

.........................
- It will be used by passengers as well as by HPTA staff.
- Selected staff members would be allowed to update the information.
- Passengers should be able to enter their start and destination, a desired time, and get a bunch
  of possible connections.
- Connections can be direct or with changing busses or trains.
- For each bus and train station, the information like opening hours and accessibility is maintained.
- Users can browse a list of all bus and train routes or check the details of a certain route.
- Some bus stops and train stops are conjoint, but some not.
- Trains have two-digit numbers and busses have three-digit numbers.
- Connections between trains and busses must have at least five minutes for the change.
.........................

For simplicity, we assume that detours and delays do not occur, stops are never skipped.

== An Overview
The following picture is an overview of this transit information system. As the root class,
HPTA_TRANSIT_INFO is a subclass of SYSTEM, which is a predefined class in Eiffel and allow its
subclasses to execute system command. Class DATABASE is a deferred class, whose subclasses, such as
class FILE_DATABASE, are responsible for maintaining system data. Class CONNECTION_FINDER is also a
deferred class, whose subclasses, such as class PRIME_FINDER, are responsible for finding the
possible connections.

image::hpta_transit_info.jpg[Class Family]
//$ HPTA_TRANSIT_INFO DATABASE FILE_DATABASE CONNECTION_FINDER PRIME_FINDER ROUTE STAFF STATION KNOT @VERTICAL

The purpose of the application is to maintain the system information, including local train or bus
service and the status of staffs, and provide users current public transit service information,
including possible connections, and routes.

== Dictionary
To understand the main terms used in the requirement, we create a dictionary.
.........................
- passenger: a person, who want to get his or her destination by bus or train.
- staff: a person, who works for HPTA.
- start: a station, where a passenger begin his or her journey.
- destination: a station to which a passenger is going or directed.
```

```
– desired time: an interval, within which one want get to the destination from the start.
– connection: a sequence of stations.
– bus: a long motor vehicle for carrying passengers, usually along a fixed route.
– train: a series of connected railroad cars pulled or pushed by one or more locomotives.
– route: a course for buses or trains to travel from one station to another.
– opening hour: a time, at which the first vehicle departs.
– accessibility: a description of the running status of a station.
– update: a change of system information.
– browse: a display of the information of all routes.
– check: a detail show of a certain route information.
.........................

== Identifying Classes
Basing on the requirements, we defined the classes as follows:

------------------------------------------------------------
class HPTA_TRANSIT_INFO
end
------------------------------------------------------------
HPTA_TRANSIT_INFO is identified as a class of the entire system.

------------------------------------------------------------
class STAFF
feature {NONE}
      number: INTEGER
      password: STRING
end
------------------------------------------------------------
STAFF is a class with attributes employee number and password. The requirements state that selected
staff members would be allowed update the system.
------------------------------------------------------------
class STATION
feature {NONE}
        name: STRING
        open: STRING
        accessibility: STRING
end
------------------------------------------------------------
STATION is a class with attributes name, opening hour, and accessibility.

------------------------------------------------------------
class ROUTE
feature {NONE}
      number: INTEGER
      stops: LINKED_LIST [STATION]
end
------------------------------------------------------------
ROUTE is a class with attributes station list and route number.

== Identifying Operations
All three operations listed in the directory belong naturally in the class HPTA_TRANSIT_INFO,
because they are dependent on the interface of the system.
.........................
– login should belong in class STAFF, because it keep the secret of a certain staff.
.........................

== Consulting The Library of Model
There is no suitable business model in our existing library, so we have to build this system from
the beginning.

== Applying Design Patterns
According to the requirements, our application needs to keep all system information and to calculate
possible connections. There exist so many different methods for these two tasks. Hence, we apply the
strategy design pattern. We declare two deferred classes

------------------------------------------------------------
deferred class DATABASE
end
```

and

```
deferred class CONNECTION_FINDER
end
```

Then, we define two private members for class HPTA_TRANSIT_INFO denoted by the class name followed by three dots as following:

```
feature {NONE}
        HPTA_TRANSIT_INFO...db: DATABASE
        finder: CONNECTION_FINDER
```

i.e.

image::hpta1.jpg[attributes of class HPTA_TRANSIT_INFO]
//$ HPTA_TRANSIT_INFO @ATTRIBUTE
In this way, we can add new algorithms easily and even change mechanisms at runtime with the following private methods:

```
feature {NONE}
        HPTA_TRANSIT_INFO...set_finder(new_finder: CONNECTION_FINDER) is
            require
               new_finder /= Void
            do
               finder := new_finder
            ensure
               finder = new_finder
            end
```

and

```
feature {NONE}
        HPTA_TRANSIT_INFO...set_database(new_database: DATABASE) is
            require
               new_database /= Void
            do
               db := new_database
            ensure
               db = new_database
            end
```

Their preconditions require that the new comers are not invalid and their postconditions ensure that the private member db and finder are set correctly.

Class CONNECTION_FINDER describes the interface that is common to all concrete mechanisms as following:

image::connection.jpg[Class connection_finder]
//$ CONNECTION_FINDER @METHOD

```
feature {HPTA_TRANSIT_INFO}
    CONNECTION_FINDER...
    get_connection(dbase: DATABASE; start, destination: STRING; time: INTEGER): STRING is
           require
               start /= Void
               destination /= Void
               time >= 0
               dbase /= Void
           deferred
           end
```

Class DATABASE describes the interface that is common to all concrete data maintain mechanisms as following:

```
image::database.jpg[Class database]
//$ DATABASE @METHOD
```

```
feature {HPTA_TRANSIT_INFO, CONNECTION_FINDER}
        DATABASE...get_first_bus_route: ROUTE is
             deferred
             end
```

This method can return the first bus route object and is used by class HPTA_TRANSIT_INFO and class CONNECTION_FINDER. Together with the following method, its clients can browse all bus routes one by one.

```
feature {HPTA_TRANSIT_INFO, CONNECTION_FINDER}
        DATABASE...get_next_bus_route(route: ROUTE): ROUTE is
             deferred
             end
```

Similarly, we can browse all train routes by the following two methods:

```
feature {HPTA_TRANSIT_INFO, CONNECTION_FINDER}
        DATABASE...get_first_train_route: ROUTE is
             deferred
             end
```

---

and

```
feature {HPTA_TRANSIT_INFO, CONNECTION_FINDER}
        DATABASE...get_next_train_route(route: ROUTE): ROUTE is
             deferred
             end
```

Browsing all staff information is not necessary, but we need to find given staff object by the following method.

```
feature {HPTA_TRANSIT_INFO, CONNECTION_FINDER}
        DATABASE...find_staff (num: INTEGER): STAFF is
             require
                num > 0
             deferred
             end
```

This method can return an STAFF object, whose employee number equals to the parameter num. It is because all employee number start from 1 that the precondition is added.

For convenience, we also provide a route finding method as follows:

```
feature {HPTA_TRANSIT_INFO, CONNECTION_FINDER}
        DATABASE...find_route (num: INTEGER): ROUTE is
             require
                num >= min_train_route_num
                num <= max_bus_route_num
             deferred
             end
```

The following method is the creation of class DATABASE and invoked by class HPTA_TRANSIT_INFO only.

```
feature {HPTA_TRANSIT_INFO}
        DATABASE...make is
             deferred
             end
```

In order to update system information, class DATABASE also requires the interfaces of adding and deleting as following:

```
feature {HPTA_TRANSIT_INFO}
        DATABASE... add_route(new_route: ROUTE) is
            require
                new_route /= Void
            deferred
            end
```

and

```
feature {HPTA_TRANSIT_INFO}
        DATABASE... delete_route (route: ROUTE) is
            require
                route /= Void
            deferred
            end
```

These two methods can add or delete a certain route to or from this system respectively and is called by class HPTA_TRANSIT_INFO only.

Similarly, class HPTA_TRANSIT_INFO also can add or delete a certain staff by the following two methods:

```
feature {HPTA_TRANSIT_INFO}
        DATABASE... add_staff(new_staff: STAFF) is
            require
                new_staff /= Void
            deferred
            end
```

and

```
feature {HPTA_TRANSIT_INFO}
        DATABASE... delete_staff (staff: STAFF) is
            require
                staff /= Void
            deferred
            end
```

As long as some system information is updated, DATABASE object must be informed to save the change by the following method.

```
feature {HPTA_TRANSIT_INFO}
        DATABASE... do_save is
            deferred
            end
```

According to the requirements, only selected members can update the system. We define that when the database is locked, only the user, who knows both employee number and password, can conduct an update.

```
feature {HPTA_TRANSIT_INFO}
        DATABASE... is_locked: BOOLEAN is
            deferred
            end
```

The subclasses of these two deferred classes implement each concrete behavior mentioned above.

The following four constants are used to point out the bound of route number

```
feature {NONE}
        DATABASE... max_bus_route_num: INTEGER is 999
        min_bus_route_num: INTEGER is 100
```

```
        max_train_route_num: INTEGER is 99
        min_train_route_num: INTEGER is 10
```

== Algorithms Design
=== File Database
For simplicity, we save the system information in a file named "sys_info.txt". So we define a subclass of class DATABASE, FILE_DATABASE as following:

```
class FILE_DATABASE
inherit DATABASE
feature {NONE}
     file_name: STRING is "sys_info.txt"
end
```

i.e.

image::databases.jpg[the hirarchy of databases]
//$ DATABASE FILE_DATABASE @HEAD @VERTICAL

class FILE_DATABASE keep bus routes, train routes and staffs with LINKED_LIST as following:

```
feature {NONE}
        FILE_DATABASE...train_routes: LINKED_LIST [ROUTE]
        bus_routes:    LINKED_LIST [ROUTE]
        employees:     LINKED_LIST [STAFF]
```

now, class FILE_DATABASE becomes:

image::file_database.jpg[attributes of file database]
//$ FILE_DATABASE @ATTRIBUTE

The creation of FILE_DATABASE is method make

```
create FILE_DATABASE...make
```

The main task of make is to initialize this three list

```
feature {HPTA_TRANSIT_INFO}
        FILE_DATABASE...make is
            do
                create employees.make
                create bus_routes.make
                create train_routes.make
                load
            ensure
                employees /= Void
                bus_routes /= Void
                train_routes /= Void
            end
```

and to load the system information for that file:

```
feature {NONE}
   FILE_DATABASE...load is
        local
           input_string : STRING
           text_file_read: TEXT_FILE_READ
           text_file_write: TEXT_FILE_WRITE
           split: ARRAY[STRING]
           new_staff: STAFF
           route: ROUTE
        do
           create text_file_read.connect_to(file_name)
           if text_file_read.is_connected then
               from  text_file_read.read_line
```

```
            until text_file_read.end_of_input
            loop
                if text_file_read.last_string.upper = 1 then
                    inspect text_file_read.last_string.first.to_upper
                    when 'S' then
                        text_file_read.read_line
                        input_string := text_file_read.last_string.twin
                        split := input_string.split
                        create new_staff.make (split.first.to_integer, split.last)
                        employees.add_last(new_staff)
                    when 'B', 'T' then
                        text_file_read.read_line
                        input_string := text_file_read.last_string.twin
                        split := input_string.split
                        route := find_route(split.item(4).to_integer)
                        if route = Void then
                            create route.make(split.item(4).to_integer)
                            route.add_station(split.first, split.item(2), split.item(3), split.last)
                            if split.item(4) .to_integer > max_train_route_num then
                                bus_routes.add_last(route)
                            else
                                train_routes.add_last(route)
                            end
                        else
                            route.add_station(split.first, split.item(2), split.item(3), split.last)
                        end
                    else
                    end
                end
                text_file_read.read_line
            end
            text_file_read.disconnect
        else
            create text_file_write.connect_to(file_name)
            if text_file_write.is_connected then
                text_file_write.disconnect
            end
        end
    end
```

By the following method, one can get the specific route object.

