DESIGN PATTERNS IN ERASMUS

Rana M. D. Issa

A thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

December 2009

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By:	Rana M. D. Issa
Entitled:	Design Patterns in Erasmus

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

______________________________ Chair
Dr. Nematollaah Shiri-Varnaamkhaasti

______________________________ Examiner
Dr. Constantinos Constantinides

______________________________ Examiner
Dr. Thiruvengadam Radhakrishnan

______________________________ Supervisor
Dr. Peter Grogono

Approved by ______________________________
Chair of Department or Graduate Program Director

____
Dr. Robin Drew, Dean
Faculty of Engineering and Computer Science

Date ______________________________

Abstract

Design Patterns in Erasmus

Rana M. D. Issa

This thesis presents design patterns for the emerging programming language Erasmus that is based on a process-oriented paradigm. We have developed these patterns using the current version of Erasmus, µE, and its corresponding compiler MEC. The design patterns are inspired on some of the patterns presented in the book [GHJV95]. Also, we have followed the elements of documentation templates that appear in that book. The patterns are presented visually; the figures show the high-level structure of the design patterns.

Since the tendency of software development is to shift to concurrent and distributed computation, new paradigms are emerging to accommodate this shift. Erasmus is one of these new developing paradigms. We try in this thesis, through developing design patterns for Erasmus, to demonstrate and prove that process-oriented programming is a viable and valid paradigm. We also compare process-oriented programming presented by Erasmus with object-oriented programming in the implementation of the design patterns.

We have chosen a set of design patterns that represent the different categories that appear in the book [GHJV95] and we tried to preserve the essential concepts of the original designs. However, the designs are not a direct translation of their counterparts in OOP. Rather, the designs make use of the abstraction of POP and find solutions from within the paradigm itself. That is why in some designs, for one pattern in OOP we have created two or three design patterns in µE.

Acknowledgments

I would like to thank a lot Dr. Grogono for taking the trouble of having to communicate with me through e-mails and for supporting me at times when I found it very difficult to continue. I really appreciate the fact that he never put pressure on me, and that he gave me freedom in researching but at the same time he was there whenever I needed his advice. I learned a lot from you Dr. Grogono... thanks a lot.

I would like to thank also my family Maiada, Laith, my father, and David for their continuous support and encouragement. Also, I would like to thank David for checking for me the document and helping me with Latex.

Contents

List of Figures

Chapter 1

Introduction

He who knows nothing, loves nothing. He who can do nothing understands nothing. He who understands nothing is worthless. But he who understands also loves, notices, sees... The more knowledge is inherent in a thing, the greater the love.... Anyone who imagines that all fruits ripen at the same time as the strawberries knows nothing about grapes – Paracelsus

After fifteen years from publishing the book *Design Patterns: Elements of Reusable Object-Oriented Software*, Richard Helm one of its authors still believes that software design is always hard [GHJO09]. It seems that the difficulty always exists in software development despite of the attempts to standardize it through creating architectural frameworks on the high level of the design to creating design patterns to solve recurring problems on the low level of software development. When requirements are growing in complexity, hardware development is taking a new path to meet the requirement of incremental speed of the CPU, and the development of the existing software paradigms are adding a new level of complexity and as a result more difficulties to software development; one has to stop and think of possibly other alternatives to overcome all these variants. In fact many software engineers and researchers started already exploring new paths to face the new challenges. Some believe in patching the current system by evolving the current paradigms or by adding new frameworks to handle the new requirements. Others believe that the process of patching the current system adds itself a new level of complexity in programming, they believe in taking a new approach but learning the lessons from the previous experience.

Following the latter approach, Peter Grogono and Brian Shearing are in the process of developing a new programming language named “Erasmus”. The programming language is still evolving having the current version named as “microErasmus” (written as µE). This language has a prototype compiler named “MEC”. Erasmus and its current version µE are based on the process-oriented paradigm with a new interpretation of the paradigm and many lessons from the limitation of object-oriented programming (OOP) paradigm, the most used and the most known paradigm nowadays.

In this thesis we present design patterns based on the design patterns that appear in the book [GHJV95] to show that problems that arise in conventional programming can be expressed in a process-oriented paradigm, precisely Erasmus language. We tried to preserve the "essential concept" of the patterns that were written originally for OOP and we have adapted them to work on µE. We do not discuss in this thesis design patterns for concurrent or distributed programming. The concurrent and distributed design patterns are specialized patterns. They intend to provide methods for maximizing concurrency in numerical methods for high performance computation ([San97], [Lea99], and [MSM04]).

1.1 Structure of the Thesis

Chapter 2: the thesis starts by introducing the origins of *Design Patterns*, the first appearances of the term, and its development. It then gives a short history and how the authors of the book [GHJV95] structured their book and their designs. Later on it introduces the main aspects of documenting a design, and finally the impacts of Design Patterns.

Chapter 3: this chapter discusses mainly Erasmus and its current prototype version µE. It starts by introducing the new approach of building processors, multicore, and its impact on software development. To take advantage of the new hardware approach that adds a new level of difficulty to the current available paradigms; solutions were explored including the creation of new paradigms. This led to the creation of Erasmus, a programming language that is based on a process-oriented paradigm. The discussion starts by introducing the definition of process-oriented programming (POP), the problems that POP encountered in the past, the new changes on hardware that led to solving some of the problems with POP, and finally the definition of Erasmus and µE. Since OOP is the most known and used paradigm, a comparison between POP and OOP is also discussed. The discussion continues to relate Erasmus to the topic of this thesis that is creating design patterns for µE. This section introduces the categories of the design patterns that appeared in the book [GHJV95], and their relation with the implemented design patterns of this thesis. Finally, the last section of the chapter discusses some features and annotations that are related to µE. These features and annotations are referred to in the following chapters.

Chapter 4: the beginning of the work of this thesis starts from this chapter. It discusses the *Factory Design Pattern*. An introduction to the pattern in OOP is discussed. A description of the implementation of the pattern in Java. Finally, a description of the Factory Pattern in Erasmus is discussed. The full code of the design pattern in µE is listed in Appendix A.

Chapter 5: discusses the *Adapter Design Pattern.* An introduction to the pattern in OOP is discussed. This is followed by a description of the implementation of the pattern in Java. Finally, a description of the Adapter Pattern in Erasmus is discussed. The Adaptation could be in two levels: Protocol and Process. The full code of the design pattern in µE is listed in Appendix B.

Chapter 6: this chapter investigates the *Composite Design Pattern.* We start the discussion by introducing the pattern in OOP and its implementation in Java. We then describe our contribution of designing the Composite Pattern in Erasmus. The Composite Design comes in three levels in µE: cell, protocol, and process. The process composition section is explained incrementally. The full code of the design pattern in µE is listed in Appendix C.

Chapter 7: the *Chain of Responsibility Design Pattern* is discussed in this chapter. An introduction to the pattern in OOP is first discussed. This is followed by a description of its implementation in Java. This leads us to discuss the Chain of Responsibility Pattern in Erasmus. The full code of the design pattern in µE is listed in Appendix D.

Chapter 8: the *Command Design Pattern* is explored in this chapter. First we introduce the pattern in OOP and its implementation in Java. Then we get to our work: the design of the Command Pattern in Erasmus. The full code of the design pattern in µE is listed in Appendix E.

Chapter 9: this chapter discusses the *Interpreter Design Pattern.* An introduction to the pattern in OOP is first discussed. A description of its implementation in Java follows. This leads us to demonstrate and discuss the implementation of the Interpreter Pattern in Erasmus. The full code of the design pattern in µE is listed in Appendix F.

Chapter 10: last but not least, this chapter demonstrates the *Iterator Design Pattern.* An introduction to the pattern in OOP is first discussed, followed by its implementation in Java. This leads us to the discussion of our contribution: the design and implementation of the Iterator Pattern in Erasmus. The discussion is incremental in this section. The design of the Iterator comes in two types: with and without sharing aggregates. Both types are explored and the full code of both designs in µE is listed in Appendix G.

Chapter 2

Design Patterns

An important lesson to learn is not to reinvent the wheel over and over again. This is applicable to software design as well as to many other aspects of life. That is why designers thought of documenting solutions to problems that will appear again and again. These documented solutions that serve for defined contexts in software development are called "Design Patterns".

2.1 Definition of design pattern

The first appearance of patterns in programming (precisely in object-oriented programming), which was originally named as "pattern languages for object-oriented programs", was proposed in a paper by Beck and Cunningham [BC87]. The authors defined pattern as: "A pattern language guides a designer by providing a workable solution to all of the problems known to arise in the course of design. It is a sequence of bits of knowledge written in a style and arranged in an order which leads a designer to ask (and answer) the right questions at the right time".

The book [GHJV95] defines design pattern in software engineering as a reusable solution for recurring problems. A design pattern is not a clearly defined program that solves problems, rather it is a documented template that describes the problem and its solution.

This definition of design pattern is based on the definition of patterns by Christopher Alexander (see Alexander definition in 2.2), adding to it the statement that in the authors' work, patterns are solutions within a context expressed in terms of objects and interfaces.

Several authors have emphasized on the principle of design [FFBS04, Hol06], which is to always design for a change. In analyzing the problem authors of both books urge software designers to analyze the aspects or parts that will change or evolve, and then separate them from the rest of the code to do minimal refactoring during the upgrade phase of the application and to make the code as robust as possible. Also, rather than redeveloping the same functionality, it is better to reuse the parts that are generic and that work well already, as much as possible.

Finally it is important to note here that the most important rules to follow in design patterns for object-oriented programming are [GHJV95]:

- "Program to an interface, not an implementation".
- "Favor object composition over class inheritance".

2.2 History of Design Patterns

The very original appearance of the term "pattern language" was meant for designing buildings and towns. The term was put forward by the architect Christopher Alexander, who defined it as: "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [AIS77]. Unlike a design or a specification, a pattern is meant to be more general, and less specific. This allows users of the pattern to judge for themselves how to modify and where to use it, without losing its essence.

The first appearance of *pattern* in programming, which was originally named as "pattern languages for object-oriented programs", was proposed in the report [BC87] by Kent Beck and Ward Cunningham, at OOPSLA (Object-Oriented Programming, Systems, Languages & Applications) conference in 1987. The idea of creating patterns for object-oriented programming was an adaptation of the *pattern language* book by Christopher Alexander. Alexander believed that the people who occupy a home or an office are the ones to decide best the architectural design of their homes or offices, given that they are provided with guidance through a "pattern language". Pattern language is a collection of patterns, each of which states a problem, specifies when to use the pattern, and gives a solution to the problem.

Beck and Cunningham demonstrated five patterns that were designed for Smalltalk windows, and they stated that they had built ten patterns and had been working on 100-150 new patterns [BC87].

In Ottawa 1990, Erich Gamma and Richard Helm (two future authors of [GHJV95]), met for the first time and they found that they shared ideas about how to write reusable object-oriented programs.

In 1994, the authors of [GHJV95] collected twenty three different patterns in a book that made design patterns in OOP an established field in software engineering. The design patterns in this book became very known and popular, although it took quite a while for it to be so. Design patterns in object-oriented programming became an important part of designing industrial OOP languages and applications; Java, for instance, followed some patterns such as Iterator, Factory, and others in building its own Class Libraries.

The authors of the book [GHJV95] grouped the design patterns according to three categories: *creational*, *structural*, and *behavioral* patterns. Each category has a common function among design patterns it contains.

- *Creational Patterns* are concerned with the patterns that create objects. In this group, patterns are designed to create objects as independent as possible from the application code. The creation could be in one of two types: *object creational patterns*, or *class creational patterns*. Object creational patterns defer creating an object to another object, whereas class creational patterns make use of inheritance to support creating more types of classes (subclasses).

- *Structural Patterns* group patterns that arrange classes or objects in a way to reduce decoupling to the least possible degree to form a larger structure.

- *Behavioral Patterns* are concerned with the functional part of objects and their interconnection. The designs suggest patterns that provide ideal solutions to how objects communicate among each other and to how to assign responsibilities to objects. In this group we find different ways of attacking the problem:

  - *behavioral class patterns* use inheritance to distribute among subclasses the behavior or the functionality.
  - *behavioral object patterns* use composition instead of inheritance to distribute behavior among composed classes. An important point to consider is how can peer objects know about each other. Another group of patterns are concerned with finding optimized solutions to encapsulate a functionality in an object and how to get the request through.

It is important to note here that the authors of [GHJV95] stated clearly that their design patterns do not deal with concurrency, distributed programming, or even real-time programming. However, since concurrent programming and parallelism became a necessity in today's software development, new design patterns that deal with concurrency and parallel development were created to tackle recurring problems that use concurrent and parallel programming (such as the patterns discussed in [Lea99] and [MSM04]).

In spite of finding from the early days that design patterns are very important as guidelines for better development, especially for large-scale programs, it took a while to establish the discipline and to spread it among software developers. The larger software programs are, the more crucial it is to follow design patterns and design architectures in designing and implementing softwares. Otherwise, upgrading and maintaining large-scale softwares become burdensome and continuous refactoring is needed to adapt new changes and to fix bugs.

The main two issues design patterns tackle and work on optimizing are *coupling* and *cohesion* between objects and modules. That is why a definition of both terms is necessary to understand the context in which design patterns work.

- *Coupling* is defined as "the degree to which software components depend on each other" [GHJV95]. Coupling can be loose or tight. **Tight coupling** happens when classes depend on each other. With tight coupling it is difficult to use classes independently from each other. **Loose coupling** on the other hand, makes it possible to use classes by themselves [SMC74].

- *Cohesion* is a term that is a measurement used in software engineering to evaluate the quality of software design. Cohesion measures how much components in a module are strongly related or how well connected they are.

2.3 Documentation of a design pattern

Since design patterns are more like templates rather than implementation oriented solutions, it is important to have design patterns documented in a clear and formal way. The authors of [GHJV95] came up with the following structure in documenting each design pattern:

- The **pattern name** is the name that will be associated to the pattern, that describes it, and with which the pattern will be referred to in future reuse.

- The **problem** describes the context of the problem that needs a solution, and how to identify the problem.

- **The solution** is the solution to the problem within a limited context. It describes the required components, their relationships, and their responsibilities.

- The **consequences** tackles the advantages and disadvantages of the pattern. It also suggests, if possible, alternative ways of achieving the aim of the pattern, or how to overcome the disadvantages.

There are other points to be documented such as *alternative names* (or what is called: *also known as*), *structure*, that is the class diagram of the pattern, *participants* which are the main modules building up or forming the pattern, *implementation* that gives an example in a special object-oriented programming language, and *sample code* in which main points are highlighted in a program or a module.

2.4 Impact

The concept of design patterns created, after its creation, opponents as well as supporters. The critics of those who are against are discussed in the following points as well as the answers to these critics that demonstrate that most of these critics are refutable:

- That design patterns cover the inability and incapability of some abstractions to solve or support some solutions. So patterns are some sort of workaround to solve failures in some abstractions [BLR96]. Even if we assume that this argument is true, a solution is still needed to overcome the failures in popularly used abstractions. Hence, design patterns are still important and necessary to design and implement robust programs.

- Novice programmers fall into what is called the *pattern trap*. The *pattern trap* happens when programmers try to enforce using different patterns inappropriately. This mainly happens when programmers do not understand well the problem they are trying to solve, do not understand well the pattern itself, and do not know exactly what kind of problem the pattern solves [Sin01].

- That design patterns should be used with caution, if not they may prevent the progress of the evolution of the software, and they may affect negatively maintainability instead of improving it [KG08].

Chapter 3

Erasmus

3.1 Introduction

Erasmus was originally designed to overcome the difficulty that professional programmers encountered in refactoring and upgrading large-scale programs. High coupling between classes in object-oriented programming makes it difficult to refactor, upgrade, and maintain those types of programs. As a result, the code becomes less flexible and less robust. Although OOP solves in principle the issue with decoupling between classes, it does not in practice, especially when it comes to large applications and in fact reducing coupling depends mostly on the amount of programmers' experience. Erasmus bases its solution on the claim that coupling is reduced between processes even if there is not actual concurrency. However, if concurrency is needed, Erasmus can still exploit it. The urging need for concurrent programming that was led by the new approach of increasing hardware speed is discussed in Section 3.2. The impact of hardware development on software development, the need for a shift of paradigm, and finally, Erasmus as a solution, are all discussed in the following sections. The discussion in this chapter also introduces some features of Erasmus that will be referred to in the discussion of the following Design Patterns chapters.

3.2 Multicore Hardware

Sutter explains the development of processors in terms of clock speed, execution optimization, and cache [Sut05]. He demonstrates the dead end in achieving higher processors' speed by increasing clock speed, as well as optimizing the execution of the instructions. Increasing the size of the cache memory, however, improves the speed but not to a big extent. This fact led hardware designers to look for another alternative in improving the speed. The approach that was taken was by building multicore processors that are composed of two or more independent cores or CPUs. To achieve higher speed in running programs in multicore hardware architecture, concurrent programming is required. Sequential programming does not take advantage of the multicore architecture. However, Sutter agrees that learning concurrent programming for programmers who have been working with sequential object-oriented programming is quite difficult. Add to that, the implementation of concurrency is not yet standardized, and difficult to visualize in large-scale applications such as industrial applications. The thing that makes learning it more difficult.

Learning or developing applications using concurrent programming is very difficult due to the divergence between the concurrency abstraction that is used in programming from the physical concurrency that happens in the real world [Lee06]. Lee argues that programs that are developed using threads are highly nondeterministic, and that thread programming relies on programming style to force nondeterminism to achieve deterministic goals. Also, he argues that most multithreaded programs have bugs and that these bugs did not become yet a real problem because the hardware architecture and operating systems deliver simple parallelism. These bugs, in his opinion, will appear when hardware architecture moves towards multicore.