```
feature {HPTA_TRANSIT_INFO, CONNECTION_FINDER}
    FILE_DATABASE... find_route (num: INTEGER): ROUTE is
        local
            i: INTEGER
            route: ROUTE
        do
            if num > max_train_route_num then
                from i := bus_routes.lower
                until i > bus_routes.upper or else bus_routes.item(i) .match(num)
                loop
                    i := i+1
                end
                if i <= bus_routes.upper then
                    route := bus_routes.item(i)
                end
            else
                from i := train_routes.lower
                until i > train_routes.upper or else train_routes.item(i) .match(num)
                loop
                    i := i+1
                end
                if i <= train_routes.upper then
                    route := train_routes.item(i)
                end
            end
            Result := route
```

```
        end
```

Similarly, using the following method, one can get the staff with such employee number:

```
feature {HPTA_TRANSIT_INFO, CONNECTION_FINDER}
        FILE_DATABASE... find_staff (num: INTEGER): STAFF is
            local
                i: INTEGER
                staff: STAFF
            do
                from i := employees.lower
                until i > employees.upper or else employees.item(i) .match(num)
                loop
                    i := i+1
                end
                if i <= employees.upper then
                    staff := employees.item(i)
                end
                Result := staff
            end
```

By the following four methods, one can browse all train routes and bus routes:

```
feature {HPTA_TRANSIT_INFO, CONNECTION_FINDER}
        FILE_DATABASE... get_first_bus_route: ROUTE is
            local
                route: ROUTE
            do
                if not bus_routes.is_empty then
                    route := bus_routes.first
                end
                Result := route
            end
        get_next_bus_route(route: ROUTE): ROUTE is
            require
                bus_routes.index_of(route) > 0
            local
                next_route: ROUTE
            do
                if bus_routes.index_of(route) < bus_routes.upper then
                    next_route := bus_routes.item(bus_routes.index_of(route)+1)
                end
                Result := next_route
            end
        get_first_train_route: ROUTE is
            local
                route: ROUTE
            do
                if not train_routes.is_empty then
                    route := train_routes.first
                end
                Result := route
            end
        get_next_train_route(route: ROUTE): ROUTE is
            require
                train_routes.index_of(route) > 0
            local
                next_route: ROUTE
            do
                if train_routes.index_of(route) < train_routes.upper then
                    next_route := train_routes.item(train_routes.index_of(route)+1)
                end
                Result := next_route
            end
```

By the following method, HPTA_TRANSIT_INFO object can add an arbitrary route to this database

```
feature {HPTA_TRANSIT_INFO}
        FILE_DATABASE... add_route(new_route: ROUTE) is
            do
                if is_bus(new_route.get_number) then
                    bus_routes.add_last(new_route)
                elseif is_train(new_route.get_number) then
                    train_routes.add_last(new_route)
                end
            end
```

By the following method, HPTA_TRANSIT_INFO object can add a staff to this database

```
feature {HPTA_TRANSIT_INFO}
        FILE_DATABASE... add_staff(new_staff: STAFF) is
            do
                employees.add_last(new_staff)
            end
```

By the following method, HPTA_TRANSIT_INFO object can remove an arbitrary route from this database

```
feature {HPTA_TRANSIT_INFO}
        FILE_DATABASE... delete_route (route: ROUTE) is
            do
                if is_bus(route.get_number) then
                    bus_routes.remove(bus_routes.index_of(route))
                elseif is_train(route.get_number) then
                    train_routes.remove(train_routes.index_of(route))
                end
            end
```

By the following method, HPTA_TRANSIT_INFO object can remove a staff from this database

```
feature {HPTA_TRANSIT_INFO}
        FILE_DATABASE... delete_staff (staff: STAFF) is
            do
                employees.remove(employees.index_of(staff))
            end
```

In FILE_DATABASE, as long as employees is not empty, this database is locked, which means the user has to log in before updating.

```
feature {HPTA_TRANSIT_INFO}
        FILE_DATABASE... is_locked: BOOLEAN is
            do
                Result := not employees.is_empty
            end
```

Whenever the database is changed, it have to save the new data to the specific file by the following method:

```
feature {HPTA_TRANSIT_INFO}
        FILE_DATABASE... do_save is
            local
                file_2_write: TEXT_FILE_WRITE
                i: INTEGER
            do
                create file_2_write.connect_to(file_name)
                if file_2_write.is_connected then
                   from i := employees.lower
                   until i > employees.upper
                   loop
                       employees.item(i) .do_save(file_2_write)
                       i := i + 1
                   end
```

```
            from i := bus_routes.lower
            until i > bus_routes.upper
            loop
                bus_routes.item(i) .do_save(file_2_write)
                i := i + 1
            end
            from i := train_routes.lower
            until i > train_routes.upper
            loop
                train_routes.item(i) .do_save(file_2_write)
                i := i + 1
            end
            file_2_write.disconnect
        else
            io.put_string("Update database failed!%N")
        end
    end
```

For convenience, we define the following two methods to tell if the current route is bus or train route:

```
feature {NONE}
    FILE_DATABASE... is_train(num: INTEGER): BOOLEAN is
        do
            Result := num >= min_train_route_num and num <= max_train_route_num
        end
    is_bus(num: INTEGER): BOOLEAN is
        do
            Result := num >= min_bus_route_num and num <= max_bus_route_num
        end
```

=== Conditional Shortest Path
According to the requirement that connections between trains and busses must have at least five minutes for the change, we have to consider bus station and train station as two different stations even they share the same name. In addition, we define a constant change_time in class CONNECTION_FINDER, whose subclasses need it.

```
feature {NONE}
    CONNECTION_FINDER... change_time: INTEGER is 5
```

For convenience, we assume that a bus needs 2 minutes to get to the second stop and a train needs only 1 minute. So we also define the following two members in class CONNECTION_FINDER.

```
feature {NONE}
    CONNECTION_FINDER... train_time: INTEGER is 1
    bus_time: INTEGER is 2
```

PRIME_FINDER is one of the subclasses of CONNECTION_FINDER

```
inherit
      PRIME_FINDER...CONNECTION_FINDER
```

i.e.

image::connection_finder.jpg[hierarchy of class HPTA_TRANSIT_INFO]
//$ CONNECTION_FINDER PRIME_FINDER @HEAD @VERTICAL

Our first algorithm, PRIME_FINDER, is that starting from the start stations, including both bus station and train station, we search for all direct neighbors one after another and calculate their time respectively. In this way, as long as we found the destination as the next neighbor or no more new neighbors before get to the destination, our searching work is done.

To implement this algorithm, we declare list in class PRIME_FINDER

```
feature {NONE}
```

```
    PRIME_FINDER... stop_list: LINKED_LIST [KNOT]
```

Every node of this list record the following information:

image::knot.jpg[attribute of class KNOT]
//$ KNOT @ATTRIBUTE

```
feature {NONE}
    KNOT... station: STATION
```

Form the start down to the destination, as long as the station is found as a valid neighbor, it will be set in a KNOT object by the following method.

```
feature {PRIME_FINDER}
    KNOT... set_station(value: STATION) is
        do

            station := value

        end
```

Of course, class KNOT requires PRIME_FINDER object give a non Void value.

```
feature {PRIME_FINDER}
    KNOT... get_station: STATION is
        do
            Result := station
        end
```

After searching, PRIME_FINDER object can get the record of station by the above method.

```
feature {NONE}
    KNOT... number: INTEGER
```

The number of KNOT object keeps the route number of the station and is set by the following method:

```
feature {PRIME_FINDER}
    KNOT... set_number(value: INTEGER) is
        require
            value >= 0
            value <= 999
        do
            number := value
        end
```

According to the requirement that train route number is a two-digit number and bus route number is a three-digit number, we set a precondition like that for this method.

```
feature {PRIME_FINDER}
    KNOT... get_number: INTEGER is
        do
            Result := number
        end
```

The above method can tell PRIME_FINDER object the route, to which this station belongs.

```
feature {NONE}
    KNOT... time: INTEGER
```

Member time records the total time needed from start and is set by the following method

```
feature {PRIME_FINDER}
    KNOT... set_time(value: INTEGER) is
        require
            value >= 0
        do
```

```
            time := value
        end
```

The time of start node is 0 and the time of destination is desire time plus one, so here KNOT object requires a nonnegative number.

```
feature {PRIME_FINDER}
    KNOT...get_time: INTEGER is
        do
            Result := time
        end
```

The above method is used to provide time for PRIME_FINDE object.

```
feature {NONE}
    KNOT...pred: INTEGER
```

This member is used to record the index of last stop in this list. The pred of start is −1. That the pred of two destination are all −1 means that there is no possible connection between the start and the destination.

PRIME_FINDER object set this member by the following method:

```
feature {PRIME_FINDER}
    KNOT...set_pred(value: INTEGER) is
        do
            pred := value
        end
```

and get the value of this member by the following method:

```
feature {PRIME_FINDER}
    KNOT...get_pred: INTEGER is
        do
            Result := pred
        end
```

Then, how can we judge if this node should be check for new neighbors? we define the member status in class KNOT.

```
feature {NONE}
    KNOT...status: INTEGER
```

If there is no more new neighbors can be found for the current station, this member should be set as permanent, which is a constant of class KNOT;

```
feature {PRIME_FINDER}
    KNOT...permanent: INTEGER is 1
```

otherwise, member status should be set as tentative, which is another constant of class KNOT.

```
feature {PRIME_FINDER}
    KNOT...tentative: INTEGER is 0
```

This member can be set by the following method

```
feature {PRIME_FINDER}
    KNOT...set_status(value: INTEGER) is
        require
            value >= tentative
            value <= permanent
        do
            status := value
        end
```

and get by the following method

```
feature {PRIME_FINDER}
    KNOT...get_status: INTEGER is
        do
            Result := status
        end
```

Method make is the creation of class KNOT

```
creation {PRIME_FINDER}
    KNOT...make
```

and its main task is to initialize this object with the given parameters as following:

```
feature {PRIME_FINDER}
    KNOT...make(sn: STATION; num, t, ss, pr: INTEGER) is
        do
            set_station(sn)
            set_number(num)
            set_time(t)
            set_status(ss)
            set_pred(pr)
        end
```

Every node is added into the list by the following method:

```
feature {NONE}
    PRIME_FINDER...add_node (pr: INTEGER; s: STATION; t, num: INTEGER) is
        require
            t >= 0
        local
            node: KNOT
        do
            create node.make(s, num, t, node.tentative, pr)
            if s = Void then
              node.set_status(node.permanent)
            end
            stop_list.add_last(node)
        end
```

If the station is Void, then the new node will be considered as dead.

The logic of possible connection finding is implemented mainly in the following method.

```
feature {HPTA_TRANSIT_INFO}
      PRIME_FINDER...
      get_connection(dbase: DATABASE; start, destination: STRING; time: INTEGER): STRING is
           require else
              stop_list.upper = 0
           local
              connection, cur_station: STRING
              node: KNOT
              i, monitor: INTEGER
              is_end, break: BOOLEAN
           do
              connection := ""
              desire_time := time
              add_bus_train_station(dbase, destination, desire_time+1)
              add_bus_train_station(dbase, start, 0)

              i := 3
              cur_station := start.twin

              from
              until is_end or else cur_station = Void
```

```
         loop
            monitor := stop_list.upper
            find_neighbor(dbase, cur_station, i)
            if monitor = stop_list.upper then
               if stop_list.item(i) /= Void then
                  stop_list.item(i).set_status(node.permanent)
               end
            end
            is_end := True

            from
            until break or else i > stop_list.upper
            loop
               if stop_list.item(i) /= Void then
                  node := stop_list.item(i)
                  if node.get_status = node.tentative and node.get_station /= Void then
                     cur_station := node.get_station.get_name
                     is_end := False
                     break := True
                  end
               end
               if not break then
                  i := i + 1
               end
            end

            if break then
               break := False
            end
         end

         connection := get_connection_mes(1)
         connection := connection + get_connection_mes(2)

         if connection.same_as("") then
            connection := "There is no connection from your start"
                       + " to your destination in such time."
         end

         Result := connection
      ensure
         Result /= Void
      end
```

The first parameter provides the source of data; the second and third parameters are the names of start station and destination station respectively; the last parameter is the desire time, which will be used to set the private member desire_time:

```
feature {NONE}
   PRIME_FINDER... desire_time: INTEGER
```

At the beginning of searching, we initialize the stop_list of a PRIME_FINDER object with four nodes, i.e. bus and train stations of destination followed by bus and train stations of start, using the following method:

```
feature {NONE}
   PRIME_FINDER... add_bus_train_station(dbase: DATABASE; name: STRING; time: INTEGER) is
      require
         name /= Void
         time >= 0
      local
         route: ROUTE
         station: STATION
         is_end: BOOLEAN
         num: INTEGER
      do
         route := dbase.get_first_bus_route
```

```
            from
            until is_end or route = Void
            loop
                station := route.get_first_station
                from
                until is_end or station = Void
                loop
                    if name.same_as(station.get_name) then
                        is_end := True
                    end
                    if not is_end then
                        station := route.get_next_station(station)
                    end
                end
                if not is_end then
                    route := dbase.get_next_bus_route(route)
                end
            end
            if not is_end then
              station := Void
            end
            if route /= Void then
              num := route.get_number
            else
              num := 0
            end
            add_node(-1, station, time, num)

            station := Void
            is_end := False
            route := dbase.get_first_train_route
            from
            until is_end or route = Void
            loop
                station := route.get_first_station
                from
                until is_end or station = Void
                loop
                    if name.same_as(station.get_name) then
                        is_end := True
                    end
                    if not is_end then
                        station := route.get_next_station(station)
                    end
                end
                if not is_end then
                    route := dbase.get_next_train_route(route)
                end
            end
            if not is_end then
              station := Void
            end
            if route /= Void then
              num := route.get_number
            else
              num := 0
            end
            add_node(-1, station, time, num)
        end
```

Then from the bus station of start, we try to find its direct neighbor by the following method:

```
feature {NONE}
    PRIME_FINDER... find_neighbor(dbase: DATABASE; sn: STRING; pr: INTEGER) is
        require
            sn /= Void
        local
            cost, index, switch: INTEGER
```

```
        p_node, node: KNOT
        route: ROUTE
        station, last: STATION
        name: STRING
        break: BOOLEAN
    do
        from switch := 0
        until switch > 1
        loop
            if switch = 0 then
               cost := bus_time
            else
               cost := train_time
            end
            if pr >= stop_list.lower and pr <= stop_list.upper then
               p_node := stop_list.item(pr)
            end
            if p_node /= Void then
                if p_node.get_station /= Void then
                   if switch = 0 then
                      if is_train(p_node.get_number) then
                         cost := change_time + cost
                      end
                      route := dbase.get_first_bus_route
                   else
                      if is_bus(p_node.get_number) then
                         cost := change_time + cost
                      end
                      route := dbase.get_first_train_route
                   end
                   from
                   until route = Void
                   loop
                      station := route.get_first_station
                      last := station
                      from
                      until station = Void or break
                      loop
                          name := station.get_name.twin
                          if name /= Void and name.is_equal(sn) then
                             if not last.get_name.is_equal(name) then
                                index := get_index(last, route.get_number)
                                if index >= 0 then
                                   node := stop_list.item(index)
                                   if node.get_station /= Void then
                                       if is_train(node.get_number) then
                                          if node.get_time > p_node.get_time + cost then
                                             node.set_pred(pr)
                                             node.set_time(p_node.get_time + cost)
                                             node.set_number(route.get_number)
                                          end
                                       end
                                   end
                                else
                                   add_node(pr, last, p_node.get_time+cost, route.get_number)
                                end
                             end
                             last := route.get_next_station(station)
                             if last /= Void then
                                index := get_index(last, route.get_number)
                                if index >= 0 then
                                   node := stop_list.item(index)
                                   if node.get_station /= Void then
                                       if is_train(node.get_number) then
                                          if node.get_time > p_node.get_time + cost then
                                             node.set_pred(pr)
                                             node.set_time(p_node.get_time + cost)
                                             node.set_number(route.get_number)
```

```
                                 end
                              end
                           end
                        else
                           add_node(pr, last, p_node.get_time+cost, route.get_number)
                        end
                     end
                     break := True
                  else
                     last := station;
                     station := route.get_next_station(station)
                  end
               end
               break := False
               if switch = 0 then
                  route := dbase.get_next_bus_route(route)
               else
                  route := dbase.get_next_train_route(route)
               end
            end
         end
      end
      switch := switch + 1
   end
end
```

For convenience, we define the following two methods to tell if the current route is train or bus:

```
feature {NONE}
   PRIME_FINDER... is_train(num: INTEGER): BOOLEAN is
      do
         Result := num >= 10 and num <= 99
      end
```

and

```
feature {NONE}
   PRIME_FINDER... is_bus(num: INTEGER): BOOLEAN is
      do
         Result := num >= 100 and num <= 999
      end
```

The following method is used to get the index of a certain station in the list; if the target station is not in the list, −1 will be return.

```
feature {NONE}
   PRIME_FINDER... get_index(s: STATION; num: INTEGER): INTEGER is
      require
         s /= Void

      local
         ind, i: INTEGER
         node: KNOT
         name: STRING
      do
         ind := -1
         from i := stop_list.lower
         until i > stop_list.upper
         loop
            node := stop_list.item(i)
            if node.get_station /= Void then
               name := node.get_station.get_name
               if name.is_equal(s.get_name) then
                  if is_bus(num) and is_bus(node.get_number) then
                     ind := i
                  elseif is_train(num) and is_train(node.get_number) then
                     ind := i
```

```
            end
          end
        end
        i := i + 1
      end
      Result := ind
    end
```

When the searching is done, we can get the information of possible connections by the following method:

```
feature {NONE}
  PRIME_FINDER... get_connection_mes(index: INTEGER): STRING is
    require
      index >= 0
    local
      node: KNOT
      mes: STRING
    do
      mes := ""
      node := stop_list.item(index)
      if node /= Void then
        if node.get_station /= Void then
          if node.get_pred /= -1 and node.get_time <= desire_time then
            mes := "-No." + node.get_number.to_string + "->"
                   + node.get_station.get_name + " in "
                   + node.get_time.to_string + " minutes%N"
            node := stop_list.item(node.get_pred)
            from
            until node = Void or else node.get_station = Void or else node.get_pred = -1
            loop
                mes := "-No." + node.get_number.to_string + "->"
                       + node.get_station.get_name + mes
                node := stop_list.item(node.get_pred)
            end
            if node /= Void then
                if node.get_station /= Void then
                  mes := "%N" + node.get_station.get_name + mes
                end
            else
                mes := ""
            end
          end
        end
      end
      Result := mes
    end
```

The creation of PRIME_FINDER is method make

```
creation {ANY} PRIME_FINDER... make
```

it is defined as following:

```
feature {HPTA_TRANSIT_INFO}
    PRIME_FINDER... make is
      do
        create stop_list.make
      ensure
        stop_list /= Void
      end
```

Now, let us talk about the root class HPTA_TRANSIT_INFO.

image::hpta2.jpg[methods of class HPTA_TRANSIT_INFO]
//$ HPTA_TRANSIT_INFO @METHOD

The creation of class HPTA_TRANSIT_INFO is make

```
create HPTA_TRANSIT_INFO ... make
```

Its main task is to initialize the database and connection finder, and then run the whole system:

```
feature {ANY}
        HPTA_TRANSIT_INFO ... make is
            local
                prime_finder: PRIME_FINDER
                file_database: FILE_DATABASE
            do
                create file_database.make
                set_database(file_database)
                create prime_finder.make
                set_finder(prime_finder)
                run
            end
```

In order to increase customer satisfaction, we run the system by a series of menus

```
feature {NONE}
        HPTA_TRANSIT_INFO ... run is
                do
                        from
                        until io.last_character.to_upper = 'Q'
                        loop
                                menu
                                io.read_character
                                io.put_new_line
                                inspect io.last_character.to_upper
                                when 'U' then do_update
                                when 'I' then do_inquire
                                else
                                end
                        end
                end
```

In order to use OS command, we let class HPTA_TRANSIT_INFO be a subclass of class SYSTEM, which is a predefined class in Eiffel.

```
inherit
     HPTA_TRANSIT_INFO ... SYSTEM
```

Method menu is the main menu of the interface of this system and

```
feature {NONE}
        HPTA_TRANSIT_INFO ... menu is
                do
                                execute_command_line("cls")
                                io.put_string("[
                                     ****************************************
                                               Welcome to HPTA
                                     ****************************************
                                        U Update System Information
                                        I Inquire about Transit Information
                                        Q Quit

                                     Enter menu choice:
                                     ]")
                end
```

This is the main menu and there are two items in it, through which users can either update or inquire system information. The first line of the method body is used to clear the screen.

If users chose the first menu item, they are going to enter the following menu, i.e. update_menu:

```
feature {NONE}
        HPTA_TRANSIT_INFO... update_menu is
                do
                        execute_command_line("cls")
                        io.put_string("[
                             ***************************************
                                       Welcome to HPTA
                             ***************************************
                                A Add
                                D Delete
                                G Go back

                             Enter menu choice:
                             ]")
                end
```

In this menu, users can add new information, such as staffs and stations, as follow:

```
feature {NONE}
        HPTA_TRANSIT_INFO... add_menu is
                do
                        execute_command_line("cls")
                        io.put_string("[
                             ***************************************
                                       Welcome to HPTA
                             ***************************************
                                S Add a station
                                E Add a staff
                                G Go back

                             Enter menu choice:
                             ]")
                end
```

Follows the logic of method do_add:

```
feature {NONE}
        HPTA_TRANSIT_INFO... do_add is
            local
                employee: STAFF
                id: INTEGER
                input, name, password, open, access, last: STRING
                is_end: BOOLEAN
                route: ROUTE
            do
                from
                until is_end
                loop
                   add_menu
                   io.read_line
                   input := io.last_string.twin
                   io.put_new_line
                   if not input.is_empty then
                      inspect input.first.to_upper
                      when 'G' then is_end := True
                      when 'S' then
                            io.put_string("%NEnter station name: ")
                            io.read_line
                            name := io.last_string.twin
                            io.put_string("%NEnter open hour: ")
                            io.read_line
                            open := io.last_string.twin
                            io.put_string("%NEnter its accessibility: ")
```