Because concurrency or parallelism is becoming a must in todays programming, and because of the problems facing programmers ([Lee06]) in using threads that is suggested in object-oriented programming languages in order to achieve concurrency, a new paradigm was created [GS07] that is based on process-oriented programming. The language is named Erasmus. The following sections will explore this language and some of its main features.

3.3 Process-Oriented Programming (POP)

Process-oriented programming (POP) was described in many different works through the history of computing. Some parts of the paradigm were implemented in building few programming languages such as Ada and Erlang, but not as the main abstraction of the language.

Process-oriented programming appears in [Hoa78] as: “Input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method”.

Another occurrence of a definition of POP appears in [Dij65] and says: “We have stipulated that processes should be connected loosely; by this we mean that part from the (rare) moments of explicit intercommunication, the individual processes themselves are to be regarded as completely independent of each other”.

A third definition of POP is mentioned in [Mag89]; it explains how processes should be different from procedures: “process is different from a procedure activation (or a ”message”) in that it is executed in parallel with other processes”.

Finally, Erasmus is a new programming language based on the process-oriented paradigm [GS07]. It is introduced in Section 3.4. This is the programming language under study and for which an adaptation of well-known design patterns are created in this thesis.

3.3.1 Problems with Process-Oriented Programming

Although process-oriented programming is around for a while, however, it did not become as popular as object-oriented programming in mainstream industrial programming languages. The following points discuss the issues with POP:

- Context-switching between processes is a typical problem that is taught in operating system courses, because the action of switching from one process to another to run on a processor is slow. That is why invoking functions is preferred over working with processes. Also, using threads was another solution because threads consume less time in switching than processes.
- Passing data in POP from one process to another is done through copying the data and not by passing it by reference. This method is costly, but allowing passing data by reference, on the other hand, increases decoupling [Zen92].

3.3.2 Solution to Process-Oriented Programming problems

Since the time of creating the concept of process-oriented programming until now many things have changed. These changes helped in reconsidering building a new paradigm based on process-oriented programming. The following is a list of the main changes:

- As explained in Section 3.2, the tendency of hardware manufacturers was shifted from building processors with higher speeds to providing more processors in one package to increase the speed.
- To increase the speed of performance mentioned in the previous point, applications must run concurrently. The definition of POP states clearly that processes run concurrently, this means that POP programmers do not need to apply any extra mechanisms to achieve concurrency and that concurrency is a basic feature of the paradigm. This makes concurrency in POP natural.
- Since the speed of multicore processors is higher if its full capacity is used, then the context-switching between processes is not anymore a significant issue.

- Learning the lesson from the weaknesses and strengths in object-oriented paradigm gives us better perspective in creating a new paradigm by overcoming the weaknesses.

3.4 Erasmus and µE

Erasmus as described in [GS07] is a programming paradigm that is based on process-oriented Programming. The main modules that build up a program in this paradigm are: cells, processes, ports, protocols, and channels. A cell defines the structure of the program; a process defines the behavior; and a port allows processes and cells, processes, and cells to communicate with each other and to send messages one to another.

A Cell instantiates processes, variables, ports, channels, and other cells it contains. A cell runs in a single thread. This includes the processes and variables it contains. Cells and processes are mutually nested to any depth.

Cells and processes communicate by sending messages. These messages are exchanged between ports. These ports are connected to each other through channels. A channel connects explicitly two ports together; one behaves as a client (sender of the message) and the other behaves as a server (the receiver of the message).

The current implementation of the language is a prototype language that is a subset of Erasmus. This prototype is called “microErasmus” or µE as an abbreviation.

3.5 Process-Oriented Programming Versus Object-Oriented Programming

Ideally, we would compare the process-oriented paradigm and the object-oriented paradigm. However, this would be an enormous task, perhaps not even feasible. The comparison in this section is based on exemplars of the two paradigms: Erasmus for POP and Java for OOP.

- For instance processes and cells in Erasmus replace methods in Java.
- Parameter’s ports in Erasmus function the same way parameters of a method do in Java, but they support more powerful actions such as synchronization (using signals) and they support more complex protocols.
- Cells in Erasmus just like objects in Java are created Dynamically.
- Although in both languages, Java and Erasmus, objects and processes are allocated memory once constructed, they are different from each other, because in Java the

object is not active until one of its methods is invoked, while the process in Erasmus is active as soon as it is instantiated.

- Same as in Java, where objects are the unit of encapsulation, cells in Erasmus are the unit of encapsulation. They are also not bound to a certain size.

- The race condition that may occur between shared states in Java does not happen in Erasmus, although the value of shared variables may change after a process blocks to exchange messages that include these variables.

- Coding with Erasmus coming from an Java background does not need a lot of adjustment. In Erasmus cells replace classes and processes replace methods.

- µE compiler accepts two source files, the file that includes the source code and another file that maps processes to processors. This way the code could run as a distributed application.

- Processes in Erasmus exchange messages through ports that define their interfaces. This makes the coupling loose. While objects in Java can be tightly coupled if not well designed. It is hard to use tightly coupled classes or objects in isolation from each other [GHJV95].

- While a process or a cell can be used anywhere provided that channels for its ports are available, an object in Java can only function in places where all other objects it calls, directly or indirectly, are available.

- Java recommends that methods are small in size, but not objects, while processes in Erasmus are recommended to "do one thing well" and a cell encapsulates processes with an appropriate interface. In large-scale programs the size of components makes a lot of difference in refactoring, changing, and maintaining the program.

- In Java, multithreading an object may end up active in many different threads (because of coupling), on the other hand, in Erasmus each cell runs as a different thread, this gives the cell more autonomy. However, as mentioned above shared resources among cells or processes still have to be controlled in a higher level in Erasmus.

- Finally, a program in Erasmus could be converted to a Java program, but not vis versa; that is an indication that Erasmus is a higher level of abstractions if compared with Java.

3.6 GoF Patterns and Erasmus

In introducing the patterns for concurrent programming in Java in [Lea99], Lea states that the discussed patterns in the book are extensions or even applications of the patterns that appear in [GHJV95] to concurrent programming problems.

Following the same pattern, we have started, in this document, adapting a selected number of design patterns from [GHJV95] to Erasmus, using the compiler µE.

The design patterns that are adapted to µE in this document are listed based on the category they belong to (as it appears in [GHJV95]—to read definitions of these categories go to Section 2.2).

- Creational Patterns
  - *Factory Design Pattern* that is discussed in Chapter 4. This pattern in µE discusses how to encapsulate the functionality of instantiating cells or processes.
- Structural Patterns in general deal with how to organize and arrange modules to achieve the least degree possible of coupling and the highest degree of cohesion.
  - *Adapter Design Pattern* that is discussed in Chapter 5. This Chapter demonstrates the possible level of adaption in µE. The level in this case is meant protocol, cell, and process levels of abstraction.
  - *Composite Design Pattern* that is discussed in Chapter 6. Similarly to the Adapter Pattern, this pattern discusses the best arrangement of modules in µE, to achieve composition with low coupling and high cohesion. The pattern again discusses composition on three levels of abstraction: protocol, process, and cell. It also covers the limitations of µE related to building this pattern.
- Behavioral Patterns focus on the functionality in building patterns to reduce coupling and increasing cohesion. This concept is also applicable to Erasmus design patterns. The structure of Erasmus is based on processes doing one job well, which means that Erasmus uses the concept of the behavioral pattern in its language abstraction.
  - *Chain Of Responsibility Design Pattern* that is discussed in Chapter 7. This pattern attempts to design chain of functions or behaviors in a way that reduces coupling, and increases cohesion. Decoupling processes that handle a request makes it easier to upgrade and maintain the code.

– *Command Design Pattern* that is discussed in Chapter 8. This pattern demonstrates the possibility of encapsulating a command in Erasmus, and the optimized method of delegating the command to its handlers.

– *Interpreter Design Pattern* that is discussed in Chapter 9. Although, the Interpreter Pattern is used less than some other patterns, it was adapted to Erasmus, to show the flexibility of the language using µE.

– *Iterator Design Pattern* that is discussed in Chapter 10. Although µE implements the Iterator Pattern as part of its structure, as explained in Section 10.2, this chapter demonstrates the possibilities of providing the Iterator functionality using processes and cells.

In discussing these patterns, limitations are tackled as well. Some of these limitations are problems to be solved once the compiler has moved forward from being a prototype (µE) to a production-quality compiler for the full Erasmus language.

3.7 Erasmus and µE features and annotations

The following Sections discuss the main features and important annotations of Erasmus and the difference between the goal of the language and the current prototype µE. These features and annotations are used in building patterns.

3.7.1 Interface and Parameter of a Process or a Cell

In Erasmus, the only way to exchange messages from the outside of a process or a cell to the inside is by means of ports. These ports define the interface of the process or the cell. They are declared after the declaration keywords "process" (in the case of process declaration) and "cell" (in the case of cell declaration) and before the bar or vertical bar character "|". These ports could also be called parameters if we borrow the terminology of parameter from OOP and apply it to Erasmus.

The following piece of code declares the processes "testProc" and "clientProc" and the cells "testCell" and "mainCell" as demonstrated in Figure 1 and the following program. The parameters that define the interfaces of the process "testProc" and the cell "testCell" are: the port "p" ("p" is a variable name) of protocol type "testTxtProt", and the port "q" of protocol type "testIntProt". Both ports "p" and "q" are server parameters (see definition of client ports and serve ports [GS09]). The process "clientProc" has also two ports "p" of protocol type "testTxtProt" and "q" of protocol type "testIntProt" and they are both of type client; they are client parameters. Note that the cell “mainCell” has no ports declared as part of its interface.

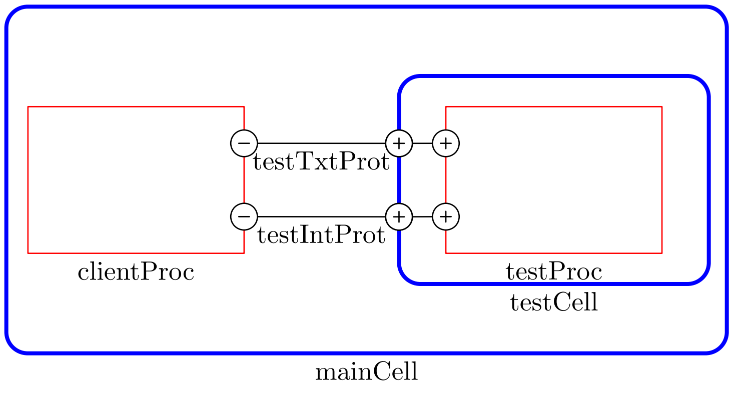

Figure 1: Interfaces and Parameters in Erasmus

```
testTxtProt = protocol
  test: Text
end

testIntProt = protocol
  val: Integer
end

testProc = process p: +testTxtProt; q: +testIntProt|
  sys.out := p.test // q.val // "\n";
end

testCell = cell p: +testTxtProt; q: +testIntProt|
  testProc(p, q);
end

clientProc = process p: −testTxtProt; q: −testIntProt|
  p.test := "I am testing\n";
  q.val := 100;
end

mainCell = cell
  p: testTxtProt;
  q: testIntProt;
  clientProc(p, q);
  testCell(p, q);
end

mainCell()
```

3.7.2 Floating Port

Erasmus's diagrams do not support yet the visual representation of passing a protocol as a message, which is on the other hand well-defined in the syntax and language capabilities. Figure 2 demonstrates a tentative solution I have suggested in my diagrams to demonstrate what we could call a *floating port*.

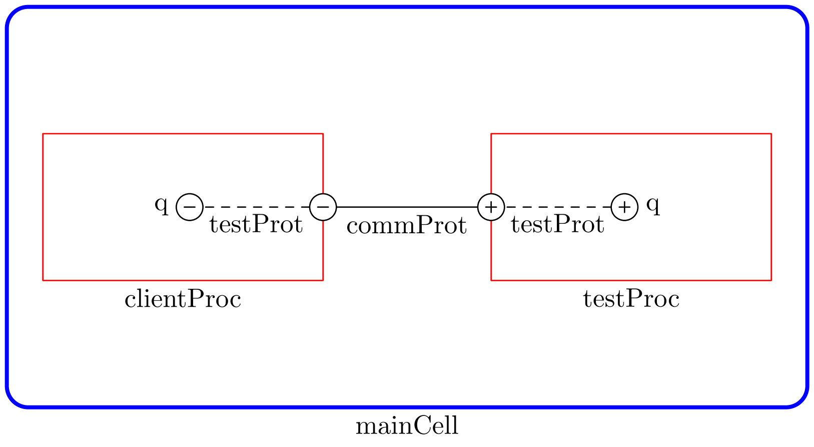

Figure 2: Erasmus Diagram Representation of Floating Port

The following parts are the Erasmus syntactical representation of the floating port demonstrated in Figure 2.

```
testProt = protocol
  test: Text
end

commProt = protocol
  test: testProt
end

testProc = process p: +commProt |
  q: +testProt := p.test;
  sys.out := q.test;
end
...
```

and

```
...
clientProc = process p: −commProt|
  q: −testProt;
  p.test := q;

  q.test := "I am testing\n";
end
```

```
...
```

The protocol “commProt” passes a port, named “test”, of protocol type “testProt”. To pass a message which is in reality a port from the “clientProc” process to the “testProc” process, we need to instantiate dynamically a port “q”, which is a floating port, of protocol type “testProt” inside the “clientProc” process. The same is applicable to the handler process “testProc”.

3.7.3 Reflection in Erasmus

In the current version of µE’s compiler, reflection is not supported. Hence, it is not yet possible to call a protocol, a process or even a cell dynamically (during runtime).

Reflection in Erasmus, as we perceive it, could be that a component, which may be either a cell or a process, is able to detect and analyze its own structure, by knowing the number and names of ports it has, the protocol types associated with these ports, and which of these ports are not connected.

On the level of protocols, reflection could provide information about the names and types of the fields of a protocol.

It is important to note here that µE’s compiler makes sure that all ports are connected. However, in a reflective language, it would be possible to allow some ports to be unconnected until runtime.

3.7.4 Genericity in Erasmus

The current version of Erasmus does not support Generics. This means that types have to be fully specified during programming time. However, this feature is going to be supported in later versions of Erasmus. The Generic feature in Erasmus would be applied to protocols, processes, and cells. The code of generics may look like:

```
testProt<T> = protocol ...
   ...
end

testProc<T> = process ...
   ...
end

testCell<T> = cell ...
   ...
end
```

Calling these components would be something like the following:

...
testProt<Text>(...);
...
testProc<Text>(...);
...
testCell<Integer>(...);
...

3.7.5 Channels in a Fork shape

In Figure 3, the port that belongs to the cell “compositeCell” is connected to three processes “nameProc”, “ageProc”, and “nationalityProc”. This structure we refer to as a “fork-shaped channels”. In previous versions of µE’s compiler, this structure did not produce any compilation error. However, the newest version of the compiler restricts a channel connection to be between exactly two ports. The reason for having this restriction is that in fork-shaped channels, any of the many servers that are connected to the client port may handle the request. The fork-shaped channel does not guarantee that the message goes to its exact target because there are many (in this example three receivers of the request, i.e., three targets). This particular program worked correctly because each server used a different field of the protocol.

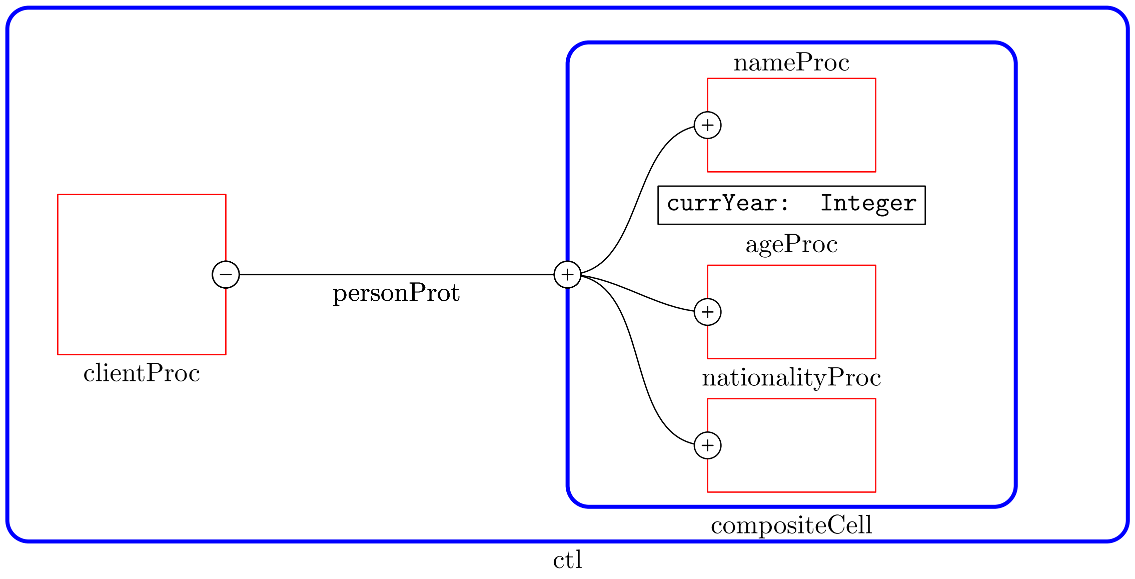

Figure 3: An Old Design of Cell Composition in Erasmus

3.7.6 Declaration Precedence over Instantiation

In Erasmus declaring a component (the component is either a protocol, a process or a cell) must precede instantiation or call. This feature does not affect having recursion in a program. In [GS08] document, a program that computes factorial uses recursion. The piece of code that calls itself recursively is:

```
prot = [ arg: Integer; ↑res: Integer ];

fac = process p: +prot |
  loop
n: Integer := p.arg;
    if n ≤ 1
      then
        p.res := 1
      else
        q: −prot;
        fac(q);
        q.arg := n − 1;
        p.res := n ∗ q.res;
    end
  end
end
...
```

This code does not produce any compilation error because the recursive call of the process “fac” appears after its declaration. However, mutual recursion as it is shown in Section “Primes revisited again” of the document [GS08] is not supported. The following part of the program demonstrates the issue:

```
...
selector = process prime: Integer; nin: +Nats; pout: −Primes |
  prime := nin.nat;
  pin: Primes;
  reporter(prime, pin, pout);
  nout: −Nats;
  tester(nout, pin);
  loop
    cand: Integer := nin.nat;
    if cand % prime ≠ 0
      then nout.nat := cand;
    end
  end
end
...

...
tester = cell nin: +Nats; pout: −Primes |
  prime: Integer;
  selector(prime, nin, pout)
end
```

..