```
                        io.read_line
                        access := io.last_string.twin
                        io.put_string("%NEnter route number: ")
                        io.read_line
                        id := io.last_string.to_integer
                        io.put_string("%NEnter the name of its last station: ")
                        io.read_line
                        last := io.last_string.twin
                        route := db.find_route(id)
                        if route = Void then
                            create route.make(id)
                            route.add_station(name, access, open, last)
                            db.add_route(route)
                        else
                            route.add_station(name, access, open, last)
                        end
                  when 'E' then
                        io.put_string("%NEnter your ID: ")
                        io.read_line
                        id := io.last_string.to_integer
                        io.put_string("%NEnter your password: ")
                        io.read_line
                        password := io.last_string.twin
                        create employee.make(id, password)
                        db.add_staff(employee)
                  else
                  end
               end
            end
        end
```

they can also delete those information as follow:

```
feature {NONE}
        HPTA_TRANSIT_INFO...delete_menu is
                do
                        execute_command_line("cls")
                        io.put_string("[
                            ***************************************
                                     Welcome to HPTA
                            ***************************************
                               S Delete a station
                               E Delete a staff
                               R Delete a route
                               G Go back

                            Enter menu choice:
                            ]")
                end
```

Follows the logic of method do_delete:

```
feature {NONE}
        HPTA_TRANSIT_INFO...do_delete is
            local
                is_end: BOOLEAN
                num: INTEGER
                staff: STAFF
                route: ROUTE
                input, name: STRING
            do
                from
                until is_end
                loop
                   delete_menu
                   io.read_line
                   input := io.last_string.twin
```

```
            io.put_new_line
            if not input.is_empty then
               inspect input.first.to_upper
               when 'G' then is_end := True
               when 'S' then
                     io.put_string("%NEnter route number: ")
                     io.read_line
                     num := io.last_string.to_integer
                     io.put_string("%NEnter station name: ")
                     io.read_line
                     name := io.last_string.twin
                     route := db.find_route(num)
                     if route /= Void then
                        route.remove_station(name)
                     else
                        io.put_string ("%NNo such a station%N")
                        io.read_line
                     end
               when 'R' then
                     io.put_string("%NEnter route number: ")
                     io.read_line
                     num := io.last_string.to_integer
                     route := db.find_route(num)
                     if route /= Void then
                        db.delete_route(route)
                     else
                        io.put_string ("%NNo such a station%N")
                        io.read_line
                     end
               when 'E' then
                     io.put_string("%NEnter ID: ")
                     io.read_line
                     num := io.last_string.to_integer
                     staff := db.find_staff(num)
                     if staff /= Void then
                        db.delete_staff(staff)
                     else
                        io.put_string ("%NNo such a staff%N")
                        io.read_line
                     end
               else
               end
            end
         end
      end
```

According to the requirement, only authorized staffs can do such things, so this system will ask the user to log in the system before he or she enter the update menu. The following method do_update has the logic to require the user to enter his or her employee number and password first.

```
feature {NONE}
      HPTA_TRANSIT_INFO...do_update is
         local
            id: INTEGER
            passed, is_end: BOOLEAN
            password, input: STRING
            staff: STAFF
         do
            io.read_line
            if db.is_locked then
               io.put_string ("%NEnter employee ID: ")
               io.read_line
               id := io.last_string.to_integer
               staff := db.find_staff(id)
               if staff /= Void then
                  io.put_string ("%NEnter password: ")
```

```
                    io.read_line
                    password := io.last_string.twin
                    passed := staff.login(password)
                end
            else
                io.put_string ("[
                    The list of authorized staff is not empty,
                    so please set authorization as soon as possible...
                    ]")
                passed := True
                io.read_line
            end
            if passed then
                from
                until is_end
                loop
                    update_menu
                    io.read_line
                    input := io.last_string.twin
                    io.put_new_line
                    if not input.is_empty then
                        inspect input.first.to_upper
                        when 'A' then do_add
                        when 'D' then do_delete
                        when 'G' then is_end := True
                        else
                        end
                    end
                end
                db.do_save
            else
                io.put_string ("%NLogin failed!%N")
                io.read_line
            end
        end
```

The actual logging responsibility is assigned to class STAFF as public feature to class HPTA_TRANSIT_INFO:

```
feature {HPTA_TRANSIT_INFO}
    STAFF...login(passwd: STRING): BOOLEAN is
        require
            passwd /= Void
        do
            Result := password.is_equal(passwd)
        end
```

If the result is True, the user can continue his or her update, otherwise, this system will remain on the main menu.

If users chose the second menu item of the main menu, they will enter the following query menu without any bother, because the requirement says that any one can have access to the transit information.

```
feature {NONE}
    HPTA_TRANSIT_INFO...inquire_menu is
        do
            execute_command_line("cls")
            io.put_string("[
                ****************************************
                         Welcome to HPTA
                ****************************************
                    F Find a possible connection
                    S Show a route
                    B Browse all routes
                    G Go back

                Enter menu choice:
                ]")
```

```
            end
```

The first item of this menu is used for users to find a possible connection. Following the logic of method do_inquire, users are required to enter their start, destination, as well as their desire time.

```
feature {NONE}
        HPTA_TRANSIT_INFO ... do_inquire is
            local
                input, start, dest: STRING
                is_end: BOOLEAN
                num, time: INTEGER
                route: ROUTE
            do
                from
                until is_end
                loop
                    inquire_menu
                    io.read_line
                    input := io.last_string.twin
                    io.put_new_line
                    if not input.is_empty then
                        inspect input.first.to_upper
                        when 'B' then
                            from route := db.get_first_bus_route
                            until route = Void
                            loop
                                route.show
                                route := db.get_next_bus_route(route)
                            end
                            from route := db.get_first_train_route
                            until route = Void
                            loop
                                route.show
                                route := db.get_next_train_route(route)
                            end
                            io.put_string ("%N%NStrike any key to continue...")
                            io.read_line
                        when 'F' then
                            io.put_string("%NEnter the station name of your start: ")
                            io.read_line
                            start := io.last_string.twin
                            io.put_string("%NEnter the station name of your destination: ")
                            io.read_line
                            dest := io.last_string.twin
                            io.put_string("%NEnter your desire time(in minutes): ")
                            io.read_line
                            time := io.last_string.to_integer
                            io.put_string(finder.get_connection(db, start, dest, time))
                            io.put_string ("%N%NStrike any key to continue...")
                            io.read_line
                        when 'S' then
                            io.put_string ("Input the route number (10 - 999): ")
                            io.read_line
                            num := io.last_string.to_integer
                            route := db.find_route(num)
                            if route /= Void then
                                route.show
                            else
                                io.put_string ("Sorry there is no such a route")
                            end
                            io.put_string ("%N%NStrike any key to continue...")
                            io.read_line
                        when 'G' then is_end := True
                        else
                        end
                    end
```

```
                end
            end
```

Now, it is time to implement the methods of class ROUTE

image::Route.jpg[class ROUTE]
//$ ROUTE @METHOD

The creation of ROUTE is make, which can be invoke by class HPTA_TRANSIT_INFO

```
creation  ROUTE...make
```

The main task of make is initialize the route number and station list

---

```
feature {HPTA_TRANSIT_INFO}
      ROUTE...make (num: INTEGER) is
          require
            num > 9
            num < 1000
          do
            number := num
            create stops.make
          ensure
            number = num
            stops /= Void
          end
```

According the requirement, route number must be two- or three-digit number, so we define the following invariant for class ROUTE.

```
invariant
        ROUTE...number > 9
        number < 1000
end
```

At any time, its client get route number by the following method:

```
feature {ANY}
     ROUTE...get_number  : INTEGER is
          do
             Result := number
          end
```

---

also, by the following method to tell if the current route is which we want:

```
feature {ANY}
      ROUTE...match (num: INTEGER): BOOLEAN is
            do
               Result := num = number
            end
```

By the following method, its client adds new stations for this ROUTE object and at the same time set the name, the accessibility, the opening hour, and last station for this new station.

```
feature {HPTA_TRANSIT_INFO, DATABASE}
      ROUTE...add_station(new_name, access, open_hour, last_stop: STRING) is
            local
               new_station: STATION
               i: INTEGER
               last: STRING
            do
               from i := stops.lower
               until i > stops.upper or else stops.item(i) .match(new_name)
               loop
                  i := i + 1
               end
```

```
            if i > stops.upper then
               create new_station.make(new_name, access, open_hour)
               last := last_stop.twin
               last.to_upper
               if last.same_as("NONE") then
                  stops.add_first (new_station)
               else
                  from i := stops.lower
                  until i > stops.upper or else stops.item(i) .match(last_stop)
                  loop
                     i := i+1
                  end
                  if i <= stops.upper then
                     stops.add (new_station, i+1)
                  else
                     create new_station.make (last_stop, access, open_hour)
                     stops.add_last (new_station)
                     stops.add_last (new_station)
                  end
               end
            end
         end
```

HPTA_TRANSIT_INFO object removes a certain station by the following method, whose only parameter is the name of the target station.

```
feature {HPTA_TRANSIT_INFO}
      ROUTE... remove_station(name: STRING) is
         local
            i: INTEGER
         do
            from i := stops.lower
            until i > stops.upper or else stops.item(i) .match(name)
            loop
               i := i + 1
            end
            if i <= stops.upper then
               stops.remove(i)
            end
         end
```

The subclasses of CONNECTION_FINDER use the following two methods to visit all stations in this route

```
feature {CONNECTION_FINDER}
      ROUTE... get_first_station: STATION is
         local
            station: STATION
         do
            if stops.upper > 0 then
               station := stops.first
            end
            Result := station
         end
      get_next_station(station1: STATION): STATION is
         require
            station1 /= Void
         local
            station: STATION
         do
            if stops.index_of(station1) < stops.upper then
               station := stops.item(stops.index_of(station1)+1)
            end
            Result := station
         end
```

Class ROUTE keep the secret of saving itself, so DATABASE object can call this method to fulfill the task. Actually, such assignment is worth to discuss. Maybe should move to the subclasses of DATABASE, because only they know exactly how to save those data.

```
feature {DATABASE}
    ROUTE...do_save(file: TEXT_FILE_WRITE) is
        require
            file.is_connected
        local
            i: INTEGER
            tag, last: STRING
        do
            if number > 99 then
                tag := "b"
            else
                tag := "t"
            end
            last := "None"
            from i := stops.lower
            until i > stops.upper
            loop
                file.put_string(tag+"%N")
                stops.item(i).do_save(file)
                file.put_string(" " + number.to_string +" " + last +"%N")
                last := stops.item(i).get_name.twin
                i := i + 1
            end
        end
```

Similarly, the following method is responsible for showing the details of this route, but only class HPTA_TRANSIT_INFO know exactly how to display with interface, so this method should be move to class HPTA_TRANSIT_INFO.