The process “selector” calls the cell “tester”, and the cell “tester” calls the process “selector”. This is called mutual recursion, and this feature produces a compilation error.

3.7.7 Erasmus Programming Annotation

There are some code conventions and annotation related to the programs used in explaining the Design Patterns; these annotation are explained in the following points:

- In writing to the console, variables have to be converted to a Text type. To achieve this, the function “text” is used to convert variables of any type to a Text type. However, in latest versions of Erasmus, the compiler provides a new operator “//” instead. Since my design programs were written before these new changes took place, the programs do not follow this syntax. The following example demonstrates the syntax in both cases: the old and the new version (“p.result” variable is of type Integer).

  Old syntax:

  ```
  ...
    sys.out := "Display: " + text p.result + "\n";
  ...
  ```

  New syntax:

  ```
  ...
    sys.out := "Display: " // p.result // "\n";
  ...
  ```

- In Erasmus the equivalent syntax to the “main method” in Java or C++ would be something like:

  ```
   ...
  mainCell = cell
     ...
  end

  mainCell()
  ```

  Though, the cell name could be any other chosen name that is not a keyword. So we may find in some examples the main process as “ctl” or any other name.

- Up to this version of the compiler, a signal that is usually named “stop” is used in the “loopselect” construct to indicate that the loop should be exited. However, the “stop” signal is not part of the core language of Erasmus, rather it is introduced in the prototype compiler for simplicity. The final version of the compiler will not need this mechanism to quit.

- Maps in Erasmus cannot be passed as messages from one element to the other. Sharing Map aggregate type is only possible within a cell and its corresponding processes.

Chapter 4

Creational Pattern: Factory Design Pattern

The Creational category is a classification that appears in the book [GHJV95]. It groups design patterns that are concerned with the instantiation of components. In OOP, the creational category groups design patterns that instantiate objects. In POP, the design patterns instantiate either processes or cells.

4.1 Factory Method Design Pattern in OOP

The Factory Method encapsulates the creation of an instance of an object and the knowledge of the type (kind) of that object. All possibly created objects in the Factory Method share the same interface.

The discussion of this chapter is based on [GHJV95], [Coo00], [Met02], and [FFBS04].

4.1.1 Definition

"Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses" [GHJV95].

4.1.2 Background

We need first to define the meaning of a "kind of a class" that is going to be used in this chapter. The kind of a class is the derived type of a class. The superclass is called the "Product" class, and the derived (or the kind) class is called the "ConcreteProduct" class (both terms "Product" and "ConcreteProduct" are shown in the class diagram in Figure 4 and explained in Section 4.1.3).

A user that creates kinds of a class (a class hierarchy) may need to know when to create instances of a class, but not what particular kind to create. Putting the decision code at the creating class produces an incoherent and messy code, because we bind specific class details (the details of different subclasses) of the class hierarchy with the creator class. A solution to this problem was introduced by [GHJV95] that is called the Factory Method Pattern.

A Factory method behaves as a manufacture of objects. It helps its user to create an object without knowing the kind (type) of the object. In this way, this Pattern defers the decision of the kind of the created object to the Factory Method instead of having the user of the method to decide it. As a result, the client needs only to deal with the interface of the created object (that will be called a "Product"). Hence the client is decoupled from the different kinds of the created object.

4.1.3 Structure and Actors

The class diagram of the Factory Method in object-oriented programming is shown in Figure 4. The participating classes are explained here:

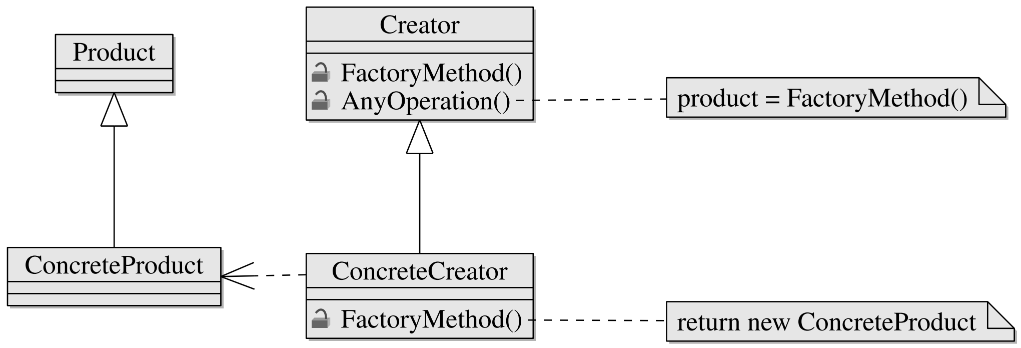

Figure 4: Factory Method Design Pattern in Object-Oriented Programming

- Product: represents the interface of the created object in the "FactoryMethod" method.
- ConcreteProduct: is the actual implementation of the "Product" interface. An application consists of many "ConcreteProduct" classes that implement the "Product" interface; these classes are called kinds of the "Product" class. When a "FactoryMethod" is called, it creates an instance of a kind of the "ConcreteProduct" class and returns it back to the caller.
- Creator: is a class that could be either an abstract class (interface) or a concrete

class depending on the application. It declares (and defines if it is a concrete class) a "FactoryMethod" method. The "FactoryMethod" creates a "ConcreteProduct" object and returns it to the caller of the method.

- ConcreteCreator: is an implementation of the "Creator" interface (if the "Creator" is an interface), or is a subclass of it. It either inherits the default implementation of the "FactoryMethod" or it overrides it depending on the requirement of the application.

4.1.4 Implementation Decisions

- "Factory Methods eliminate the need to bind application-specific classes into your code. The code only deals with the Product interface; therefore it can work with any user-defined ConcreteProduct classes" [GHJV95].

- The Factory Method centralizes the knowledge of creating associated classes. So as in Figure 4, the Factory Method encapsulates the knowledge of which particular "ConcreteProduct" to create, depending on which "ConcreteCreator" object is requesting the action of creation.

- The "Creator" class can be implemented in two ways:

  1. It can either be declared as an abstract class. In this case, the "FactoryMethod" method it declares has no implementation, instead the implementation of the method is deferred to the subclasses ("ConcreteCreator").
  2. Or it can be declared as a concrete class, which allows the "FactoryMethod" method to implement a default behavior. As a result, the subclasses ("ConcreteCreator") have the choice either to take the behavior of the "FactoryMethod" from the parent class ("Creator"), or to override the behavior with another that suits them.

- A "FactoryMethod" method could be parameterized. The parameter that is passed to the method defines the type (kind) of object that is to be created in the method (which particular instance "ConcreteCreator" to create).

- In cases when a client of the "Creator" class does not need subclasses of the "Creator" class, a template solution is possible [GHJV95]. A template subclass of the "Creator" class is created. This way the template subclass will be passed the right instance of the "Product" (which is the "ConcreteProduct") as a template parameter. However, this method cannot be implemented in all programming languages, because some languages do not support templates.

- The Factory Method allows connections between parallel hierarchies. "A Parallel hierarchy is a pair of class hierarchies in which each class in one hierarchy has a corresponding class in the other hierarchy" [Met02]. One class hierarchy represents the "Product" interface and its subclasses and the other is the "Creator" class (or abstract class) and its subclasses.

4.1.4.1 Java Implementation

In Java, the Factory Pattern meets the general design of the above discussion. The "Creator" as per the examples in [FFBS04], [Coo00] and [Met02] is declared as an abstract class in which an abstract method that represents the Factory Method is declared. The implementation of this method is delegated to the subclasses (the classes that are represented by the class "ConcreteCreator" in the class diagram of Figure 4).

The following points help in taking the decision for whether to use the Factory Method Design or not:

- To Create a new object.
- The creating class cannot anticipate which kind of class of object to create.
- The created object is a subclass of an abstract class or an implementation of an interface.
- The need to localize the knowledge about the created object.
- The method is implemented by several classes.

In fact Java follows the Factory Method Design Pattern in building its own Class Libraries. A Java Class Library that implements this Factory Method design is the Iterator Class (see the Java Iterator Class in 10.1.4.1).

4.2 Factory Design Pattern in Erasmus

The concept of the Factory Pattern in Erasmus is not that different from the Factory Method in OOP. While a Factory Method in OOP creates objects, the Factory Pattern in Erasmus creates processes. The client of this Pattern does not know which process is to be created. In this design the Factory Process encapsulates the knowledge of the creation of other processes. The decision of creation is delegated to the Factory Process instead of having it implemented in the client side. The Factory Process localizes the creation of processes. This helps decoupling requesting (clients) processes from their handles (servers).

The above description shows how processes can be created, but the Factory Pattern can also be used to create cells.

The demonstrative program of the Pattern is a simple example thats aim is to show the essential components that implement the Pattern.

4.2.1 Structure and Actors

Figure 5 demonstrates the Factory Pattern in Erasmus. The protocols, processes, and cells of the Factory explanatory program are analyzed here.

Note that the term "Product Process" that could be extended to "Product Cell" is a term I have borrowed from the Factory Method Pattern in OOP "Product" or "ConcreteProduct" that is shown in the class diagram in Figure 4. The Product process is a process that is created in the Factory Process and that handles requests from the Client Process.

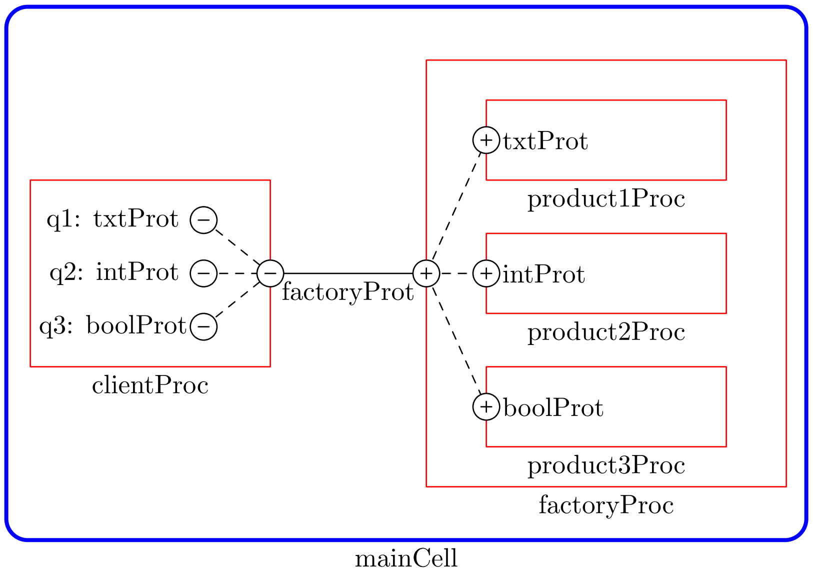

Figure 5: Factory Design Pattern in Erasmus

- Protocols
  - txtProt: is a protocol that consists of two elements a Text typed variable, and a signal "stop". This protocol is associated with the process "product1Proc".
  - intProt: is a protocol that consists of two elements an Integer typed variable "num", and a signal "stop". The protocol is associated with the process "product2Proc".

  - boolProt: is a protocol that consists of two elements a Boolean typed variable named “val” and a signal “stop”. The protocol is associated with the process “product3Proc”.
  - factoryProt: consists of three ports of protocol types “txtProt”, “intProt”, and “boolProt”. This protocol sends ports between processes and is an important element in building up the Factory Pattern in Erasmus.

- Processes

  - product1Proc: is a simple demonstrative example of a Product Process that is created by the “factoryProc” process. This process concatenates texts that are sent from the “clientProc”. This functionality is just a demonstrative behavior of what a Product Process can do. It is not bound to the Factory Pattern.
  - product2Proc: is another simple demonstrative example of a Product Process that is created by the “factoryProc” process. This process sums up integer values that are sent from the client process “clientProc”.
  - product3Proc: is a third simple demonstrative example of a Product process that can also be created by the “factoryProc” process. This process only outputs to the console the boolean value that is sent from the client process “clientProc”.
  - factoryProc: implements the Factory Pattern that is under discussion. The parameter that defines the interface of this process is of protocol type “factoryProt”.
  - clientProc: is the user process of the factory process.

- Cells

  - mainCell: instantiates the processes “clientProc” and “factoryProc” and connects them with a channel of protocol type “factoryProt”.

4.2.2 Consequences

As in OOP Pattern, the Factory Pattern in Erasmus can be extended to support parameters that identify the process to be created in the Factory Process. To demonstrate this point, we could extend the factory protocol “factoryProt” to the following:

```
factoryProt = protocol
  *(param: Integer; (val1: txtProt| val2: intProt| val3: boolProt));
  stop
end
```

or:

```
factoryProt = protocol
  ∗(param: Boolean ;(val1: txtProt| val2: intProt));
  stop
end
```

or even the following, if the Product Processes (Cells) have the same parameters:

```
factoryProt = protocol
  ∗(param1; val1: txtProt| param2; val2: txtProt| param3; val3: txtProt);
  stop
end
```

In the following product processes “product1Proc” and “product2Proc” note the similarity in their functionality, although the parameters of both processes are different:

```
...
product1Proc = process p: +txtProt|
  conct: Text := "";
  loopselect
     || temp: Text := p.txt; conct += temp;
     || p.stop; exit;
  end;
  sys.out := "Server One: " // conct // CR;
end

product2Proc = process p: +intProt|
  sum: Integer := 0;
  loopselect
     || temp: Integer := p.num; sum += temp;
     || p.stop; exit;
  end
  sys.out := "Server Two sum: " // sum // CR;
end
...
```

This piece of code could be reduced to the following process if the current version of Erasmus supported the Generic feature (Section 3.7.4 provides details about Genericity in Erasmus).

```
...
valProt<T> = protocol
  ∗(val: T);
  stop
end
...
productProc<T> = process p: +valProt<T>|
  conct: T := "";
  loopselect
     || temp: T := p.val; conct += temp;
     || p.stop; exit;
  end;
  sys.out := "Server : " // conct // CR;
end
```

...

Until Erasmus supports Generics, the Factory Pattern has to be implemented when a Generic behavior is required.

Also, the Factory Pattern is used in Erasmus when one or many processes (or cells) are created, and the knowledge of what process (cell) to create is wanted to be separated from the requesters of the service.

4.2.3 Implementation

The developed program demonstrates the Factory Pattern in a simple form. In more realistic applications, the Product Process or Cell is composed of many other processes or even cells that handle together the requested functionality. The protocol of the Product Process (Cell) is not restricted to elements of primitive data types; they can be complex.

The Factory Pattern encapsulates the creation of handlers of requests (as in the previous sections). This means that the created processes (cells) are always of type servers (their parameters are server parameters— for more information about server and client parameters refer to Section 3.7.1).

The factory protocol has to be a super set of the protocols of the handling processes (cells) it instantiates as shown below:

```
factoryProt = protocol
  *(val1: paramOfProcessOneProtocol| val21: paramOneOfProcessTwoProtcol;
val22: paramTwoOfProcessTwoProtocol| val3: ....);
  stop
end
 ...
ProcessOne = process p: +paramOfProcessOneProtocol|
 ...
end
 ...
```

The protocol supports sending the elements “val1”, “val21” and “val22”, or “val3” many times. Each time an element is sent a new process is instantiated (depending on the sent element). So for example, every time a “val1” element is sent a new product process “ProcessOne” in the example, is instantiated.

4.2.4 Code

The key parts that demonstrate the Factory Pattern in Erasmus are:

- The factory protocol:

```
factoryProt = protocol
  *(val1: txtProt| val2: intProt| val3: boolProt);
  stop
end
```

- The factory process:

  The Factory Process must be a process and not a cell, because a cell can only instantiate processes or other cells and connect them together. If we change the Factory Process “factoryProc” to a Factory Cell “factoryCell”, as shown in the following piece of code, the Erasmus compiler produces a compilation error for having the statement “loopselect...end” in a cell.

  ```
  factoryCell = cell p: +factoryProt |
    loopselect
      || product1Proc(p.val1);
      || product2Proc(p.val2);
      || product3Proc(p.val3);
      || p.stop; exit;
    end
  end
  ```

  The only compilable code that includes a factory cell is the following piece of code. The factory cell encapsulates the factory process; this is not that different, in reality, from having the “factoryProc” alone.

  ```
  factoryProc = process p: +factoryProt |
    loopselect
      || product1Proc(p.val1);
      || product2Proc(p.val2);
      || product3Proc(p.val3);
      || p.stop; exit;
    end
  end

  factoryCell = cell p: +factoryProt |
    factoryProc(p);
  end
  ```

The complete code and its results are found in Appendix A.

4.3 Summary

In this chapter we have shown how to implement the Factory Pattern in Erasmus. The Erasmus implementation differs from the OOP implementation in that Erasmus abstraction does not support interfaces as in OOP (of processes or cells). So controlling which process to create at runtime is achieved by sending messages from the client to the factory process. The factory process, in turn, decides (knowing the port type from which the message is sent) which process to instantiate and to pass the message to. An advantage in Erasmus over OOP lies in the fact that “Product” processes (the processes that are created dynamically) are encapsulated in one factory process. This makes the factory process highly coherent and the encapsulation ensures low coupling between the participating processes. However, the disadvantage of the current version of Erasmus, µE, is the inability to support Generic behavior. If the Generic behavior was supported, it would have substituted the Factory Pattern in some cases.

Chapter 5

Structural Pattern: Adapter Design Pattern

The Adapter Pattern belongs to the Structural category among other patterns, a classification that appeared in the book [GHJV95]. The Structural category groups design patterns that are concerned with arranging modules that build up the pattern. The aim of this family of patterns is to achieve the least of coupling and the highest of cohesion between the components of the structural pattern.

5.1 Adapter Design Pattern in OOP

The Adapter is an object that wraps another object (called adaptee) to be able to function with a third object (called client) that has incompatible interface with the adaptee.

5.1.1 Definition

"Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of the incompatible interfaces" [GHJV95].

5.1.2 Background

In some cases, already existing components (that could be a class or a set of classes working together, a package or a library) can solve (either completely or partially) a problem. However, the interfaces of these components do not match the requirements. This is when the Adapter Pattern comes in handy. This Pattern wraps the component with an interface that fulfills the requirement of the user class.

The Adapter Pattern is very logical, and could be thought of intuitively. Add to that it is very useful, because it makes re-using code possible and cheap. In business applications, changes of requirements are frequent and a system cannot be extremely robust to the point it could support many changes without adaptation. To meet deadlines and to refactor with minimum costs, the Adapter Pattern is indeed a very powerful solution.

5.1.3 Structure and Actors

The original design of the Adapter Pattern in object-oriented paradigm has two possible variants, represented in Figures 6 and 7. Both are explained in detail in Section 5.1.4

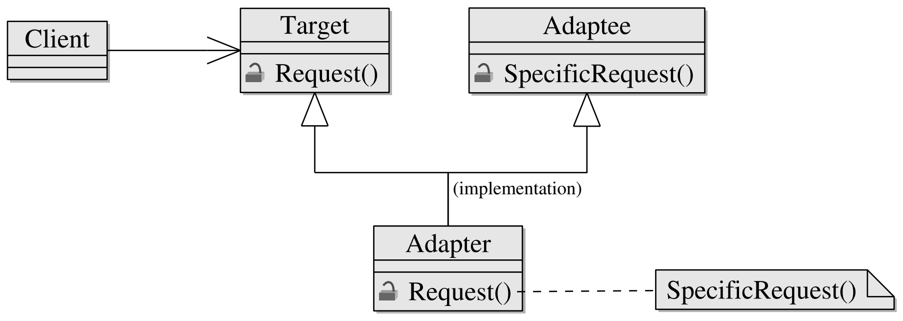

Figure 6: Class Adapter Design Pattern in Object-Oriented Programming

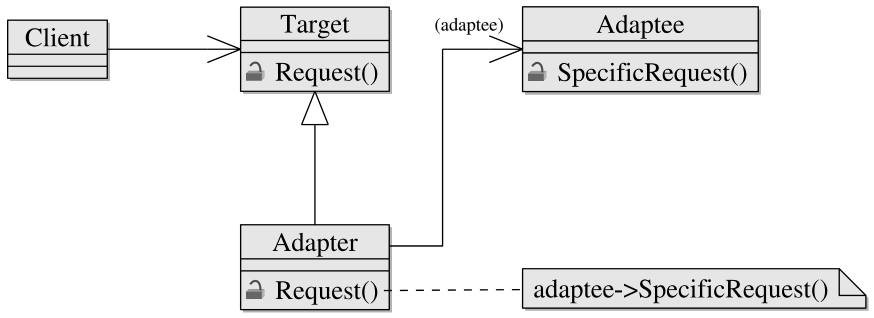

Figure 7: Object Adapter Design Pattern in Object-Oriented Programming

The actors that make up the design pattern are:

- Client: is the user of the Adapter Design Pattern.
- Target: is the interface or model which the Adapter should implement to meet client's expectations. Note that in some cases the Target and the Client are in reality only one component [GHJV95].

- Adapter: is the element that makes it possible for both the Target and the Adaptee to interoperate [Gea03].
- Adaptee: is the component from which the adapter is, in reality, providing functionalities. The Adapter wraps the Adaptee with an interface that meets the requirements of the Client (or Target).

5.1.4 Implementation Decisions

In object-oriented programming an Adapter Pattern can be in two flavors: Class Adapter Pattern (Figure 6) and Object Adapter Pattern (Figure 7).

To choose which type of the Adapter Pattern is to be used in building an application during design phase, the following points help in taking a decision:

- If the language supports multiple inheritance, the Class Adapter Pattern is implemented by just following the class diagram (Figure 6). However, in some languages, such as Java, where multi-inheritance of classes is restricted, an alternative way could be suggested. For example in Java, extending a class and implementing many interfaces is possible, so this criteria is used to build the Class Adapter Pattern.
- If subclasses of an adapted (Adaptee) are needed to be incorporated in the new system, then the Class Adapter Pattern is not that helpful, simply because subclassing a class does not give it access to the other subclasses of that class.
- If the Adaptee's methods are not to be overridden, then the Class Adapter Pattern is not suitable; rather the Object Adapter Pattern is the solution because the Adapter is composed of the Adaptee but is not an inheritance of it. This ensures that these methods cannot be overridden.
- When having a composition of the Adaptee in the Adapter is not wanted, a Class Adapter Pattern is better.
- If the Adapter is a composition of many Adaptees, then the Object Adapter Pattern is a good solution.
- The degree of adaptation depends on how different the Adaptee is from the Target (or Client). In some cases new functionalities have to be added to the Adapter in order to fulfill the requirements of the client. At other times, only minor additions are performed on the Adapter, such as changing signatures of methods.

- Smalltalk introduced the term "pluggable Adapter". A Pluggable Adapter is an adapter that adapts classes dynamically at runtime (not during compile time as the general Adapter Pattern stands for). To do so, the adapter has to have a mechanism of knowing the class content during runtime.
- A two-way Adapter is a concept used when an Adapter is designed in such a way to be able to support both the Adaptee's functionalities by not overriding them, and the Target's functionalities at the same time.

5.1.4.1 Java Implementation

In Java, the implementation of the Object Adapter Pattern is achieved by having the Adapter composed of the Adaptee (among other objects), and by implementing the Target interface. The Class Adapter Pattern, on the other hand, is implemented by subclassing the Adaptee and implementing the Target which is an interface.

A Pluggable Adapter is implemented by using reflection. Reflection in Java makes it possible to know the name of the class, the name of the method, and the name of the parameters dynamically (at runtime) [Coo00].

A two-way Adapter in Java is achieved by ensuring that the methods of the Adaptee are not overridden in the Class Adapter Pattern type, or by making the methods of the Adaptee public in the Object Adapter Pattern type.

5.2 Adapter Design Pattern in Erasmus

Since an interface of a component (either a process or a cell) in Erasmus is defined by its parameters (as explained in Section 3.7.1), then adapting a cell or a process to another is done by incorporating the same port of the Target in constructing the Adapter's port. From there the Adapter should transform the Target's port to the port type of the Adaptee, in order to pass the appropriate messages to the Adaptee. Therefore an Adapter in Erasmus, being a cell or a process, wraps the Adaptee (which can also be either a cell or a process).

In Erasmus, adaptation is easier than in the object-oriented paradigm, because a functionality in a process or a cell is enclosed in its process and the process behaves like a black box for the outside world, not like an object in which public methods are accessible directly by other objects. This black box, in the case of Erasmus, may be accessed through its ports (if any), and not by methods that carry out the functionalities of the cell or the process. Hence the part that is exposed to other processes or cells is much less, and as a result adapting a process or a cell is much easier.

To demonstrate the Adapter Pattern in Erasmus, I have developed two methods: the Protocol Adapter Pattern and the Process Adapter Pattern. The Protocol Adapter Pattern is based on adapting two processes with different protocol type parameters. The Process Adapter Pattern works by adapting a new functionality to an interpreting program, by exchanging the tree based grammar with a post-fix grammar without redeveloping the whole program.

5.2.1 Protocol Adapter Pattern

In this case an Adapter Process (or cell) wraps another process (or cell) with a different protocol type. The goal is to adapt two different protocols by passing the relevant messages from one protocol type to the other.

In the demonstrated example (also as in Figure 8), the aim is to provide the Client process "clientProc", which sends messages from a port of protocol type "targetProt", with the functionality of the legacy process "serverCompositeProc" that handles messages of protocol type "triangleProt". Both processes cannot function together, because of their incompatible interfaces. To solve the incompatibility issue, the process "adapterProc" is created as the Adapting Process to wrap the legacy component (the Adaptee Process) in order to meet the requirements of the Client (by adapting to the target protocol type "targetProt").

5.2.1.1 Structure and Actors

In this design the Protocol, Process and Cell structure is as in Figure 8:

- Protocols
  - lineProt: protocol that represents a line, and that is represented by a primitive data type.
  - triangleProt: is a channel of three lineProt protocol types. This protocol is the Adaptee Protocol.
  - targetProt: is a channel of two lineProt protocol types, and it represents the protocol of the Target.
- Processes
  - serverCompositeProc: is a process whose functionality is needed but with another protocol type. This represents the Adaptee Process and it is a legacy component.

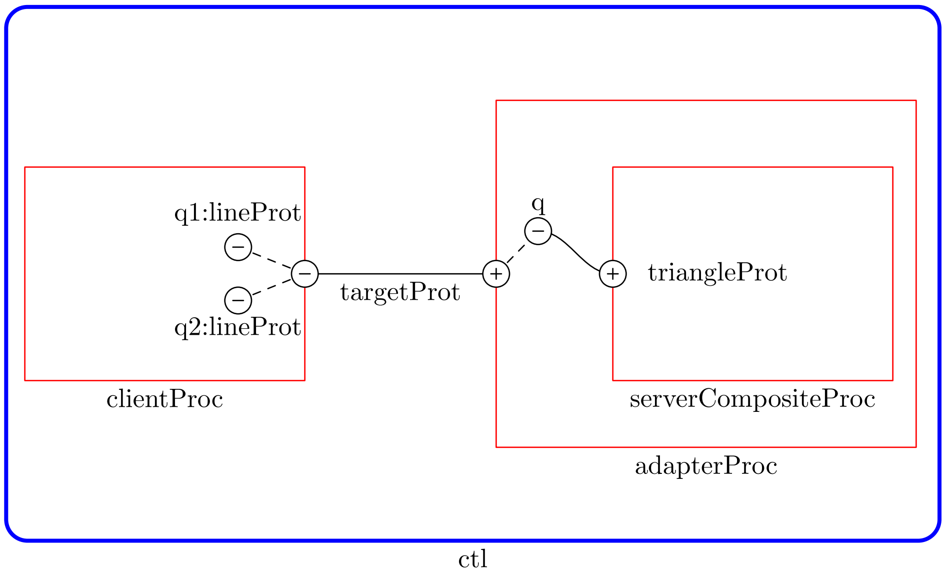

Figure 8: Protocol Adapter Design Pattern in Erasmus

  - adapterProc: this process works as an adapter from “targetProt” protocol to triangleProt.
  - clientProc: a client process that passes the data to the server, but the client requires protocol of type “targetProt” protocol. This is the Client of the Pattern (the Term “Client” refers to what is the equivalent in OOP as in Figure 7) .

- Cells
  - ctl: cell to run the program.

5.2.1.2 Consequences

First, the protocol of the Target process (or cell) does not have to be a super set of the protocol of the Adaptee (which is in this case the “serverCompositeProc” process). Obviously, if we are trying to adapt one process (or cell) to another, we need to pass messages to the Adaptee. So even primitive types of protocols (such as Text, Integer, etc) are allowed in both.

Second, a two-way Adapter is also supported in Erasmus because both processes (or cells), the Adaptee and the Adapter, could be used independently by the client at the same time without having to change anything in the Adaptee. The following piece of code is an extension of the program demonstrated in Figure 8. We just need to change the client process and the main cell, to make the client process “clientProc” work with the adaptee process "serverCompositeProc" and the adapter process. It is true, that the way Erasmus handles two-way Adapter is different from the OOP way, in the sense that in OOP, the object encapsulates both behaviors: the ones of the Adaptee and of the Adapter in one.

```
clientProc = process p: −targetProt; q: −triangleProt|
  p1: −lineProt;
  p.line1 := p1;
  p1.point := 1;
  p2: −lineProt;
  p.line2 := p2;
  p2.point := 2;

  q1: −lineProt;
  q.line1 := q1;
  q1.point := 10;
  q2: −lineProt;
  q.line2 := q2;
  q2.point := 20;
  p3: −lineProt;
  q.line2 := q2;
  q2.point := 30;
  q.stop;
end;

ctl = cell
  p: targetProt;
  q: triangleProt;
  clientProc(p, q);
  adapterProc(p);
  serverCompositeProc(q);
end
```

Finally, the Pluggable Adapter is not supported currently in Erasmus simply because Erasmus does not support reflection, as discussed in Section 3.7.3.

5.2.1.3 Implementation

In the implementation of this pattern, and to emphasize on the concept of re-using existing code, I have reused an already developed protocol "triangleProt" (see Section 6.2.2).

The "adapterProc" process creates dynamically a protocol of type "triangleProt", gets the needed information from the Target's protocol "targetProt", and then passes them to the Adaptee process "serverCompositeProc".

5.2.1.4 Code

The full code and the results of the demonstrative example of the Protocol Adapter Pattern are found in Appendix B.1. The protocols that interoperate are:

```
lineProt = protocol
  point: Integer
end;

triangleProt = protocol
  line1: lineProt;
  line2: lineProt;
  line3: lineProt;
  stop
end;

targetProt = protocol
  line1: lineProt;
  line2: lineProt
end;
 ...
```

The Protocol Adapter is implemented in the process "adapterProc". The process adapts from the Adaptee Protocol "triangleProt" to the Target Protocol "targetProt":

```
 ...

adapterProc = process p: +targetProt |
  q: −triangleProt;
  serverCompositeProc(q);
  q.line1 := p.line1;

  q.line2 := p.line2;
  q.stop;
end

 ...
```

The "adapterProc" process could even be extended to pass information through the message "line3.point" that belongs to the port "q" of protocol type "triangleProt" as demonstrated in the following code. Passing a data through the message "line3.point" or not does not change a lot in the design of the Adapter process "adapterProc", or in the Adaptee process "serverCompositeProc". The difference between both codes is that the process "serverCompositeProc" handles the data sent through the message "line3.point".

```
 ...

adapterProc = process p: +targetProt |
  q: −triangleProt;
  serverCompositeProc(q);
  q.line1 := p.line1;
  q.line2 := p.line2;

  r: −lineProt;
  q.line3 := r;
  r.point := 100;
  q.stop;
end

 ...
```

The output of the extended program is:

```
Line1: point: 1
Line2: point: 2
Line3: point: 100
```

5.2.2 Process Adapter Pattern

This pattern demonstrates another aspect of adaptation. It demonstrates how a functionality can be converted to another without having to change the rest of the program. The adaption is not done in this case by converting the interface of the client to work with the target, rather the functionality (which is encapsulated in a process) is replaced altogether to do something else, and the interface of the new process has to be similar to the one of the old process. To demonstrate this, an interpreter program was developed to work with a tree-based grammar. The adaptation here facilitates changing the functionality from a tree-based grammar to a post-fix one. The legacy program is the interpreter program (that includes the processes “scannerProc”, “expressionBuilderProc”, and “converterProc” and the cell “aCell”) and the adapted process is the “evaluateProc” process.

5.2.2.1 Structure and Actors

The protocol, process, and cell structure of the Process Adapter Pattern is as listed and shown in Figure 9

- Protocols
  - tokProt: is a protocol that breaks down the given expression into valid tokens. Tokens consist of an enumeration type “kind” that in turn consists of “number”, “name”, and “op”; and a “value”.
  - tokValprot: is similar to “tokProt”, but a “stop” signal is added to it.
  - expressionProt: consists of the valid terminals of the grammar (operations set: “plus”, “minus”, and “multiplication”), and “integers”.
  - commProt: is a channel that passes a port of protocol type “expressionProt”.
- Processes

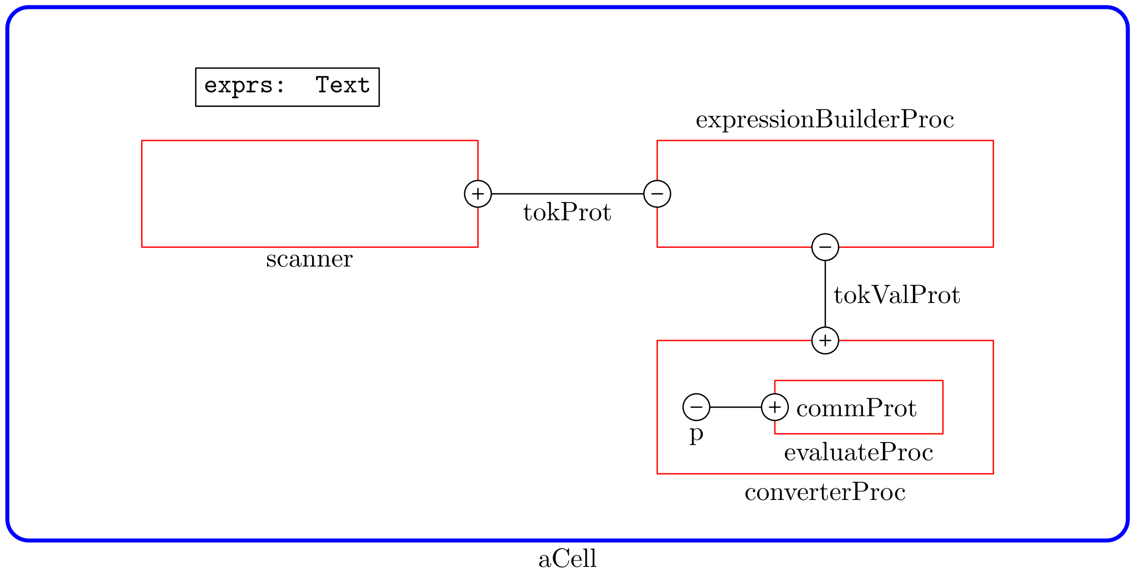

Figure 9: Process Adapter Design Pattern in Erasmus

- scannerProc: scans the input and converts it into tokens.
- evaluateProc: replaces the grammar by an "IntegerExp", that is the result/-translation of the expression.
- expressionBuilderProc: builds the given tokens of the given expression into a valid grammar.
- converterProc: converts the tokens into a valid grammar.