```
feature {HPTA_TRANSIT_INFO}
    ROUTE...show is
        local i: INTEGER
        do
            if number > 99 then
                io.put_string ("%NBus route No.")
            else
                io.put_string ("%NTrain route No.")
            end
            io.put_integer (number)
            io.put_string (": ")
            from i := stops.lower
            until i > stops.upper
            loop
                stops.item(i).show
                if i < stops.upper then
                    io.put_string ("->")
                end
                i := i+1;
            end
            io.put_new_line
        end
```

Same problem can be found on the method show of class STATION

```
feature {ROUTE}
    STATION...show is
        do
            io.put_string (name)
        end
```

Now, let us look at the class STATION, whose creation is method make too,

```
create STATION...make
```

and defined as following:

```
feature {ROUTE}
        STATION... make (new_name, new_open, new_acc: STRING) is
            require
                new_name /= Void
                new_open /= Void
                new_acc /= Void
            do
                name := new_name.twin
                open := new_open.twin
                accessibility := new_acc.twin
            end
```

The main task of it is to initial these three features of class STATION. At any time, its client can visit these three features by the following methods:

```
feature {ROUTE, CONNECTION_FINDER}
        STATION... get_name: STRING is
            do
                Result := name.twin
            end
        get_acc: STRING is
            do
                Result := accessibility.twin
            end
        get_open: STRING is
            do
                Result := open.twin
            end
```

Similar with the method do_save of class ROUTE, this method should be moved into the subclasses of DATABASE.

```
feature {ROUTE}
        STATION... do_save(file: TEXT_FILE_WRITE) is
            require
                file.is_connected
            do
                file.put_string(name + " " + accessibility + " " + open)
            end
```

The same problem can be found on class STAFF

```
feature {DATABASE}
        STAFF... do_save(file: TEXT_FILE_WRITE) is
            require
                file.is_connected
            do
                file.put_string("s%N" + number.to_string + " " + password + "%N")
            end
```

We identify station with name only, i.e. if two stations share the same name, we assume they are the same station. Here case is insensitive.

```
feature {ROUTE}
        STATION... match (targetname: STRING): BOOLEAN is
            require
                targetname /= Void
            do
                Result := name.same_as (targetname)
            end
```

Now, let us talk about the implementation of class STAFF.

image::staff.jpg[class STAFF]

```
//$ STAFF @METHOD

The creation of class STAFF is make

creation{ANY} STAFF...make

it is defined as following:

feature {ANY}
        STAFF...make (id: INTEGER; passwd: STRING) is
            require
                id >= 0
                passwd /=Void
            do
                number := id;
                password := passwd.twin
            ensure
                number >= 0
                password = passwd
            end

its main task is initialize staff's id and password.

Method match is used to identify a certain staff and is defined as following:

feature {ANY}
        STAFF...match (id: INTEGER): BOOLEAN is
            do
                Result := id = number
            end

Any staff has an unique employee number, which is generated from 0, and a password,
which must not be Void:

invariant
        STAFF...number_positive: number >= 0
        password_not_void: password /= Void
end

So far, we have implement the system.

== Testing
=== Updating system
When no staff is authorized, we try to update system information.
The result is

image::empty.jpg[empty]

Otherwise, we try to update system information. The system requires
ID and password for logging in as following:

image::updating.jpg[updating]

These results satisfy the design requirements.

=== Browsing all routes
We try to browse the information of all routes as following:

image::browse.jpg[browse]

These result satisfies the design requirements.

=== Finding connection
We try to find a connection between two stations as following:

image::connection.jpg[connection]
```

```
These result satisfies the design requirements.

=== Strategy pattern
We construct a sample connection finder class and change the
algorithm a run-time.

------------------------------------------------------------
class FAKE_FINDER
inherit
    CONNECTION_FINDER
creation {ANY}
    make
feature {HPTA_TRANSIT_INFO}
    make is
        do
        end
    get_connection(start, destination: STRING; time: INTEGER): STRING is
        do
            Result := "This is a test%N";
        end
end
------------------------------------------------------------

image::strategy.jpg[strategy]

These result satisfies the design requirements.

In order to give an integrated view for ones who
are used to read code, we list all program code here.
```

Appendix C

Generated Code of Case Study

```
class KNOT
creation {PRIME_FINDER}
      make
feature {PRIME_FINDER}
      get_number : INTEGER is
            do
                  Result := number
            end
      make(sn: STATION; num, t, ss, pr: INTEGER) is
            do
                  set_station(sn)
                  set_number(num)
                  set_time(t)
                  set_status(ss)
                  set_pred(pr)
            end
      set_pred(value: INTEGER) is
            do
                  pred := value
            end
      get_time : INTEGER is
            do
                  Result := time
            end
      set_time(value: INTEGER) is
            require
                  value >= 0
            do
                  time := value
            end
      get_pred : INTEGER is
            do
                  Result := pred
            end
      permanent : INTEGER is 1
      get_station : STATION is
            do
                  Result := station
            end
      tentative : INTEGER is 0
      set_station(value: STATION) is


            do
```

```
            station := value
        end
    get_status : INTEGER is
        do
            Result := status
        end
    set_status(value: INTEGER) is
        require
            value >= tentative
            value <= permanent
        do
            status := value
        end
    set_number(value: INTEGER) is
        require
            value >= 10
            value <= 999
        do
            number := value
        end
feature {NONE}
    station : STATION
    number : INTEGER
    status : INTEGER
    time : INTEGER
    pred : INTEGER
end

deferred class DATABASE
feature {HPTA_TRANSIT_INFO}
    do_save is
        deferred
        end
    add_route(new_route: ROUTE) is
        require
            new_route /= Void
        deferred
        end
    make is
        deferred
        end
    delete_staff(staff: STAFF) is
        require
            staff /= Void
        deferred
        end
    delete_route(route: ROUTE) is
        require
            route /= Void
        deferred
        end
    add_staff(new_staff: STAFF) is
        require
            new_staff /= Void
        deferred
        end
    is_locked : BOOLEAN is
        deferred
        end
feature {NONE}
    min_train_route_num : INTEGER is 10
    max_bus_route_num : INTEGER is 999
    max_train_route_num : INTEGER is 99
    min_bus_route_num : INTEGER is 100
feature {HPTA_TRANSIT_INFO, CONNECTION_FINDER}
    find_route(num: INTEGER) : ROUTE is
        require
            num >= min_train_route_num
```

```
            num <= max_bus_route_num
        deferred
        end
    find_staff(num: INTEGER) : STAFF is
        require
            num > 0
        deferred
        end
    get_next_bus_route(route: ROUTE) : ROUTE is
        deferred
        end
    get_first_bus_route : ROUTE is
        deferred
        end
    get_next_train_route(route: ROUTE) : ROUTE is
        deferred
        end
    get_first_train_route : ROUTE is
        deferred
        end
end

deferred class CONNECTION_FINDER
feature {HPTA_TRANSIT_INFO}
    get_connection(dbase: DATABASE; start, destination: STRING; time: INTEGER) : STRING is
        require
            start /= Void
            destination /= Void
            time >= 0
            dbase /= Void
        deferred
        end
feature {NONE}
    train_time : INTEGER is 1
    bus_time : INTEGER is 2
    change_time : INTEGER is 5
end

class STAFF
creation {ANY}
    make
feature {HPTA_TRANSIT_INFO}
    login(passwd: STRING) : BOOLEAN is
        require
            passwd /= Void
        do
            Result := password.is_equal(passwd)
        end
feature {DATABASE}
    do_save(file: TEXT_FILE_WRITE) is
        require
            file.is_connected
        do
            file.put_string("s%N" + number.to_string + "_" + password + "%N")
        end
feature {NONE}
    password : STRING
    number : INTEGER
feature {ANY}
    make(id: INTEGER; passwd: STRING) is
        require
            id >= 0
            passwd /= Void
        do
            number := id
            password := passwd.twin
        ensure
            number >= 0
```

```
            password = passwd
        end
    match(id: INTEGER) : BOOLEAN is
        do
            Result := id = number
        end
invariant
    number_positive : number >= 0
    password_not_void : password /= Void
end

class PRIME_FINDER
inherit
    CONNECTION_FINDER
creation {ANY}
    make
feature {HPTA_TRANSIT_INFO}
    make is
        do
            create stop_list.make
        ensure
            stop_list /= Void
        end
    get_connection(dbase: DATABASE; start, destination: STRING; time: INTEGER) : STRING is
        require else
            stop_list.upper = 0
        local
            connection : STRING
            cur_station : STRING
            node : KNOT
            i : INTEGER
            monitor : INTEGER
            is_end : BOOLEAN
            break : BOOLEAN
        do
            connection := "_"
            desire_time := time
            add_bus_train_station(dbase, destination, desire_time+1)
            add_bus_train_station(dbase, start, 0)
            i := 3
            cur_station := start.twin
            from
            until is_end or else cur_station = Void
            loop
                monitor := stop_list.upper
                find_neighbor(dbase, cur_station, i)
                if monitor = stop_list.upper then
                    if stop_list.item(i) /= Void then
                        stop_list.item(i).set_status(node.permanent)
                    end
                end
                is_end := True
                from
                until break or else i > stop_list.upper
                loop
                    if stop_list.item(i) /= Void then
                        node := stop_list.item(i)
                        if node.get_status = node.tentative and node.get_station /= Void then
                            cur_station := node.get_station.get_name
                            is_end := False
                            break := True
                        end
                    end
                    if not break then
                        i := i + 1
                    end
                end
                if break then
```

```
                break := False
              end
            end
            connection := get_connection_mes(1)
            connection := connection + get_connection_mes(2)
            if connection.same_as("") then
              connection := "_There_is_no_connection_from_your_start" + "_to_your_destination_in_such_time."
            end
            Result := connection
          ensure
            Result /= Void
          end
feature {NONE}
      get_index(s: STATION; num: INTEGER) : INTEGER is
          require
            s /= Void
          local
            ind : INTEGER
            i : INTEGER
            node : KNOT
            name : STRING
          do
            ind := - 1
            from
              i := stop_list.lower
            until i > stop_list.upper
            loop
              node := stop_list.item(i)
              if node.get_station /= Void then
                name := node.get_station.get_name
                if name.is_equal(s.get_name) then
                  if is_bus(num) and is_bus(node.get_number) then
                    ind := i
                  elseif is_train(num) and is_train(node.get_number) then
                    ind := i
                  end
                end
              end
              i := i + 1
            end
            Result := ind
          end
      add_bus_train_station(dbase: DATABASE; name: STRING; time: INTEGER) is
          require
            name /= Void
            time >= 0
          local
            route : ROUTE
            station : STATION
            is_end : BOOLEAN
            num : INTEGER
          do
            route := dbase.get_first_bus_route
            from
            until is_end or route = Void
            loop
              station := route.get_first_station
              from
              until is_end or station = Void
              loop
                if name.same_as(station.get_name) then
                  is_end := True
                end
                if not is_end then
                  station := route.get_next_station(station)
                end
              end
              if not is_end then
```