• Cells

- aCell: arranges the messages between processes, instantiates the processes "scannerProc", "converterProc", and "expressionBuilderProc", and runs the program.

5.2.2.2 Consequences

In this demonstration of the Process Pattern in Erasmus, the process "evaluateProc" is invoked in the process "converterProc". The Pattern demonstrates that it is possible to replace a functionality of a process with another. Having the same name for both (the implementation using post-fix grammar, and the one using tree-based grammar) is only meant to demonstrate that they both do the same job: evaluation of the grammar. However, each of them does it differently. The design could be even extended to give the possibility to the user to instantiate any required functionality during runtime. The example could be extended, by adding a Factory Process as follows:

```
...
factoryProt = protocol
  ∗(cond1; message1: commProt | cond2; message2: commProt);
  stop
end

...

factoryProc = process p: +factoryProt|
  loopselect
    || p.cond1; evaluateTreeBasedProc(p.message1);
    || p.cond2; evaluatePFixProc(p.message2);
    || p.stop;  exit;
  end
end
...
```

In the process “factoryProc”, depending on the passed condition (either “cond1”, “cond2” or even both) a different functionality is instantiated and is passed messages. This way, any functionality or both can be instantiated, depending on the requirement.

5.2.2.3 Implementation

This Pattern is meant to demonstrate that Erasmus supports as part of the language changes or adaptation of functionalities without having to add more complexity to the program.

In object-oriented programming, an object has a state and methods that perform functionalities. The functionalities of an object cannot be separated from the object itself; as a result, adapting functionalities from an object has to be done on the object and not on the functionalities alone, either through object composition or inheritance.

However, in Erasmus, functionalities are independent: they are encapsulated in processes. Hence, getting new functionality or incorporating a functionality from an already developed process is made easy.

5.2.2.4 Code

This is the version of the evaluator process “evaluateProc” implements a tree-based grammar. Note that the parameter of the process is of protocol type “commProt”, and it is the same as in the second implementation of the grammar that follows the post-fix method.

```
...

evaluateProc = process p: +commProt|
  q:  +expressionProt := p.prot;
```

```
    stack: Integer indexes Integer;
    sum: Integer := 0;
    counter: Integer := −1;
    loopselect
      || number: Integer := q.number; counter +=1;
        stack[counter]:= number;
        sys.out := "Number Received: " + text number + "\t counter: " + text counter + "\n";
      || q.minus; sys.out := "Minus Sign\n";
        sum := stack[counter];
        sys.out := "Counter Value: " + text counter + "\t stack" + text stack[counter] + "\n";
        number:Integer := q.number;
        sum −= number;
        stack[counter] := sum;
        sys.out := "Counter Value: " + text counter + "\t stack" + text stack[counter] + "\n";
        sum := 0;

      || q.plus; sys.out := "Plus Sign\n";
        sum := stack[counter];
        sys.out := "Counter Value: " + text counter + "\t stack" + text stack[counter] + "\n";
        number:Integer := q.number;
        sum += number;
        stack[counter] := sum;
        sys.out := "Counter Value: " + text counter + "\t stack" + text stack[counter] + "\n";
        sum := 0;
      || q.multiplication; sys.out := "Multiplication Sign\n";
        sum := stack[counter];
        sys.out := "Counter Value: " + text counter + "\t stack" + text stack[counter] + "\n";
        number:Integer := q.number;
        sum *= number;
        stack[counter] := sum;
        sys.out := "Counter Value: " + text counter + "\t stack" + text stack[counter] + "\n";
        sum := 0;

      || q.stop; sys.out := "Result: " + text stack[0] + "\n"; exit;
    end
  end

  ...
```

And this process "evaluateProc", is the post-fix grammar version that was plugged into the code to provide a different functionality.

```
  ...

  evaluateProc = process p: +commProt|
    q: +expressionProt := p.prot;
    stack: Integer indexes Integer;
    sum: Integer := 0;
    counter: Integer := −1;
    loopselect
      || number: Integer := q.number;
        counter +=1;
        stack[counter]:= number;
        sys.out := "Number Received: " + text number + "\t counter: " + text counter + "\n";
      || q.minus; sys.out := "Minus Sign\n";
        sum −= stack[counter];
        sys.out :=  "Counter Value: " + text counter + "\t stack" + text stack[counter] + "\n";
        counter −= 1;
        sum += stack[counter];
        sys.out :=  "Counter Value: " + text counter + "\t stack" + text stack[counter] + "\n";
        stack[counter] := sum;
        sys.out :=  "Counter Value: " + text counter + "\t stack" + text stack[counter] + "\n";
        sum := 0;

      || q.plus;  sys.out := "Plus Sign\n";
        sum += stack[counter];
        sys.out :=  "Counter Value: " + text counter + "\t stack" + text stack[counter] + "\n";
        counter −= 1;
        sum += stack[counter];
        sys.out :=  "Counter Value: " + text counter + "\t stack" + text stack[counter] + "\n";
        stack[counter] := sum;
        sys.out :=  "Counter Value: " + text counter + "\t stack" + text stack[counter] + "\n";
        sum := 0;
      || q.stop;  sys.out := "Final Sum: " + text stack[0] + "\n";
        exit;
    end
  end

  ...
```

5.3 Summary

In this chapter we have shown how to implement the Adapter Pattern in Erasmus. The Erasmus implementation differs from the OOP implementation in that Erasmus abstraction is based on the encapsulation of actions (a process doing a job). Hence implementing the Adapter Pattern could be seen as being part of the language. While adapting an object in OOP (through object composition or even inheritance) may lead to carrying unneeded states or methods from the adaptee, adapting a process in Erasmus is very specific; it bounds the adaptation to a specific functionality the adaptee process provides (following the main principle of a process "do one thing well"). The Adapter Pattern in Erasmus could be found in two types: Protocol Adaptation and Process Adaptation.

The Erasmus version has an advantage when compared to the OOP version in that implementing the two-way adaptation does not require any additional work, if compared with the two-way adaptation in OOP. However, a disadvantage in the current version of Erasmus, µE, is the absence of the Pluggable feature by reflection that is supported in most OOP paradigms.

Chapter 6

Structural Pattern: Composite Design Pattern

6.1 Composite Design Pattern in OOP

The basic idea of the Composite Pattern is that it allows individual objects as well as groups of them to have the same interface, so they are both treated the same way. The reason for doing that is to make access to objects or groups of them uniform, easier to use, and easier to maintain.

6.1.1 Definition

"Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and composition of objects uniformly" [GHJV95].

6.1.2 Background

A design of a system that represents a company's employees, for example, with the information about each employee including the department the employee belongs to, and her/his salary as well as information about the departments in relation to their employees' salaries, number of employees in each department and maybe in the whole company, etc. Such a design becomes easier to work with if the design enables a unified interface to process both the part which is represented by the employee object, and the whole represented by the departments and the company objects. Note here the relationship between an employee, a department, and the company: the department is a collection of employees, and the company is a collection of departments. The Composite Pattern is useful for this case because it makes it possible to treat an employee, a department, and the company in the same way.

Treating primitive components differently from containers of these components makes the system more difficult and less flexible from the point of view of the system (or subsystem) that is trying to use this design (this user will be called here the client). Furthermore, like trees, nodes (branches) cannot be separated from leaves. Separating leaves from nodes weakens the concept of part-whole relationship between the leaves and the nodes, and makes it just a collection of different parts that are not correlated. As a result, the Composite Pattern in [GHJV95] was designed to have a uniform interface that represents both: the leaf and the composite (collection of components). This interface is used by the client to access both of them. Consequently, the client can ignore the differences between the leaves and the nodes. The interface that is shown to the client is called the component. If the request is to a leaf, the request is forwarded to the leaf to handle it, but if the request is to a composite, then the request is propagated through the tree of the composition until it reaches its final destiny: to target leaves. This concept sounds very nice and simple, but in implementation, questions arise such as how do we deal with the differences between the leaves and the nodes? The answers to this question and others are discussed in the implementation phase. Some decisions have to be made to keep the uniformity of the interface for both the part and the whole.

6.1.3 Structure and Actors

In [GHJV95], the class diagram in Figure 10 shows the design structure. The actors of this design are:

- Client: the user of the system that is represented by the Composite Design Pattern.
- Component: represents the interface that is used by the Client to access the Leaves and the Composites. It implements the functionalities that are common for both.
- Leaf: is the primitive component of the composition. It represents the part in the part-whole structure.
- Composite: is the container (or collection) of primitives. It represents the whole in the part-whole structure.

6.1.4 Implementation Decisions

As mentioned above, to implement the Composite Pattern in a way that meets both the part-whole structure and the uniformity of the interface at the same time, implementation decisions have to be made.

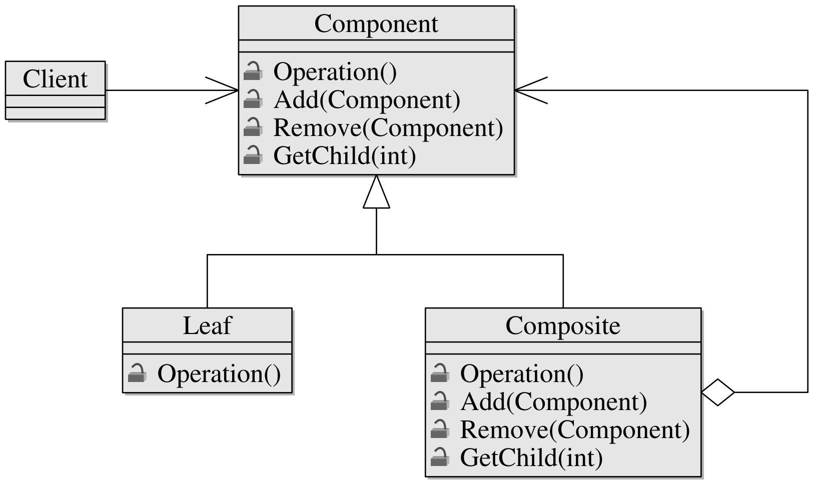

Figure 10: Composite Design Pattern in Object-Oriented Programming

The varying functionalities that distinguish the leaves from the composites need to be implemented either in the highest level of the design structure, which in this case is the interface (the component), or it should be left to the lower level of the design structure, which are the leaves and the composite classes.

Having the varying functionalities in the top level of the design makes the interface for both leaves and composites uniform to the client. However, this may produce confusion or inconsistency.

- The functionality of adding and removing children (Add(Component), Remove(Component)) depends on whether the node is a leaf or a composite object. a Leaf does not have any children, so having these two methods in the interface is firstly confusing to the client, and secondly may cause an unexpected failure during runtime if not handled properly.

  Another choice could be to remove the varying functionality of adding and removing children to the lower level of the structure, which means moving the non-common functionalities from the interface to the leaf and the composite classes. This implementation breaks the uniformity (that is also called transparency) of the interface and since the priority in the design was given to uniformity in [GHJV95], the latter choice is discarded.

- Other design decisions depend on the Programming Implementation of the object-oriented paradigm. I list hereby the most important points:

- For bottom-up traversal, a reference to the parent object must be stored. The possibilities are either to have references stored in the Component level of the design structure or to have them instead stored in the Composite class, since that is the actual parent of the collection of sub-components. However, having references to parents placed in the parent level (which is the Component) limits the child from being shared. Another design could store the reference to a list of parents at the child level, but this approach may be misleading.
- Another issue is whether it is important to have a list of children components (sub-components) in the parent or root component.
- Which level of the design structure is responsible for deleting or adding sub-components: The Component level or the Leaf/Composite level?
- Whether there is a real need for sorting lists of components (sub-components).
- And finally whether there is a suitable data structure for storing components.

6.1.4.1 Java Implementation

As an example, Java Implements the Composite Design Pattern. As any other object-oriented programming language, Java makes use of the power of its language structure to achieve optimal design decisions in implementing this Pattern. The following is a list of the main decisions taken by the Java design constructors. [Coo00]

- Following the Composite Design Pattern preference of uniformity (transparency), Java locates "add" and "remove" children in the Component level. It overcomes the exception of leaves not having children by checking first for the existence of a sub-component using the method "hasNext()" or "hasMoreElements()". Both methods are supported in Java Data Structures; they prevent runtime null pointer exceptions.
- To achieve bottom-up traversal, Java keeps a reference to the parent component in the constructor of the sub-component.
- To avoid the expensive functionality of sorting and re-sorting, Java provides a functionality to cache expensive queries or calculations in the parent component.

6.2 Composite Design Pattern in Erasmus

Although Erasmus is a completely different paradigm from object-oriented, I tried to keep the essential concept of Composition in the Composite Design Pattern in Erasmus. Keeping the definition of part-whole is the main concept I tried to achieve in this design Pattern. As a result I have found that in Erasmus the Composite Design Pattern can be demonstrated in three levels: Cell, Process, and Protocol.

6.2.1 Cell Level Composition

A Cell in Erasmus works like a Component in the Composite Pattern in object-oriented paradigm. It encapsulates in it a composition of different composite or primitive cells. The cell is called primitive if it is not composed of another cell, while a composite cell is a cell that is composed of one or more cells.

Figure 11 demonstrates the Cell Composition in Erasmus in an example. The example consists of a client that reads data from a console. The information includes the full name, date of birth, and nationality. The information related to the name part is processed in a cell ("nameCell"), so is the information about the date of birth and the nationality ("ageCell" and "nationalityCell" respectively). A composite cell, on the other hand, encapsulates all these functionalities in one cell by calling these individual cells. The composite cell also supports the option of using some functionalities rather than all. To achieve this option, a Factory Pattern is used in implementing the composite process (that is instantiated in the composite cell). From this design a client is left with the possibility of using the composite cell as a "whole" and/or the individual cell as a "part" in the part-whole design. To demonstrate both possibilities in the example, the client process "clientProc" sends messages to the composite cell "compositeCell", which propagates eventually the messages to the "nameCell" cell, and instantiates the individual cell "nameCell" dynamically then passes messages to it. However, the part and the whole in this design do not have a uniform interface. The reason why the design cannot support this feature is explained in Section 6.2.1.2

6.2.1.1 Structure and Actors

The Cell Composite Design Pattern consists of Protocols, Processes, and Cells as part of the design of the application.

- Protocols:
    - nameProt: contains information about a name. The protocol has a variable "fName" that stores the first part of the name, and a "lName" variable that stores the family name. Both variables are of type Text.

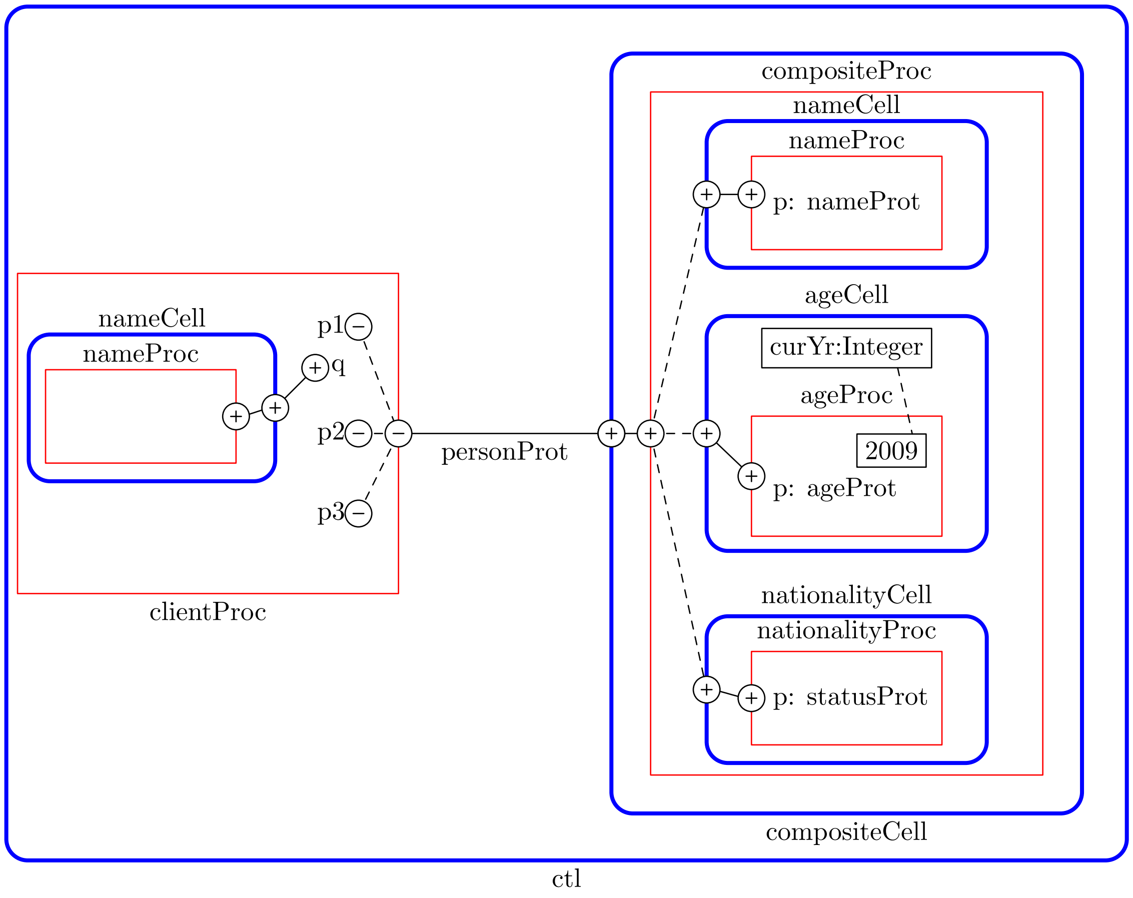

Figure 11: Cell Composition

  - ageProt: the protocol contains three Integer variables that represent the day of birth “dayOfBirth”, month of birth “monthOfBirth”, and year of birth “yearOfBirth”.
  - statusProt: contains information about the nationality, the value is stored in a variable of type Text, “nationality”.
  - personProt: this is a communication channel that apart from passing ports, it also contains signals “name”, “age”, “status”, and “stop”. This protocol includes port declarations “aName” of type “nameProt”, “anAge” of type “ageProt”, and “aStatus” of type “statusProt”.

- Processes:
  - nameProc: server to handle the name part of personProt data. It has a port of type “nameProt”.
  - ageProc: server to handle the age part of personProt data. It is passed a static integer, representing the current year. It has a port of protocol type “ageProt”.
  - nationalityProc: server to handle the nationality part of personProt data. It has a port of protocol type “statusProt”.
  - clientProc: is a client that passes information of personProt from the user of the program to be handled by the cell compositeCell.
  - compositeProc: behaves as a Factory that instantiates primitive cells that it is composed of.

- Cells
  - nameCell: instantiates the “nameProc” process. It encapsulates, in a cell, the functionality of storing and manipulating the name of a person.
  - ageCell: instantiates the “ageProc” process. It encapsulates, in a cell, the functionality of storing and manipulating the date of birth information of a person.
  - nationalityCell: instantiates the “nationalityProc” process. It encapsulates, in a cell, the functionality of storing and manipulating the nationality of a person.
  - compositeCell: is composed of the “compositeProc” process. It instantiates the “compositeProc” process, and passes through the messages from the clientProc.
  - ctl: instantiates the process “clientProc” and the cell “compositeCell” and connects them together through the port “p” of protocol type “personProt”.

6.2.1.2 Consequences

The Cell Composite Design allows a part-whole structure. The part part is represented in this example by the cells “nameCell”, “ageCell”, and “nationalityCell”. These cells do not contain other cells, so I will describe them as primitive (or individual) cells. The whole part, on the other hand, is represented by the “compositeCell” cell that is indirectly composed of the primitive cells. I used the expression “indirectly” here to point out that the “compositeProc” process is the real container of the primitive cells.

As shown in Figure 11, one notices that the “compositeCell” instantiates the “compositeProc” process. This process may seem redundant at first, but it is not in reality. The reason for having the process is explained in detail in Section 6.2.1.3.

In the design, the Client, “clientProc”, sends requests to the part that is the “nameCell” cell and to the whole that is the “compositeCell” cell and that is in turn composed of the “nameCell” cell among other cells. The protocols of the part and the whole are not the same, rather, the protocol of the “compositeCell” is a composition of the protocol of the “nameCell” cell and other primitive cells.

This above composition does not follow strictly the definition of the Composite Pattern in Section 6.1.1 in the sense that it does not provide a uniform interface to the primitive (or individual) and composite cells. A design that fulfills that requirement is represented in the following piece of code:

```
compositeProc = process p: +personProt|
  loopselect
    || p.name; nameCell(p.aName);
    || p.age;  ageCell(p.anAge);
    || p.status;  nationalityCell (p.aStatus);
    || p.comp; componentCell(p);
    || p.stop;  exit;
  end
end

componentCell = cell p: +personProt|
  compositeProc(p)
end
```

Note that in this design we have a cell (“componentCell”) that is composed of a process (“compositeProc”). This process is designed as a Factory Pattern; it instantiates primitive cells such as “nameCell”, “ageCell” etc, and a composite cell that is “componentCell”. This design fulfills the uniformity of the primitive and composite structure on one hand, and allows a composition of cells to be represented as a tree structure on the other hand. However, this design produces a compilation error because it contradicts the principle of declaration precedence in Erasmus (explained in Section 3.7.6).

The Cell Composite Pattern encapsulates the part and the whole in cells. It makes adding new functionalities to the system easier by having to instantiate the new cell in the "compositeProc" process, and by adding the protocol of the new cell to the protocol of the composite cell.

6.2.1.3 Implementation

In Cell Composite Design there are some points to be highlighted:

- To prevent compilation error for using a fork-shaped channels (explained in Section 3.7.5), the protocol of the composite cell must contain the ports of the primitive cells it is composed of. This type of protocol design is discussed in Section 6.2.2.

  In the example, the protocol type of the composite cell is "personProt"; it is defined in the following piece of code:

  ```
  personProt = protocol
    *(name; aName: nameProt| age; anAge: ageProt| status; aStatus: statusProt);
    stop
  end
  ```

  This protocol is in fact a composite protocol that is made up of ports of protocol type "nameProt", "ageProt" and "statusProt" (as well as signals). The latter protocols are the ports' protocol types of the primitive cells "nameCell", "ageCell", and "nationalityCell" respectively.

- Since a client sends a request, the composite cell as well as the primitive cells must be servers to handle the request.

- Since a cell can only declare processes and links them with other processes through declaring ports, processes are created in each cell to carry out the functionality. However, in general, a primitive cell in the design can contain many processes, this does not break the annotation of primitive from primitive cell.

- If the design of Cell Composition is changed in a way such that the composite process looks like the following piece of code (by removing the Factory Pattern from the process):

  ```
  compositeProc = process p: +personProt|
    nameCell(p.aName);
    ageCell(p.anAge);
    nationalityCell (p.aStatus)
  end

  compositeCell = cell p: +personProt|
  ```

```
    compositeProc(p)
  end
```

This design could trigger the idea of simplifying it to the following:

```
  compositeCell = cell p: +personProt|
    nameCell(p.aName);
    ageCell(p.anAge);
    nationalityCell (p.aStatus)
  end
```

This code gives compilation errors, because the operations that a cell can perform are limited to creating channels, processes, and cells, and then linking them together. The “compositeCell” cell specifies communications, which a cell cannot support.

- For the same last restriction of cells mentioned above, a Factory Pattern can only be implemented in a process and not in a cell.

6.2.1.4 Code

The signals that are in the “personProt” protocol namely: “name”, “age”, “status” are used in specifying which cell to instantiate. The following code suggests that if the “name” signal is sent, then it should be followed by sending a port of protocol type “nameProt”. In the same way sending the signal “age” precedes sending “anAge” port, etc.

```
personProt = protocol
  *(name; aName: nameProt| age; anAge: ageProt| status; aStatus: statusProt);
  stop
end
```

The following part of code demonstrates a primitive cell and its corresponding process:

```
nameProc = process p: +nameProt |
  temp: Text;
  temp := "First Name: " + p.fName;
  temp += "\t";
  temp += " Last Name: " + p.lName + "\n";
  sys.out := temp;
end

nameCell = cell p: +nameProt |
  nameProc(p);
end
```

As per Figure 11, the design demonstrates how a client can call a primitive cell alone and/or call it as part of the composite cell. This is demonstrated in instantiating dynamically the “nameCell” cell in both cases.

```
clientProc = process p: −personProt|
 ...
   q: −nameProt;
   nameCell(q);
   q.fName := fName;
   q.lName := lName;
 ...
end
```

and

```
compositeProc = process p: +personProt|
   loopselect
      || p.name; nameCell(p.aName);
      || p.age; ageCell(p.anAge);
      || p.status; nationalityCell (p.aStatus);
      || p.stop; exit;
   end
end
```

The main cell "ctl" declares the client process "clientProc" and the composite cell "compositeCell" and links them together through the port "p" of protocol type "personProt".

```
ctl = cell
   p: personProt;
   clientProc(p);
   compositeCell(p);
end

ctl()
```

6.2.2 Protocol Level Composition

The Protocol Composite Design demonstrates how Erasmus supports composition within protocols as part of the language. A Protocol can consist of a component that is a composition of other protocols. Figure 12 demonstrates the Protocol Composition, which also achieves the part-whole principle of the Composite Design Pattern.

6.2.2.1 Structure and Actors

- Protocols
  - lineProt: a protocol that consists of a primitive data type. This is considered a leaf protocol.
  - triangleProt: is a composite protocol that consists of three protocols of type "lineProt". This represents the composition of protocols.
- Processes

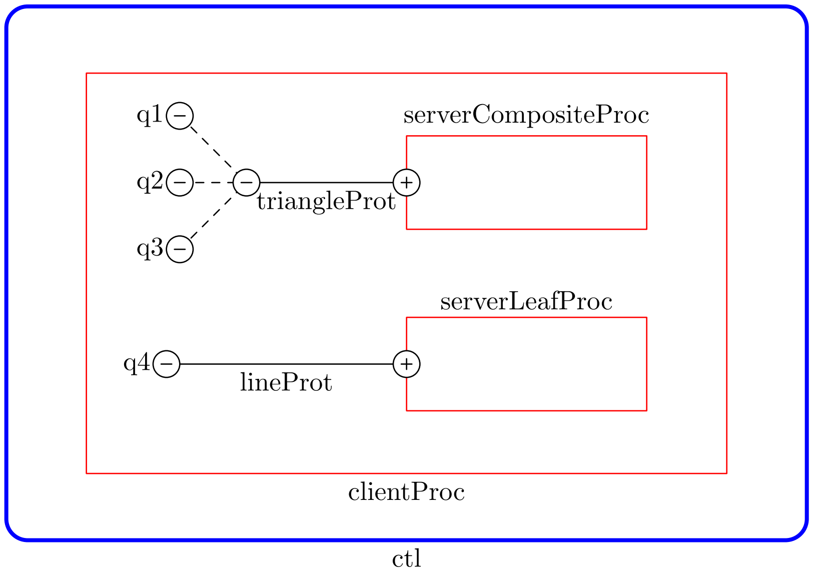

Figure 12: Protocol Composition

- serverCompositeProc: simple process that behaves as a server to the composite protocol "triangleProt". This process displays the values of the composite protocol by breaking it down into leaf protocols "lineProt".
- serverLeafProc: another simple process that behaves as a server to the leaf protocol "linePoint". It displays primitive data of the leaf protocol.
- clientProc: the process that creates both instances "ServerCompositeProc" and "serverLeafProc", and issues a request to both processes.

• Cells

- ctl: this cell only runs the process "serverLeafProc".

6.2.2.2 Consequences

In the demonstrated design pattern the composite protocol consists of several protocols of the same type. However, the design does not have any restriction on having a composition of different types of protocols. Again in this design pattern, the downside is that it does not allow recursive traversal for the principle of declaration precedence that also restricts recursive traversal in the "Cell Level Composition" (Section 6.2.1.2).

6.2.2.3 Implementation

The composite protocol is decomposed into its primitive protocols inside the processes or cells.

6.2.2.4 Code

```
lineProt = protocol
   point: Integer
end;
```

The “triangleProt” protocol represents the composite design pattern part. Each of its elements is composed of a primitive (leaf) protocol, in this case of type “lineProt” protocol.

```
triangleProt = protocol
   line1: lineProt;
   line2: lineProt;
   line3: lineProt;
   stop
end;
```

In the process “ServerCompositeProc”, the composite protocol “triangleProt” delegates to its leaf protocols: “line1”, “line2”, and “line3”.

```
ServerCompositeProc = process p: +triangleProt |
   sys.out := "Composite Protocol\n";
   loopselect
      || line1: +lineProt := p.line1;
        sys.out := "Line1: point: " + text line1.point + "\n";
      || line2: +lineProt := p.line2;
        sys.out := "Line2: point: " + text line2.point + "\n";
      || line3: +lineProt := p.line3;
        sys.out := "Line3: point: " + text line3.point + "\n";
      || p.stop; exit;
   end;
end;
...
```

In the “clientProc” process, “ServerCompositeProc” is passed a message that is in reality of a port type. This port is composed of three elements: “q1: -lineProt”, “q2: -lineProt”, and “q3: -lineProt”. The following part of the code demonstrates how these ports are connected to the port “p: -triangleProt”, and how data is passed through.

```
clientProc = process |
   p: −triangleProt;
   ServerCompositeProc(p);
   q1: −lineProt;
   p.line1 := q1;
   q1.point := 1;
```

```
        q2: −lineProt;
        p.line2 := q2;
        q2.point := 2;

        q3: −lineProt;
        p.line2 := q3;
        q3.point := 3;

        q4: −lineProt;
        serverLeafProc(q4);
        q4.point := 4;

        p.stop;

    ....
```

6.2.3 Process Level Composition

As in the Cell Composite Pattern, Erasmus provides the facility of composing a process from other processes. The Process Composite Pattern again fulfills the part-whole general idea of the Composite Design Pattern.

To demonstrate the Process Design Pattern there are three versions of the same program. The first version starts with a simplified demonstration of a Composite Process that consists of only one process. The second version of the program demonstrates how to have a process that is composed of two other processes. Finally the third version of the program consists of a chain of composite processes. This last version is not far in principle from the Chain Of Responsibility Design Pattern in Erasmus as described in Chapter 7.

6.2.3.1 Structure and Actors

Process Composite Pattern Version One

The design is shown in Figure 13

- Protocols
  - personProt: carries the personal information (data) of a person.
  - commProt: is a protocol that represents a channel that passes a protocol of type “personProt”.
- Processes
  - serverProc: handles the data of the protocol type “personProt”. It just displays the data on the console.

  - componentProc: extracts the "personProt" from "commProt" protocol and propagates the former protocol to "serverProc".
  - clientProc: creates data and sends a request to the "componentProc" process to handle.

- Cells
  - ctl: executes the "clientProc" requests and runs the application.

Process Composite Pattern Version Two

The design is shown in Figure 14

- Protocols: same as in version one
- Processes
  - server1Proc: accepts messages of protocol type "personProt", and it handles only a set of data in this protocol.
  - server2Proc: this process also accepts messages of protocol type "personProt", and it handles another set of data in this protocol (other than the set of data in "server1Proc"). Both sets are disjoint.
  - componentProc: accepts messages of protocol type "commProt", and it propagates messages of type "personProt" to both processes "server1Proc" and "server2Proc", which demonstrates the composition pattern of processes.
  - clientProc: creates a request to be handled by the servers.
- Cells
  - ctl: runs the application/program.

Process Composite Pattern Version Three

This version has the same protocols, processes, and cells, as in version two, however, it is different in the processes distribution. This version follows a sequential composition that looks like a chain (as shown in Figure 15). In version two, the process "componentProc" is composed of the two other processes "server1Proc", and "server2Proc", but in version three, the process "componentProc" is composed of the process "server1Proc", which in turn is composed of "server2Proc".

6.2.3.2 Consequences

As in the two previous composite design patterns (Sections 6.2.1 and 6.2.2), the process design pattern fulfills the part-whole design structure, in the sense that a client process or cell can call a process that is either a composition of other processes or a primitive process. A disadvantage of the design in version two, having it working with a prototype compiler of Erasmus, is the need to send two stop signals from the client to the servers to ensure that both server processes terminate running and then the runtime system could resume garbage collection. However, the third version overcomes this particular point by re-sending the message from "server1Proc" to "server2Proc", although this version makes the plugging of new processes to "server1Proc" more difficult. The Second version of the Process Composite Design Pattern allows more flexibility in plugging or adding new processes to the process "componentProc".

6.2.3.3 Implementation

The following Figures 13, 14, and 15 demonstrate how the three versions of the Process Composite Design Patterns are implemented. In the third version the composition takes a chain of nested processes. As mentioned above, in the second version the number of "stop" signals (to be sent from the client process to "componentProc") has to match the number of processes that compose "componentProc", to ensure terminating all the server processes successfully. Fortunately the stopping mechanism is not part of the language (as explained in Section 3.7.7).