```
                route := dbase.get_next_bus_route(route)
            end
        end
        if not is_end then
            station := Void
        end
        if route /= Void then
            num := route.get_number
        else
            num := 0
        end
        add_node(-1, station, time, num)
        station := Void
        is_end := False
        route := dbase.get_first_train_route
        from
        until is_end or route = Void
        loop
            station := route.get_first_station
            from
            until is_end or station = Void
            loop
                if name.same_as(station.get_name) then
                    is_end := True
                end
                if not is_end then
                    station := route.get_next_station(station)
                end
            end
            if not is_end then
                route := dbase.get_next_train_route(route)
            end
        end
        if not is_end then
            station := Void
        end
        if route /= Void then
            num := route.get_number
        else
            num := 0
        end
        add_node(-1, station, time, num)
    end
  get_connection_mes(index: INTEGER) : STRING is
    require
        index >= 0
    local
        node : KNOT
        mes : STRING
    do
        mes := "_"
        node := stop_list.item(index)
        if node /= Void then
            if node.get_station /= Void then
                if node.get_pred /= - 1 and node.get_time <= desire_time then
                    mes := "_-No." + node.get_number.to_string + "_->" + node.get_station.get_name + "_in_" +
                    node := stop_list.item(node.get_pred)
                    from
                    until node = Void or else  node.get_station = Void or else
node.get_pred = - 1
                    loop
                        mes := "_-No." + node.get_number.to_string + "_->" + node.get_station.get_name + mes
                        node := stop_list.item(node.get_pred)
                    end
                    if node /= Void then
                        if node.get_station /= Void then
                            mes := "_%N" + node.get_station.get_name + mes
                        end
```

```
                else
                    mes := "_"
                end
            end
        end
    end
    Result := mes
  end
is_train(num: INTEGER) : BOOLEAN is
  do
    Result := num >= 10 and num <= 99
  end
add_node(pr: INTEGER; s: STATION; t, num: INTEGER) is
  require
    t >= 0
  local
    node : KNOT
  do
    create node.make(s, num, t, node.tentative, pr)
    if s = Void then
      node.set_status(node.permanent)
    end
    stop_list.add_last(node)
  end
stop_list : LINKED_LIST[KNOT]
desire_time : INTEGER
is_bus(num: INTEGER) : BOOLEAN is
  do
    Result := num >= 100 and num <= 999
  end
find_neighbor(dbase: DATABASE; sn: STRING; pr: INTEGER) is
  require
    sn /= Void
  local
    cost : INTEGER
    index : INTEGER
    switch : INTEGER
    p_node : KNOT
    node : KNOT
    route : ROUTE
    station : STATION
    last : STATION
    name : STRING
    break : BOOLEAN
  do
    from
      switch := 0
    until switch > 1
    loop
      if switch = 0 then
        cost := bus_time
      else
        cost := train_time
      end
      if pr >= stop_list.lower and pr <= stop_list.upper then
        p_node := stop_list.item(pr)
      end
      if p_node /= Void then
        if p_node.get_station /= Void then
          if switch = 0 then
            if is_train(p_node.get_number) then
              cost := change_time + cost
            end
            route := dbase.get_first_bus_route
          else
            if is_bus(p_node.get_number) then
              cost := change_time + cost
            end
```

```
                    route := dbase.get_first_train_route
                end
                from
                until route = Void
                loop
                    station := route.get_first_station
                    last := station
                    from
                    until station = Void or break
                    loop
                        name := station.get_name.twin
                        if name /= Void and name.is_equal(sn) then
                            if not last.get_name.is_equal(name) then
                                index := get_index(last, route.get_number)
                                if index >= 0 then
                                    node := stop_list.item(index)
                                    if node.get_station /= Void then
                                        if is_train(node.get_number) then
                                            if node.get_time > p_node.get_time + cost then
                                                node.set_pred(pr)
                                                node.set_time(p_node.get_time + cost)
                                                node.set_number(route.get_number)
                                            end
                                        end
                                    end
                                else
                                    add_node(pr, last, p_node.get_time+cost, route.get_number)
                                end
                            end
                            last := route.get_next_station(station)
                            if last /= Void then
                                index := get_index(last, route.get_number)
                                if index >= 0 then
                                    node := stop_list.item(index)
                                    if node.get_station /= Void then
                                        if is_train(node.get_number) then
                                            if node.get_time > p_node.get_time + cost then
                                                node.set_pred(pr)
                                                node.set_time(p_node.get_time + cost)
                                                node.set_number(route.get_number)
                                            end
                                        end
                                    end
                                else
                                    add_node(pr, last, p_node.get_time+cost, route.get_number)
                                end
                            end
                            break := True
                        else
                            last := station
                            station := route.get_next_station(station)
                        end
                    end
                    break := False
                    if switch = 0 then
                        route := dbase.get_next_bus_route(route)
                    else
                        route := dbase.get_next_train_route(route)
                    end
                end
            end
        end
        switch := switch + 1
    end
  end
end

class STATION
```

```
creation
    make
feature {ROUTE, CONNECTION_FINDER}
    get_open : STRING is
        do
            Result := open.twin
        end
    get_acc : STRING is
        do
            Result := accessibility.twin
        end
    get_name : STRING is
        do
            Result := name.twin
        end
feature {NONE}
    accessibility : STRING
    open : STRING
    name : STRING
feature {ROUTE}
    do_save(file: TEXT_FILE_WRITE) is
        require
            file.is_connected
        do
            file.put_string(name + "_" + accessibility + "_" + open)
        end
    make(new_name, new_open, new_acc: STRING) is
        require
            new_name /= Void
            new_open /= Void
            new_acc /= Void
        do
            name := new_name.twin
            open := new_open.twin
            accessibility := new_acc.twin
        end
    show is
        do
            io.put_string(name)
        end
    match(targetname: STRING) : BOOLEAN is
        require
            targetname /= Void
        do
            Result := name.same_as(targetname)
        end
end

class FILE_DATABASE
inherit
    DATABASE
creation
    make
feature {HPTA_TRANSIT_INFO}
    do_save is
        local
            file_2_write : TEXT_FILE_WRITE
            i : INTEGER
        do
            create file_2_write.connect_to(file_name)
            if file_2_write.is_connected then
                from
                    i := employees.lower
                until i > employees.upper
                loop
                    employees.item(i).do_save(file_2_write)
                    i := i + 1
                end
```

```
            from
               i := bus_routes.lower
            until i > bus_routes.upper
            loop
               bus_routes.item(i).do_save(file_2_write)
               i := i + 1
            end
            from
               i := train_routes.lower
            until i > train_routes.upper
            loop
               train_routes.item(i).do_save(file_2_write)
               i := i + 1
            end
            file_2_write.disconnect
         else
            io.put_string("Update_database_failed!%N")
         end
      end
   add_route(new_route: ROUTE) is
      do
         if is_bus(new_route.get_number) then
            bus_routes.add_last(new_route)
         elseif is_train(new_route.get_number) then
            train_routes.add_last(new_route)
         end
      end
   make is
      do
         create employees.make
         create bus_routes.make
         create train_routes.make
         load
      ensure
         employees /= Void
         bus_routes /= Void
         train_routes /= Void
      end
   delete_staff(staff: STAFF) is
      do
         employees.remove(employees.index_of(staff) )
      end
   delete_route(route: ROUTE) is
      do
         if is_bus(route.get_number) then
            bus_routes.remove(bus_routes.index_of(route) )
         elseif is_train(route.get_number) then
            train_routes.remove(train_routes.index_of(route) )
         end
      end
   add_staff(new_staff: STAFF) is
      do
         employees.add_last(new_staff)
      end
   is_locked : BOOLEAN is
      do
         Result := not employees.is_empty
      end
feature {NONE}
   file_name : STRING is "sys_info.txt"
   is_bus(num: INTEGER) : BOOLEAN is
      do
         Result := num >= min_bus_route_num and num <= max_bus_route_num
      end
   employees : LINKED_LIST[STAFF]
   is_train(num: INTEGER) : BOOLEAN is
      do
         Result := num >= min_train_route_num and num <= max_train_route_num
```

```
      end
   bus_routes : LINKED_LIST [ROUTE]
   load is
      local
         input_string : STRING
         text_file_read : TEXT_FILE_READ
         text_file_write : TEXT_FILE_WRITE
         split : ARRAY[STRING]
         new_staff : STAFF
         route : ROUTE
      do
         create text_file_read.connect_to(file_name)
         if text_file_read.is_connected then
            from
               text_file_read.read_line
            until text_file_read.end_of_input
            loop
               if text_file_read.last_string.upper = 1 then
                  inspect
                     text_file_read.last_string.first.to_upper
                  when 'S' then
                     text_file_read.read_line
                     input_string := text_file_read.last_string.twin
                     split := input_string.split
                     create new_staff.make(split.first.to_integer , split.last)
                     employees.add_last(new_staff)
                  when 'B', 'T' then
                     text_file_read.read_line
                     input_string := text_file_read.last_string.twin
                     split := input_string.split
                     route := find_route(split.item(4).to_integer)
                     if route = Void then
                        create route.make(split.item(4) .to_integer)
                        route.add_station(split.first , split.item(2) , split.item(3) , split.last)
                        if split.item(4).to_integer > max_train_route_num then
                           bus_routes.add_last(route)
                        else
                           train_routes.add_last(route)
                        end
                     else
                        route.add_station(split.first , split.item(2) , split.item(3) , split.last)
                     end
                  else
                  end
               end
               text_file_read.read_line
            end
            text_file_read.disconnect
         else
            create text_file_write.connect_to(file_name)
            if text_file_write.is_connected then
               text_file_write.disconnect
            end
         end
      end
   train_routes : LINKED_LIST [ROUTE]
feature {HPTA_TRANSIT_INFO, CONNECTION_FINDER}
   find_route(num: INTEGER) : ROUTE is
      local
         i : INTEGER
         route : ROUTE
      do
         if num > max_train_route_num then
            from
               i := bus_routes.lower
            until i > bus_routes.upper or else  bus_routes.item(i).match(num)
            loop
               i := i + 1
```

```
            end
            if i <= bus_routes.upper then
                route := bus_routes.item(i)
            end
        else
            from
                i := train_routes.lower
            until i > train_routes.upper or else  train_routes.item(i).match(num)
            loop
                i := i + 1
            end
            if i <= train_routes.upper then
                route := train_routes.item(i)
            end
        end
        Result := route
    end
get_next_bus_route(route: ROUTE) : ROUTE is
    require
        bus_routes.index_of(route) > 0
    local
        next_route : ROUTE
    do
        if bus_routes.index_of(route) < bus_routes.upper then
            next_route := bus_routes.item(bus_routes.index_of(route) +1)
        end
        Result := next_route
    end
get_first_bus_route : ROUTE is
    local
        route : ROUTE
    do
        if not bus_routes.is_empty then
            route := bus_routes.first
        end
        Result := route
    end
find_staff(num: INTEGER) : STAFF is
    local
        i : INTEGER
        staff : STAFF
    do
        from
            i := employees.lower
        until i > employees.upper or else  employees.item(i).match(num)
        loop
            i := i + 1
        end
        if i <= employees.upper then
            staff := employees.item(i)
        end
        Result := staff
    end
get_next_train_route(route: ROUTE) : ROUTE is
    require
        train_routes.index_of(route) > 0
    local
        next_route : ROUTE
    do
        if train_routes.index_of(route) < train_routes.upper then
            next_route := train_routes.item(train_routes.index_of(route) +1)
        end
        Result := next_route
    end
get_first_train_route : ROUTE is
    local
        route : ROUTE
    do
```