6.2.3.4 Code

- Process Composite Pattern Version One

```
personProt =
protocol
    (fName: Text|
    lName: Text|
    dayOfBirth: Integer|
    monthOfBirth: Integer|
    yearOfBirth: Integer|
    nationality: Text);
    stop
end

commProt = protocol
    channel: personProt
end

serverProc = process p: +personProt |
```

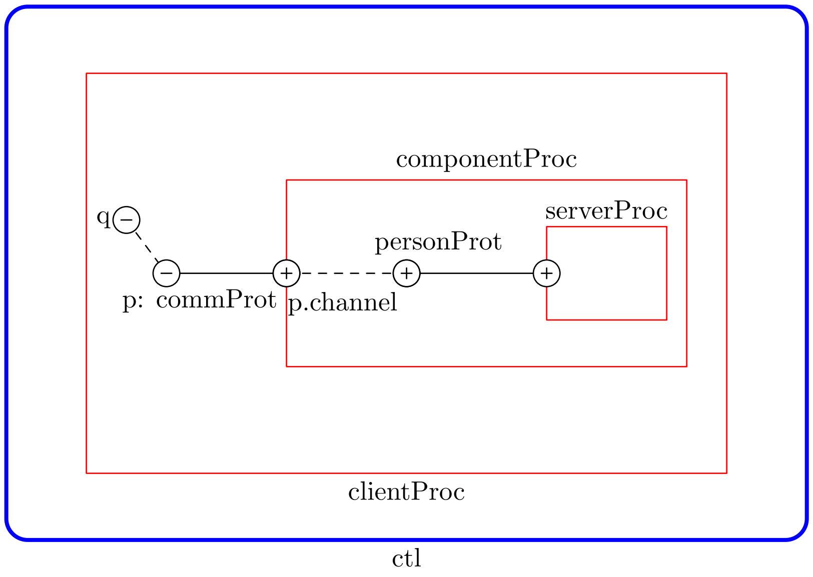

Figure 13: Process Composition Version One

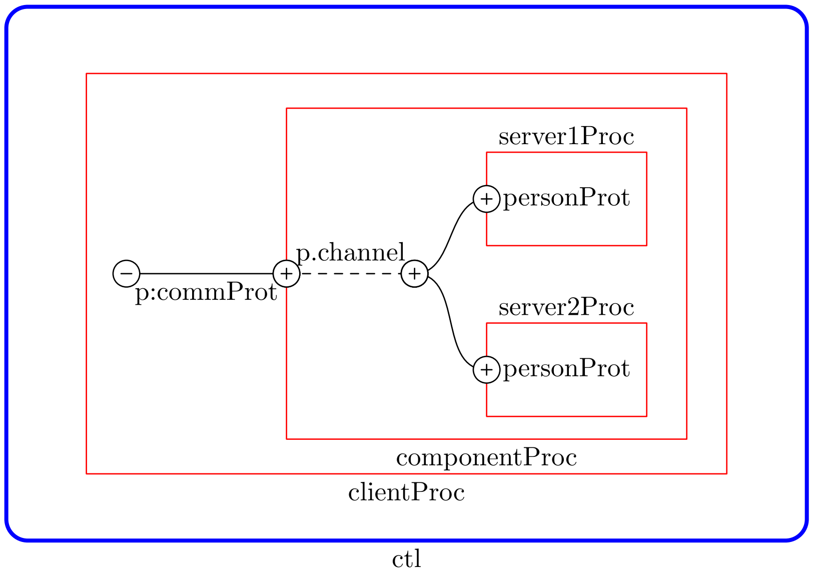

Figure 14: Process Composition Version Two

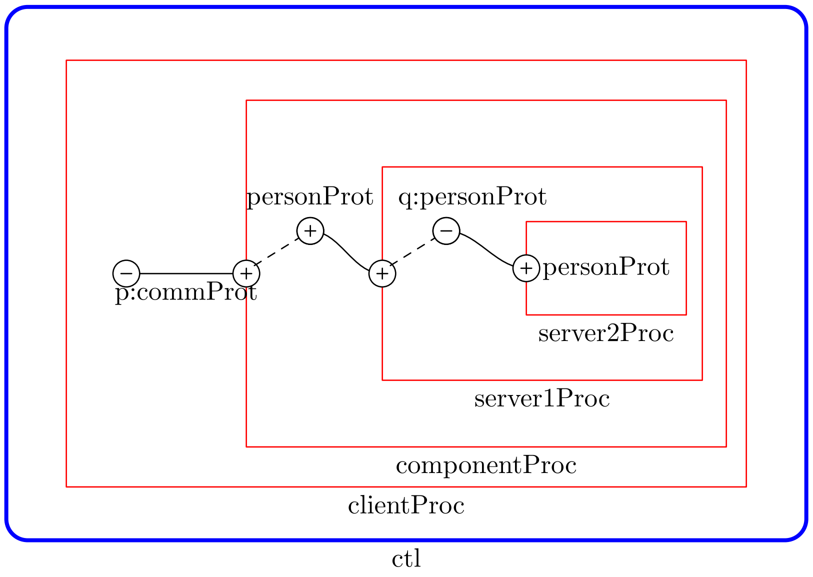

Figure 15: Process Composition Version Three

```
    loopselect
        || $sys.out$ := "Your First Name is: " + $p.fName$ + "\n";
        || $sys.out$ := "Your Last Name is: " + $p.lName$ + "\n";
        || $sys.out$ := "Your Day of Birth: " + text $p.dayOfBirth$ + "\n";
        || $sys.out$ := "Your Month of Birth: " + text $p.monthOfBirth$ + "\n";
        || $sys.out$ := "Your Year of Birth: " + text $p.yearOfBirth$ + "\n";
        || $sys.out$ := "Your Nationality: " + $p.nationality$ + "\n";
        || $p.stop$; exit;
    end
end

$componentProc$ = process $p$: $+commProt$ |
    $q$: $+personProt$ := $p.channel$;
    $serverProc(q)$;
end

$clientProc$ = process |

...

    $p$: $-commProt$;
    $componentProc(p)$;
    $q$: $-personProt$;
    $p.channel$ := $q$;

    $q.fName$ := $fName$;
    $q.lName$ := $lName$;
    $q.dayOfBirth$ := $dBirth$;
    $q.monthOfBirth$ := $mBirth$;
    q.yearOfBirth := yBirth;
    q.nationality := nationality;
    q.stop;
...
```

- Process Composite Pattern Version Two

```
...

server1Proc = process p: +personProt |
    loopselect
        || sys.out := "Your First Name is: " + p.fName + "\n";
        || sys.out := "Your Last Name is: " + p.lName + "\n";
        || sys.out := "Your Nationality: " + p.nationality + "\n";
        || p.stop; exit;
    end
end

server2Proc = process p: +personProt |
    loopselect
        || sys.out := "Your Day of Birth: " + text p.dayOfBirth + "\n";
        || sys.out := "Your Month of Birth: " + text p.monthOfBirth + "\n";
        || sys.out := "Your Year of Birth: " + text p.yearOfBirth + "\n";
        || p.stop; exit;
    end
end

componentProc = process p: +commProt |
    q: +personProt := p.channel;
    server1Proc(q);
    server2Proc(q);
end

clientProc = process |

...

    p: −commProt;
    componentProc(p);
    q: −personProt;
    p.channel := q;

...
    q.stop;
    q.stop;
...
```

- Process Composite Pattern Version Three

```
...
```

```
server2Proc = process p: +personProt |
    loopselect
        || sys.out := "Your Day of Birth: " + text p.dayOfBirth + "\n";
        || sys.out := "Your Month of Birth: " + text p.monthOfBirth + "\n";
        || sys.out := "Your Year of Birth: " + text p.yearOfBirth + "\n";
        || p.stop; exit;
    end
end

server1Proc = process p: +personProt |

...
```

In the following, a dynamic port is created in "server1Proc", passes information from p to the server "server2Proc".

```
...
    q: −personProt;
    server2Proc(q);
    loopselect
        || sys.out := "Your First Name is: " + p.fName + "\n";
        || sys.out := "Your Last Name is: " + p.lName + "\n";
        || sys.out := "Your Nationality: " + p.nationality + "\n";
        || q.dayOfBirth := p.dayOfBirth;
        || q.monthOfBirth := p.monthOfBirth;
        || q.yearOfBirth := p.yearOfBirth;
        || p.stop; q.stop; exit;
    end
end
...
```

In "componentProc", only "server1Proc" is created and propagated into the dynamic message "q" of protocol type "personProt".

```
...
componentProc = process p: +commProt |
    sys.out := "componentProc is Running\n";
    q: +personProt := p.channel;
    server1Proc(q);
end

clientProc = process |


...
    p: −commProt;
    componentProc(p);
    q: −personProt;
    p.channel := q;

...
```

6.3 Summary

In this chapter we have shown how to implement the Composite Pattern in Erasmus. The composition was demonstrated in three levels: cell, protocol, and process. The Erasmus implementation differs from the OOP implementation in that it does not support recursive traversal through the components in any level of composition. The implementation in the case of the cell level composition demonstrates that µE's feature, "declaration precedence", makes it impossible to support recursive traversal. The disadvantage of the Erasmus version is clearly the inability to support recursive traversal through its components in any level of composition being a protocol, process, or even a cell.

Chapter 7

Behavioral Pattern: Chain Of Responsibility Design Pattern

The Chain of Responsibility Design Pattern among others (e.g. Command, Interpreter, and Iterator Design Patterns that are discussed in Chapters 8, 9, and 10 respectively) is a Behavioral pattern. Behavioral patterns are classified by the book [GHJV95] in the Behavioral category. This category is concerned with the functionality of the different components that construct the pattern, the way responsibilities are distribution, and the communication between the components.

7.1 Chain of Responsibility Design Pattern in OOP

The Chain of Responsibility Pattern is useful when a request may fall in a set of one or none handlers among many options to handle requests of that type.

The discussion of this section is based on the following references: [GHJV95], [Coo00], and [Met02]

7.1.1 Definition

"Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it"[GHJV95].

7.1.2 Background

In this Pattern a client's request can be potentially handled by many objects, and the actual handling object is only defined during runtime. A possible solution would be to pass the request through a set of "if" statements to determine the appropriate handler during runtime, but this approach increases coupling between handling objects. A more refined solution to this problem, which reduces decoupling, would be to introduce a chain of handling objects in which the request gets propagated until it reaches either its appropriate handler, or it is not handled at all.

A simple example in which the Chain of Responsibility Pattern could be used, would be in having a form in which a full name is input. The full name may appear in different formats: "first-name, last-name", "last-name first-name", "first-name middle-name last-name", etc. Depending on the input, a parser should detect the format of the name to get the first name, middle name if any, and the last name separately. Objects that represent each format are created and linked one to another in a chain. The input full name is passed to the first link of the chain, if handled in this link the control is returned to the client, if not the full name is passed through to the following link, until it reaches the last link. The last link could handle the input string in a general way (by taking the whole name as first name for example). However, we could still not handle the input string, if we enforce a set of allowed formats, in this case if the input string does not match any of the allowed formats, the request is not handled.

7.1.3 Structure and Actors

The class diagram of the Pattern is as demonstrated in Figure 16 and the main actors of this Pattern are:

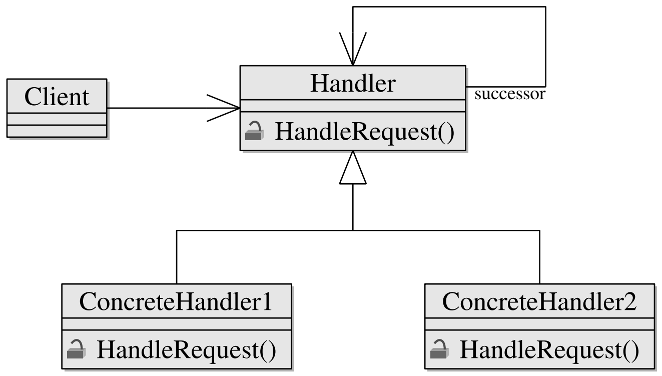

Figure 16: Chain Of Responsibility Design Pattern in Object-Oriented Programming

- Client: is the object that generates the request.

- Handler: is the interface that provides functionality to handle a request, and to connect to another handler to produce eventually a chain.
- ConcreteHandler: extends the “Handler’s” functionalities, and is a concrete object that represents a link in the chain. This object receives the request, either handles it or forwards it to the following “ConcreteHandler” object, if any.

7.1.4 Implementation Decisions

Chain of Responsibility is used when the sender of a request may have many possible handlers, and the sender knows only implicitly who the handler is. The set of processing objects should be determined dynamically.

To get the best out of this Pattern, the links of the chain should be arranged by starting with the most specific condition or handler and continue till the most general one. If the chain is arranged the other way around, we risk handling requests with the most general handler and not reaching the exact suitable handler because the latter falls after the general handler in the chain.

The Chain of Responsibility does not guarantee that the request gets any reply by the end of the last object in the chain. As a consequence, the request may not be handled at all. However, if the application requires a reply, then a general handler or an error handler is placed at the end of the chain.

The Chain of Responsibility Pattern is good in distributing functionalities among handling objects, in the sense that each object is restricted to its own functionality to handle the request so each object knows what it should do and does not mind about what other handlers are doing or should do. Decoupling functionalities makes it easier if changes are needed [Coo00].

This Pattern participates in decoupling in two levels, decoupling the handler objects from each other, and the request sender from its handlers.

7.1.4.1 Java Implementation

Java makes using Chain of Responsibility Pattern easy by having the “ConcreteHandler” objects extend the “Handler” interface. In this way the concrete objects can inherit other features from other parent objects, and implement at the same time the functionalities of the “Handler” interface.

Java also makes use of this Pattern in its language structure. Inherited methods, for example, function exactly the way Chain of Responsibility does, in the sense that the compiler attempts to find the method implementation in the last child deep in the hierarchy;

if not found, it goes up the tree of hierarchy, until it reaches the highest parent in the hierarchical tree in which the method is defined. Also, Exception handling in Java works the same way, the exception bubbles up, until it either finds a handler or it raises a runtime exception which causes the termination of the program.

In [Coo00], it is shown that Chain of Responsibility Design is not only limited to a linear chain, rather the design could take a tree-like shape. In the tree design, links or nodes of the chain point up to the following higher level in the hierarchy. However, the tree design makes building a chain a more complex job, because linear structure is simpler in general than tree structure; many chains may point to the same parent chain in the hierarchical structure. For that reason, the tree itself could be simplified by flattening it to a linear chain instead. The suggested structure is to start the chain from the lowest level in the tree starting by the most left side node moving to the right, then up to the following level again starting by the most left node, and so on.

The "ConcreteHandler" objects do not have to be completely independent, they could be correlated instead. One type that is mentioned in [Met02] is a Composite Pattern. The handling objects follow in their structure the Composite Design, but they still implement the Chain of Responsibility Pattern in being chained together.

7.2 Chain of Responsibility Design Pattern in Erasmus

The basic idea of Chain of Responsibility Pattern in Erasmus is to propagate the request through processes using ports. In this design the processes (starting from the first in the chain until the last but one) work as clients and servers at the same time in respect to the protocol that carries the request (the parameters). In this way, each process in the chain, receives the request (the first process in the chain receives the request from the client, and all of the others receive it from the ancestor in the chain) so it behaves as a server, and then it propagates the message to the successor process (if any), if it cannot handle the request; in this way it behaves as a client. However, the last process in the chain does not have a client parameter because it does not have any successor process.

Forming the chain is achieved in a cell by connecting the ports of the processes one to the other as shown in Figure 17.

As it is the case in the object-oriented pattern, this Pattern in Erasmus does not restrict "programmatically" arranging the links in a chain from the most special to the most general. However, this is a convention the user of the design should follow.

7.2.1 Structure and Actors

Figure 17 demonstrates the Design Pattern. The actors that build up this pattern are:

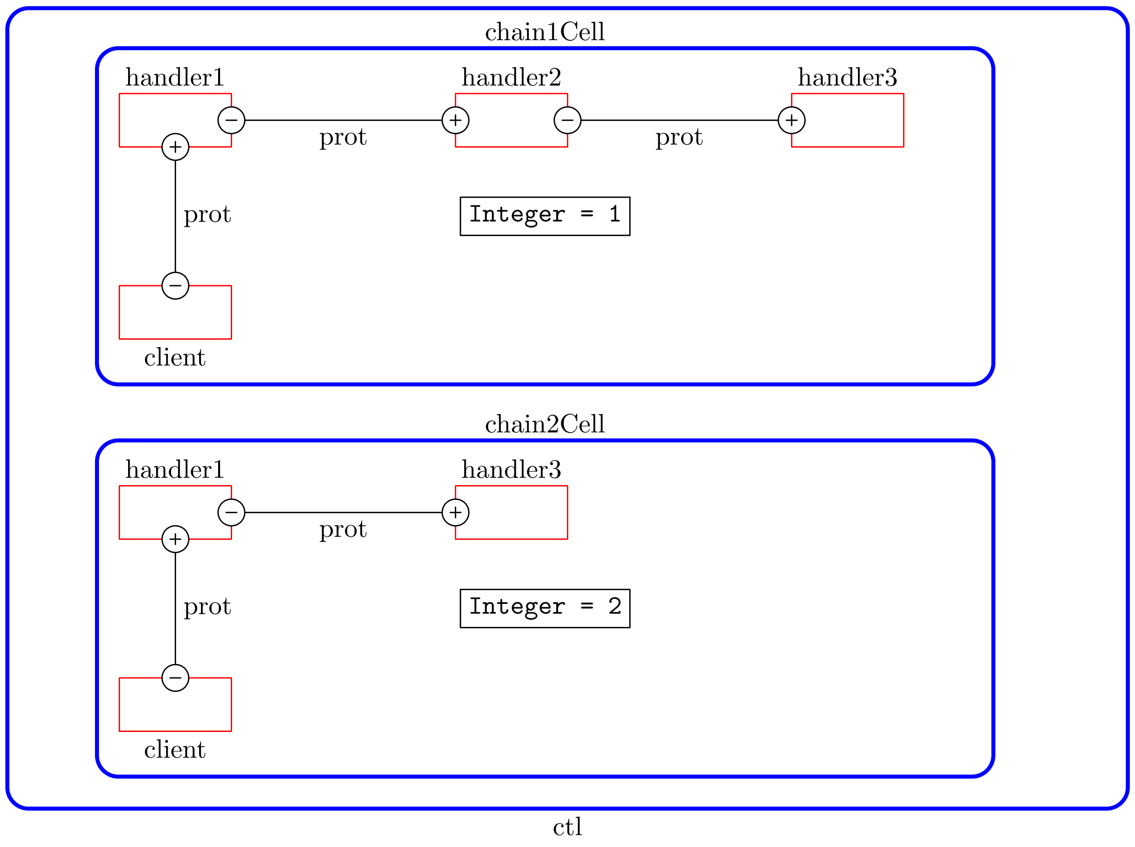

Figure 17: Chain Of Responsibility Design Pattern in Erasmus

- Protocols
  - prot: carries the data that is to be processed. “i” is a variable of type Integer, its value is tested in “handler1” and “handler2” processes to decide whether to handle the request in the process or to propagate it to the following in the chain. The other variable is “txt” of type Text, it represents the data that is to be processed. The last component of the protocol is the signal “stop” that announces the end of the sent messages. Both variables “i” and “txt” could be repeated many times before sending the end signal “stop”.
- Processes

  - client: is the process that creates requests.
  - handler1: is the first link/node in both chains which are defined in “chain1Cell”, and “chain2Cell”.
  - handler2: is the process that represents the second link/node in the “chain1Cell” chain. In a general term “handler1” and “handler2” can be repeated as many times as needed in the program.
  - handler3: is the last handler of the client request in both chains “chain1Cell” and “chain2Cell”. Note that this process has only one port, because the process does not have a successor process.