```
            if not train_routes.is_empty then
                route := train_routes.first
            end
            Result := route
        end
end

class ROUTE
creation
    make
feature {HPTA_TRANSIT_INFO}
    remove_station(name: STRING) is
        local
            i : INTEGER
        do
            from
                i := stops.lower
            until i > stops.upper or else  stops.item(i).match(name)
            loop
                i := i + 1
            end
            if i <= stops.upper then
                stops.remove(i)
            end
        end
    show is
        local
            i : INTEGER
        do
            if number > 99 then
                io.put_string ("%NBus_route_No.")
            else
                io.put_string ("%NTrain_route_No.")
            end
            io.put_integer(number)
            io.put_string (":_")
            from
                i := stops.lower
            until i > stops.upper
            loop
                stops.item(i).show
                if i < stops.upper then
                    io.put_string ("->")
                end
                i := i + 1
            end
            io.put_new_line
        end
    make(num: INTEGER) is
        require
            num > 9
            num < 1000
        do
            number := num
            create stops.make
        ensure
            number = num
            stops /= Void
        end
feature {DATABASE}
    do_save(file: TEXT_FILE_WRITE) is
        require
            file.is_connected
        local
            i : INTEGER
            tag : STRING
            last : STRING
        do
```

```
            if number > 99 then
               tag := "_b"
            else
               tag := "_t"
            end
            last := "_None"
            from
               i := stops.lower
            until i > stops.upper
            loop
               file.put_string(tag+"%N")
               stops.item(i).do_save(file)
               file.put_string("_" + number.to_string +"_" + last +"%N")
               last := stops.item(i).get_name.twin
               i := i + 1
            end
         end
feature {NONE}
      number : INTEGER
      stops : LINKED_LIST[STATION]
feature {CONNECTION_FINDER}
      get_first_station : STATION is
         local
            station : STATION
         do
            if stops.upper > 0 then
               station := stops.first
            end
            Result := station
         end
      get_next_station(station1: STATION) : STATION is
         require
            station1 /= Void
         local
            station : STATION
         do
            if stops.index_of(station1) < stops.upper then
               station := stops.item(stops.index_of(station1) +1)
            end
            Result := station
         end
feature {HPTA_TRANSIT_INFO, DATABASE}
      add_station(new_name, access, open_hour, last_stop: STRING) is
         local
            new_station : STATION
            i : INTEGER
            last : STRING
         do
            from
               i := stops.lower
            until i > stops.upper or else  stops.item(i).match(new_name)
            loop
               i := i + 1
            end
            if i > stops.upper then
               create new_station.make(new_name, access, open_hour)
               last := last_stop.twin
               last.to_upper
               if last.same_as("NONE") then
                  stops.add_first(new_station)
               else
                  from
                     i := stops.lower
                  until i > stops.upper or else  stops.item(i).match(last_stop)
                  loop
                     i := i + 1
                  end
                  if i <= stops.upper then
```

```
                    stops.add(new_station , i+1)
                else
                    create new_station.make(last_stop , access , open_hour)
                    stops.add_last(new_station)
                    stops.add_last(new_station)
                end
            end
        end
    end
feature {ANY}
    get_number : INTEGER is
        do
            Result := number
        end
    match(num: INTEGER) : BOOLEAN is
        do
            Result := num = number
        end
invariant
    number > 9
    number < 1000
end

class HPTA_TRANSIT_INFO
inherit
    SYSTEM
creation
    make
feature {NONE}
    delete_menu is
        do
            execute_command_line("cls")
            io.put_string("[
______________________________***************************************
________________________________________Welcome_to_HPTA
______________________________***************************************
_________________________________S_Delete_a_station
_________________________________E_Delete_a_staff
_________________________________R_Delete_a_route
_________________________________G_Go_back
______________________________Enter_menu_choice:
______________________________]")
        end
    db : DATABASE
    update_menu is
        do
            execute_command_line("cls")
            io.put_string("[
______________________________***************************************
________________________________________Welcome_to_HPTA
______________________________***************************************
_________________________________A_Add
_________________________________D_Delete
_________________________________G_Go_back
______________________________Enter_menu_choice:
______________________________]")
        end
    menu is
        do
            execute_command_line("cls")
            io.put_string("[
______________________________***************************************
________________________________________Welcome_to_HPTA
______________________________***************************************
_________________________________U_Update_System_Information
_________________________________I_Inquire_about_Transit_Information
_________________________________Q_Quit
______________________________Enter_menu_choice:
```

```
______________________________]")
        end
    set_finder(new_finder: CONNECTION_FINDER) is
        require
            new_finder /= Void
        do
            finder := new_finder
        ensure
            finder = new_finder
        end
    inquire_menu is
        do
            execute_command_line("cls")
            io.put_string("[
______________________________*************************************
_________________________________________Welcome_to_HPTA
______________________________*************************************
___________________________________F_Find_a_possible_connection
___________________________________S_Show_a_route
___________________________________B_Browse_all_routes
___________________________________G_Go_back
______________________________Enter_menu_choice:
______________________________]")
        end
    do_delete is
        local
            is_end : BOOLEAN
            num : INTEGER
            staff : STAFF
            route : ROUTE
            input : STRING
            name : STRING
        do
            from
            until is_end
            loop
                delete_menu
                io.read_line
                input := io.last_string.twin
                io.put_new_line
                if not input.is_empty then
                    inspect
                        input.first.to_upper
                    when 'G' then
                        is_end := True
                    when 'S' then
                        io.put_string("%NEnter_route_number:_")
                        io.read_line
                        num := io.last_string.to_integer
                        io.put_string("%NEnter_station_name:_")
                        io.read_line
                        name := io.last_string.twin
                        route := db.find_route(num)
                        if route /= Void then
                            route.remove_station(name)
                        else
                            io.put_string ("%NNo_such_a_station%N")
                            io.read_line
                        end
                    when 'R' then
                        io.put_string("%NEnter_route_number:_")
                        io.read_line
                        num := io.last_string.to_integer
                        route := db.find_route(num)
                        if route /= Void then
                            db.delete_route(route)
                        else
                            io.put_string ("%NNo_such_a_station%N")
```

```
                    io.read_line
                end
            when 'E' then
                io.put_string("%NEnter_ID: ")
                io.read_line
                num := io.last_string.to_integer
                staff := db.find_staff(num)
                if staff /= Void then
                    db.delete_staff(staff)
                else
                    io.put_string ("%NNo_such_a_staff%N")
                    io.read_line
                end
            else
            end
        end
    end
end
finder : CONNECTION_FINDER
do_update is
    local
        id : INTEGER
        passed : BOOLEAN
        is_end : BOOLEAN
        password : STRING
        input : STRING
        staff : STAFF
    do
        io.read_line
        if db.is_locked then
            io.put_string ("%NEnter_employee_ID: ")
            io.read_line
            id := io.last_string.to_integer
            staff := db.find_staff(id)
            if staff /= Void then
                io.put_string ("%NEnter_password: ")
                io.read_line
                password := io.last_string.twin
                passed := staff.login(password)
            end
        else
            io.put_string("[
__________________________The_list_of_authorized_staff_is_not_empty,
__________________________so_please_set_authorization_as_soon_as_possible...
__________________________]")
            passed := True
            io.read_line
        end
        if passed then
            from
            until is_end
            loop
                update_menu
                io.read_line
                input := io.last_string.twin
                io.put_new_line
                if not input.is_empty then
                    inspect
                        input.first.to_upper
                    when 'A' then
                        do_add
                    when 'D' then
                        do_delete
                    when 'G' then
                        is_end := True
                    else
                    end
                end
```

```
            end
            db.do_save
         else
            io.put_string ("%NLogin_failed!%N")
            io.read_line
         end
      end
   run is
      do
         from
         until io.last_character.to_upper = 'Q'
         loop
            menu
            io.read_character
            io.put_new_line
            inspect
               io.last_character.to_upper
            when 'U' then
               do_update
            when 'I' then
               do_inquire
            else
            end
         end
      end
   add_menu is
      do
         execute_command_line("cls")
         io.put_string("[
______________________________****************************************
________________________________________Welcome_to_HPTA
______________________________****************************************
_________________________________S_Add_a_station
_________________________________E_Add_a_staff
_________________________________G_Go_back
______________________________Enter_menu_choice:
______________________________]")
      end
   do_add is
      local
         employee : STAFF
         id : INTEGER
         input : STRING
         name : STRING
         password : STRING
         open : STRING
         access : STRING
         last : STRING
         is_end : BOOLEAN
         route : ROUTE
      do
         from
         until is_end
         loop
            add_menu
            io.read_line
            input := io.last_string.twin
            io.put_new_line
            if not input.is_empty then
               inspect
                  input.first.to_upper
               when 'G' then
                  is_end := True
               when 'S' then
                  io.put_string("%NEnter_station_name:_")
                  io.read_line
                  name := io.last_string.twin
                  io.put_string("%NEnter_open_hour:_")
```

```
                        io.read_line
                        open := io.last_string.twin
                        io.put_string("%NEnter_its_accessibility:_")
                        io.read_line
                        access := io.last_string.twin
                        io.put_string("%NEnter_route_number:_")
                        io.read_line
                        id := io.last_string.to_integer
                        io.put_string("%NEnter_the_name_of_its_last_station:_")
                        io.read_line
                        last := io.last_string.twin
                        route := db.find_route(id)
                        if route = Void then
                            create route.make(id)
                            route.add_station(name, access, open, last)
                            db.add_route(route)
                        else
                            route.add_station(name, access, open, last)
                        end
                    when 'E' then
                        io.put_string("%NEnter_your_ID:_")
                        io.read_line
                        id := io.last_string.to_integer
                        io.put_string("%NEnter_your_password:_")
                        io.read_line
                        password := io.last_string.twin
                        create employee.make(id, password)
                        db.add_staff(employee)
                    else
                    end
                end
            end
        end
    set_database(new_database: DATABASE) is
        require
            new_database /= Void
        do
            db := new_database
        ensure
            db = new_database
        end
    do_inquire is
        local
            input : STRING
            start : STRING
            dest : STRING
            is_end : BOOLEAN
            num : INTEGER
            time : INTEGER
            route : ROUTE
        do
            from
            until is_end
            loop
                inquire_menu
                io.read_line
                input := io.last_string.twin
                io.put_new_line
                if not input.is_empty then
                    inspect
                        input.first.to_upper
                    when 'B' then
                        from
                            route := db.get_first_bus_route
                        until route = Void
                        loop
                            route.show
                            route := db.get_next_bus_route(route)
```

```
                    end
                    from
                        route := db.get_first_train_route
                    until route = Void
                    loop
                        route.show
                        route := db.get_next_train_route(route)
                    end
                    io.put_string ("%N%NStrike_any_key_to_continue...")
                    io.read_line
                when 'F' then
                    io.put_string("%NEnter_the_station_name_of_your_start:_")
                    io.read_line
                    start := io.last_string.twin
                    io.put_string("%NEnter_the_station_name_of_your_destination:_")
                    io.read_line
                    dest := io.last_string.twin
                    io.put_string("%NEnter_your_desire_time(in_minutes):_")
                    io.read_line
                    time := io.last_string.to_integer
                    io.put_string(finder.get_connection(db, start, dest, time) )
                    io.put_string ("%N%NStrike_any_key_to_continue...")
                    io.read_line
                when 'S' then
                    io.put_string ("Input_the_route_number_(10_-_999):_")
                    io.read_line
                    num := io.last_string.to_integer
                    route := db.find_route(num)
                    if route /= Void then
                        route.show
                    else
                        io.put_string ("Sorry_there_is_no_such_a_route")
                    end
                    io.put_string ("%N%NStrike_any_key_to_continue...")
                    io.read_line
                when 'G' then
                    is_end := True
                else
                end
            end
        end
    end
feature {ANY}
    make is
        local
            prime_finder : PRIME_FINDER
            file_database : FILE_DATABASE
        do
            create file_database.make
            set_database(file_database)
            create prime_finder.make
            set_finder(prime_finder)
            run
        end
end
```

Appendix D

Reference Manual of Spark

D.1 Code Block Tag

Code Block Tag is used to identify the class, to which this block belongs. So there is nothing need to do for the class block, but for class member block, including invariant block, ones must put code block tag, class name followed by three dots, at the beginning.

D.2 Graphic Notation Setting

All the tags listed as following should be put in setting line, which is right behind graphic including command.

- @VERTICAL: if set, the diagram will be drawn vertically, otherwise horizontally.
- @HEAD: if set, the class diagram will be shown with class name nodes only.
- @BRIEF: if set, the class diagram will be shown without parameters and types.
- @CONCISE: if set, the class diagram will hide all the information about the method's parameters of the involved class.
- @METHOD: if set, all class methods only will be shown in this diagram.

- @ATTRIBUTE: if set, only attributes of class can be saw in the diagram.
- @ACTION" if set, only actions of class can be saw in the diagram.

D.3 Program Code Quotation

"CODE LIST BEGIN" and "CODE LIST END" are the specific tags used to include continuous program code into the source file. This tags can be put anywhere in the source file as comments. Front ends will insert the parsed code between them, if they find them.

Appendix E

Document Structure of AsciiDoc

An AsciiDoc document consists of a series of block elements. Almost any combination of zero or more elements constitutes a valid AsciiDoc document: documents can range from a single sentence to a multi-part book. In the following table of AsciiDoc document structure, parentheses '(' and ')' indicate grouping when needed, square brackets '[' and ']' enclose optional items, curly parentheses '{' and '}' show the (zero or more) repeatable items, and vertical bars '|' separate alternatives.

Document ::= [Header] [Preamble] { Section }

Header ::= Title [AuthorLine [RevisionLine]]

AuthorLine ::= FirstName [[MiddleName] LastName] [EmailAddress]

RevisionLine ::= [Revision] Date

Preamble ::= SectionBody

Section ::= Title [SectionBody] { Section }

SectionBody ::= (([BlockTitle] Block) | BlockMacro) { ([BlockTitle] Block) | BlockMacro }

Block ::= Paragraph | DelimitedBlock | List | Table

List ::= BulletedList | NumberedList | LabeledList | CalloutList

BulletedList ::= ListItem { ListItem }

NumberedList ::= ListItem { ListItem }

CalloutList ::= ListItem { ListItem }

LabeledList ::= ItemLabel { ItemLabel } ListItem { ItemLabel { ItemLabel } ListItem }

ListItem ::= ItemText { List | ListParagraph | ListContinuation }

Table ::= Ruler [TableHeader] TableBody [TableFooter]

TableHeader ::= TableRow { TableRow } TableUnderline

TableFooter ::= TableRow { TableRow } TableUnderline

TableBody ::= TableRow { TableRow } TableUnderline

TableRow ::= TableData { TableData }

Table E.1: The block structure of AsciiDoc.

Appendix F

Syntax of Dot

The following is an abstract grammar for the **dot** language. Terminals are shown in bold font and nonterminals in italics. Literal characters are given in single quotes. Parentheses '(' and ')' indicate grouping when needed. Square brackets '[' and ']' enclose optional items. Curly parentheses '{' and '}' show the (zero or more) repeatable items. Vertical bars '|' separate alternatives.

---

graph ::= [**strict**] (**digraph** | **graph**) *id* '{' *stmt-list* '}'

id ::= *letter* { *letter* | *digital* | _ }

letter ::= **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z** | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **D** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z**

digital ::= **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **0**

stmt-list ::= [*stmt* [';'] [*stmt-list*]

stmt ::= *attr-stmt* | *node-stmt* | *edge-stmt* | *subgraph* | *id* '=' *id*

attr-stmt ::= (**graph** | **node** | **edge**) *attr-list*

attr-list ::= '[' [*a-list*] ']' [*attr-list*]

a-list ::= *id* '=' *id* [','] [*a-list*]

node-stmt ::= *node-id* [*attr-list*]

node-id ::= *id* [*port*]

port ::= *port-location* [*port-angle*] | *port-angle* [*port-location*]

port-location ::= ':' *id* | ':' '(' *id* ',' *id* ')'

port-angle ::= '@' *id*

edge-stmt	::=	(*node-id* | *subgraph*) *edgeRHS* [*attr-list*]
edgeRHS	::=	*edgeop* (*node-id* | *subgraph*) [*edgeRHS*]
subgraph	::=	[**subgraph** *id*] '{' *stmt-list* '}' | **subgraph** *id*
edgeop	::=	-> | --

Table F.1: Abstract grammar for the dot language

The language supports C++ style comments: /* */ and //.

Semicolons aid readability but are not required except in the rare case that a named subgraph with no body immediate precedes an anonymous subgraph, because under precedence rules this sequence is parsed as a subgraph with a heading and a body.

Complex attribute values may contain characters, such as commas and white space, which are used in parsing the dot language. To avoid getting a parsing error, such values need to be enclosed in double quotes.

Bibliography

[1] *FunnelWeb Developer Manual*, 2000. Version 3.2d for FunnelWeb V3.2.

[2] "Asciidoc." Stuart Rackham, 2007.

[3] S. W. Ambler, *Process Patterns: Building Large-Scale Systems Using Object Technology.* New York: Cambridge University Press, first ed., 1998.

[4] W. J. Brown, H. W. McCormick, and S. W. Thomas, *Anti-Patterns and Patterns in Software Configuration Management.* New York: Wiley, first ed., 1999.

[5] T. Budd, *An Introduction to Object-Oriented Programming.* Oregon State University: Pearson Education, second ed., 1996.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns.* Chichester, U.K.: Wiley, first ed., 1996.

[7] D. Cameron and B. Rosenblatt, *Learning GNU Emacs.* Sebastopol, CA: O'Reilly and Associates, first ed., 1991.

[8] S. J. Chapman, *MATLAB Programming for Engineers.* Toronto, Ontario: Thomson, third ed., 2004.

[9] B. Childs, "Literate Programming, A Practioner's View," *TUGboat Journal*, vol. 13, no. 3, pp. 261–268, 1992.

[10] B. J. Cox and A. J. Novobilski, *Object-Oriented Programming: An Evolutionary Approach.* The Stepstone Corporation: Addison-Wesley Publishing Company, second ed., 1991.

[11] S. Cozens, *Advanced Perl Programming.* Sebastopol, CA: O'Reilly, second ed., 2005.

[12] D. de Champeaux, D. Lea, and P. Faure, *Object-Oriented System Development.* Massachusetts: Addison-Wesley Publishing Company, first ed., 1993.

[13] D. F. D'Souza and A. C. Wills, *Objects, Components, and Frameworks with UML, The Catalysis Approach.* New Jersey: Addison Wesley Longman, Inc., first ed., 1999.

[14] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, "Graphviz and Dynagraph - Static and Dynamic Graph Drawing Tools," *Graph Drawing Software Journal*, pp. 127–148, January 2003.

[15] P. H. Feiler and W. F. Tichy, "Propagator: A Family of Patterns," *Proceedings of the Tools-23: Technology of Object-Oriented Languages and System*, p. 355, August 1997.

[16] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef, *Validated Designs for Object-oriented Systems.* Springer, first ed., 2004.

[17] M. Fowler, *Analysis Patterns: Reusable Object Models.* MA: Addison-Wesley, first ed., 1997.

[18] E. Freeman, E. Freeman, K. Sierra, and B. Bates, *Head First Design Patterns.* Cambridge, MA: O'Reilly Media, first ed., 2004.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Massachusetts: Addison-Wesley Publishing Company, first ed., 1995.

[20] E. R. Gansner and S. C. North, "An Open Graph Visualization System and Its Applications to Software Engineering," *Software-Practice and Experience Journal*, vol. 30, no. 11, pp. 1203–1233, 1999.

[21] D. V. Heesch, *Doxygen.* http://www.stack.nl/ dimitri/doxygen/, 2007.

[22] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–583, October 1969.

[23] D. E. Knuth, *The WEB System of Structured Documentation.* Stanford University, 1983. WEB user manual, version 2.5.

[24] D. E. Knuth, "Literate Programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, May 1984.

[25] D. E. Knuth and S. Levy, *The CWEB System of Structured Documentation.* American Methematical Society, 1994. CWEB user manual, version 3.0.

[26] D. E. Knuth, *The TEXbook.* Stanford University: Addison-Wesley Professional, first ed., 1984.

[27] J. A. Krommes, *FWEB.* http://w3.pppl.gov/ krommes/fweb_toc.html, 1998. A WEB System of Structured Documentation for multiple languages.

[28] C. Larman, *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design.* Upper Saddle River, NJ: Prentice Hall, first ed., 2001.

[29] C.-A. Lehalle, *Documentation for ocamaweb.ml.* ocamaweb.sourceforge.net, 2002.

[30] J. L. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, April 1960.

[31] B. Meyer, *Object-Oriented Software Construction.* Santa Barbara, California: Prentice Hall PTR, second ed., 1997.

[32] B. Meyer, "An Eiffel Tutorial," *ISE Technical Report, Interactive Software Engineering Inc.(ISE)*, July 2001.

[33] F. Mittelbach, M. Goossens, J. Braams, D. Carlisle, and C. Rowley, *The LATEXCompanion.* Addison-Wesley Professional, second ed., 2004.

[34] T. J. Mowbray and R. C. Malveau, *CORBA Design Patterns.* New York: Wiley, first ed., 1997.

[35] S. Oualline, *Vi IMproved — Vim.* Indianapolis, Indiana: Sams, first ed., 2001.

[36] T. W. Pratt and M. V. Zelkowitz, *Programming Languages: Design and Implementation.* Maryland: Prentice Hall, fourth ed., 2001.

[37] PTLogica, *Source Code Documentation as a Live User Manual.*

[38] N. Ramsey, "Literate programming simplified," *IEEE Software*, vol. 11, no. 5, pp. 97–105, September 1994.

[39] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design.* New Jersey: Prentice-Hall, Inc., first ed., 1991.

[40] R. W. Sebesta, *Concepts of Programming Languages.* University of Colorado: Addison Wesley Longman, Inc., first ed., 1999.

[41] E. Sekerinski, "Concurrent Object-Oriented Programs: From Specification to Code," in *First International Symposium on Formal Methods for Components and Objects*, (Leiden, Netherlands), pp. 403–423, Springer-Verlag, 2003.

[42] H. V. Vliet, *Software Engineering: Principles and Practice.* New York, NY 10158-0012, USA: Wiley, second ed., 2000.

[43] K. Walden and J.-M. Nerson, *Seamless Object-Oriented Software Architecture-Analysis and Design of Reliable Systems.* New Jersey: Addison Wesley Longman, Inc., first ed., 1994.

[44] L. Wall, T. Christiansen, and R. Schwartz, *Programming Perl.* Sebastopol, CA: O'Reilly & Associates, second ed., 1996.

[45] N. Walsh and L. Muellner, *DocBook: The Definitive Guide.* O'Reilly & Associates, first ed., 1999.

[46] R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software.* New Jersey: Prentice-Hall, Inc., first ed., 1990.