- Cells
  - chain1Cell: instantiates the processes “client”, “handler1”, “handler2”, and “handler3”, and it connects their ports through variable ports “p1”, “p2”, and “p3” of type “prot”. It also passes an Integer parameter to the processes. This cell defines the chain of processes; this chain consists of three handlers.
  - chain2Cell: defines and instantiates the requesting process and the handlers as in “chain1Cell”, but with another chain configuration. The defined chain in this cell is composed of only two processes.
  - ctl: it runs the application. It instantiates both cells “chain1Cell” and “chain2Cell”.

7.2.2 Consequences

As in [Met02], Erasmus Pattern supports the following:

- The Pattern, in addition to the general structure of Erasmus, decreases coupling between processes. The only connection between processes in a chain is done by passing messages through the channels that connect ports (which belong to the processes) to each other.

- Each process encapsulates in itself the functionality it provides without knowing anything about what other processes do. This facilitates removing links (processes) from the chain, or re-arranging the chain to another shape or form.

- It is left up to the user of the pattern to decide whether to handle each request by having the last process as the general handler, or to not handle the request at all, if it does not fall within the conditions of the handling processes. However, when a message is required to be sent back from the handler to the requester (client), leaving it open to the handler to decide not to send back a response causes a deadlock as explained in Section 7.2.4.

7.2.3 Implementation

As mentioned above, the structure of the Pattern is as follows:

- Processes' ports: are responsible for propagating messages or the request from the client along the links or nodes of the chain. The protocol, in general does not have any restrictions in building it. The protocol represents only the message or request that is passed from the client to the chain of processes.
- Processes: produce the functionality, each process has a different handling functionality from the others. Each process knows nothing about what other processes handle or do, and when a process cannot handle the request, it passes the request through its client port to the successor process, if any. The last process can only have one server port (that is denoted in Figure 17 as a plus "+" sign), because there is no successor to this process.
- Cells: build the processes to shape a chain of processes by connecting the ports of these processes as desired.

7.2.4 Code

Processes of a chain that could be placed from the first to the one before the last in the chain have to have at least two ports one behaves as a server, which receives the request, and the other behaves as a client to propagate the request through it to the successor process.

```
...
handler1 = process i:Integer; p: +prot; q: −prot |
  loopselect
  || tempVal: Integer := p.i;
     str: Text := p.txt;
     if tempVal = 1 then
       sys.out := "Round" + text i + " Chain1 has processed " + str + "\n";
     else
       q.i := tempVal;
       q.txt := str;
     end;
  || p.stop; q.stop; exit
  end
end
```

The last node in the chain should have at least one server type port to receive the request.

```
...

handler3 = process i:Integer; p: +prot |

...
```

The client process has to have a client port in which it sends its request through to the chain of handlers.

```
...

client  = process i:Integer; p: −prot |

...
```

The main idea of passing the request through a test or condition, is to decide depending on the result whether to handle the request or to propagate it.

```
...

||  tempVal: Integer := p.i ;

...

    if  tempVal = 2
    then
```

If the condition is true, the request is handled, else it is passed to the out/client port to be propagated:

```
...

else
   q.i  := tempVal;
   q.txt  := str ;

...
```

The cells build chains, as in the following examples:

```
...

chain1Cell = cell
   p1,  p2,  p3:  prot;
    client (1,  p1);
   handler1(1,  p1,  p2);
   handler2(1,  p2,  p3);
   handler3(1,  p3);
end

chain2Cell = cell
   p1,  p2:  prot;
    client (2,  p1);
```

```
  handler1(2, p1, p2);
  handler3(2, p2);
end

 ...
```

The protocol of the above design is:

```
prot = protocol
  *(i: Integer; txt: Text);
  stop
end
```

Suppose that we change the protocol “prot” to add another message that is a reply from the server (handler) to the client:

```
prot = protocol
  *(i: Integer; txt: Text; ↑response: Text);
  stop
end
```

Also, suppose that we change the design to make it open not to handle a request. The changes of the program look like:

```
 ...

handler3 = process i:Integer; p: +prot |
  loopselect
    || tempVal: Integer := p.i; str: Text := p.txt;
       if tempVal = 3 then
         sys.out := "Round" + text i + " Chain3 has processed " + str + "\n";
         p.response := str;
       end
    || p.stop; exit
  end
end

 client = process i:Integer; p: −prot |
  sys.out := "\n";
  p.i := 4;
  p.txt := "Four";
  sys.out := "First Response: " // i // ": " // p.response // "\n";

  p.i := 1;
  p.txt := "One";
  sys.out := "Second Response: " // i // ": " // p.response // "\n";

  p.i := 2;
  p.txt := "Two";
  sys.out := "Third Response: " // i // ": " // p.response // "\n";

  p.stop
end
```

...

Note that the process "handler3" handles the request only if it meets the condition (by having the message "p.i" equals to three). So there is no guarantee of returning a response through the message "response" of type Text. The process "client" sends a request that does not match any condition of any available handler (having "p.i := 4"). This message causes the program to block and as a result to produce a deadlock, because the handler does not send back a response, while the client is waiting for it.

7.3 Summary

In this chapter we have shown how to implement the Chain of Responsibility Pattern in Erasmus. The Erasmus implementation differs from the OOP implementation in that decoupling is brought down to the minimum if compared with the OOP implementation. Also the specialization of the components (protocol/channel, process, and cell) of the language makes the code neater than the OOP. The channels in the design propagate through the message to be handled. Each process handles the request independently from the other processes. The cell encapsulates the arrangement of chains. The advantage of the Erasmus version when compared to the OOP version relies in the fact that the specialization in Erasmus of its components makes plugging or adding new nodes to the chain easier. However, the main disadvantage of the Erasmus version is the possible deadlock that can be caused if a response message is required from the handler(s) back to the client.

Chapter 8

Behavioral Pattern: Command Design Pattern

8.1 Command Design Pattern in OOP

The simplest definition of the Command Pattern is that it decouples the requester of an action from the actual performer of that action. It encapsulates an action or a command in an object. The discussion of this chapter is based on [GHJV95], [Coo00], and [FFBS04].

8.1.1 Definition

"Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations" [GHJV95].

8.1.2 Background

Freeman et al. provide a nice analogy to the Command Pattern [FFBS04]. The actors of this analogy that build up the Design Pattern are explained in detail in Section 8.1.3 and demonstrated in Figure 18.

The analogy runs like this: a Customer (Client) in a restaurant orders a dish and something to drink. The Waitress/Waiter (Invoker) takes the order disregarding the precise details (ingredients of the dish, and that of the drink). The Order represents the Command, in which the name of the dish and the drink are written down. Details about how the dish is cooked or implemented are not the main concern of the Order. The Order is passed to the Cook (who represents the Receiver). The Cook is the one who cooks the dish, who knows about the needed ingredients. He is the one who carries out the actual Command or Order. In this scenario we could see that the Cook does not know much about the Customer (they are both decoupled), and the Invoker does not know how the action of cooking is carried out (the Invoker is decoupled from the concrete act of cooking i.e. “ConcreteCommand”).

8.1.3 Structure and Actors

The players of the Command Pattern are as shown in Figure 18 and as explained below.

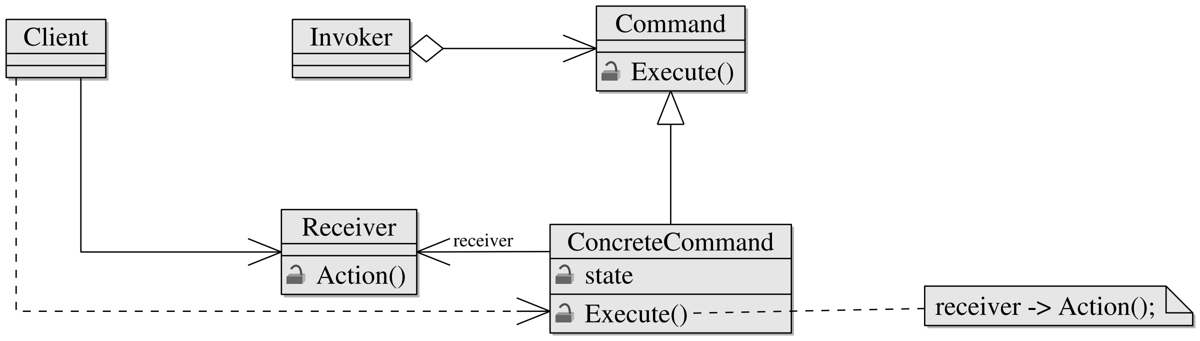

Figure 18: Command Design Pattern in Object-Oriented Programming

- Client: is the requester of an action to be carried out.
- Command: is an interface that has originally one method called “execute()”. In this method the action is encapsulated, and the functionality differs from one “ConcreteCommand” to the other. The “Command” interface is supposed to support also, as per the definition of the Pattern 8.1.1, an “undo()” method. The “undo()” can undo the last action taken, or a list of the last actions taken; this is left to the application that is developed.
- ConcreteCommand: represents the actual implementation of the “Command” interface. This class implements actually the action of the “execute()” method. The number of “ConcreteCommands” depends on the application and on the number of “Receivers”. The “ConcreteCommand” is composed of a “Receiver” object, and the “execute()” method. The “execute()” method is performed on the “Receiver” object.
- Invoker: invokes by instantiating the “ConcreteCommands”. The “Invoker” delegates the request from the “Client” to the “Command” so the action is carried out.
- Receiver: is the object, or objects that carry the actual action. The Command Pattern ensures having a uniform interface for carrying actions that are requested by the

"Client". The non-uniform functionalities or actions are in fact carried out by the "Receiver" inside the "execute()" method that belongs to the "ConcreteCommand".

8.1.4 Implementation Decisions

The way in which the Command Pattern is designed makes it easy to add new actions because the "ConcreteCommands" are independent from each other, and the decoupling between the "Invoker" and the "Receiver" is low.

In [GHJV95], it is mentioned that a command in some cases can have no "Receiver", either because it could carry out the functionality alone, or when the "Receiver" is implicitly defined or known.

As mentioned above, the Command Pattern supports undoing actions. The basic functionality of "undo()" method is to undo the last "execute()" action that was taken. An extended "undo" would consist on a list of undoing actions. This latter approach requires saving the executed actions in a stack that is used in undoing actions whenever needed.

In the same way a command is carried out, a sequence of commands is also possible. This sequence of actions is also called "MacroCommand".

8.1.4.1 Java Implementation

In Java, the Command Pattern is implemented by having the "Command" as an interface, and the "ConcreteCommand" as an implementation of it. This gives flexibility to the "ConcreteCommand" to extend another class.

The Command Pattern also supports undoing functionalities by tracking the last action taken or the list of latest taken actions, and then by undoing them depending on the semantic of the action.

Finally, the "MacroCommand" is carried out using one of the supported data structures in Java. The data structure stores objects of type "Command", and when a macro action is requested, the "execute()" method traverses the data structure and delegates the action to the "execute()" command (or method) to each individual "Command" object in the structure.

8.2 Command Design Pattern in Erasmus

In the same way that an object in object-oriented programming encapsulates a command, a process in Erasmus is the component that encapsulates commands in them. The encapsulation of an action is part of the process's nature, since the principal job of a process is to carry out a functionality.

8.2.1 Structure and Actors

Figure 19 demonstrates the design of the Command Pattern in Erasmus.

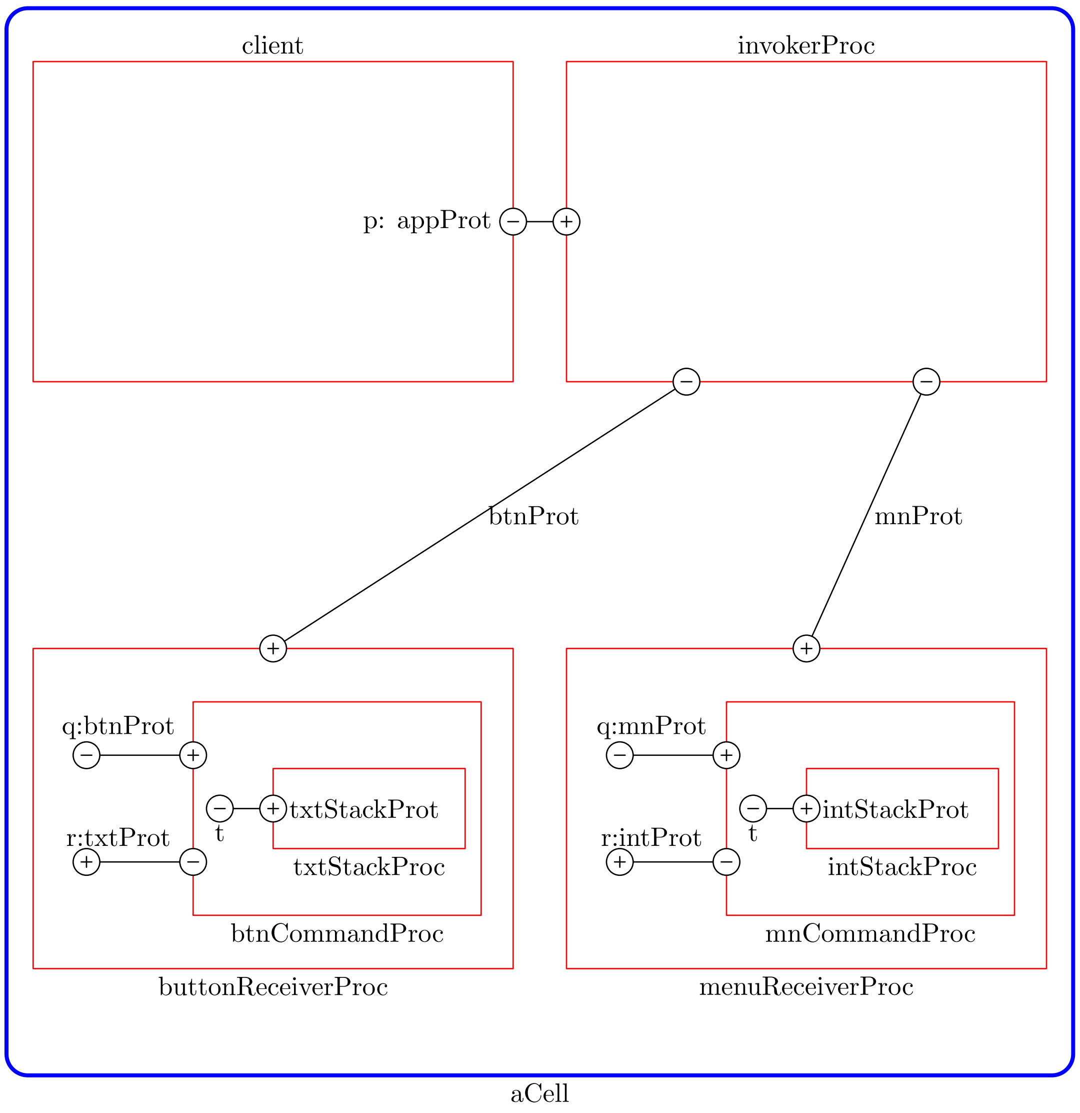

Figure 19: Command Design Pattern in Erasmus

The participating protocols, processes and cells that are involved in building up the Pattern are explained here:

- Protocols

  - appProt: encapsulates the requested data that is sent from the “client”, and to be processed. In this case the data contains a Text type variable that belongs to the button command, a signal “btnUndo” to undo the last requested operation, an Integer type variable that belongs to the menu command, and a signal “mnUndo” that undoes the last requested operation.
  - btnProt: is a subset of “appProt”, it encapsulates the data that is associated with the button command.
  - txtProt: is a subset of “btnProt”, it only includes the data part, and not the undo part of the “btnProt”. Using “btnProt” instead of “txtProt” is also possible.
  - mnProt: is a subset of “appProt”, it encapsulates the data that is only concerned with the menu command.
  - intProt: is a subset of “mnProt”, it only includes the data part, and not the undo part of the “mnProt” protocol. Using “mnProt” instead of “intProt” is also possible.

- Processes
  - client: is the process that sends requests to be handled.
  - invokerProc: is a process that redirects the request to the appropriate receiver and command processes to handle the request.
  - menuReceiverProc: is the server or receiver of the request, it handles the request that is associated with menus. It creates dynamically a corresponding command process, “mnCommandProc” in this case. The pushed integer values are summed up; this operation represents the functionality of the menu in this example.
  - buttonReceiverProc: is the server or receiver of the request, it handles the request that is associated with buttons. It creates dynamically a corresponding command process, the “btnCommanProc” process. In this example, the popped text is sent back to the Receiver, that displays it.
  - mnCommandProc: encapsulates the command that is related to the menu, and supports the undo functionality.
  - btnCommanProc: encapsulates the command, that is related to the button, and supports the undo functionality.

- Cells
  - ctl: instantiates the processes “client”, “invokerProc”, “buttonReceiverProc”, “menuReceiverProc” and runs the program.

8.2.2 Consequences

Similar to the OOP, the Command Pattern in Erasmus has a client, invoker, command, and receivers. However, these components are processes and not objects.

The flow of the design starts by initiating and sending requests from the client process. The invoker ("invokerProc") works as a dispatcher or delegate; it redirects the request to the appropriate Receiver ("buttonReceiverProc" or "menuReceiverProc"). The Receiver is the process that is responsible for handling the request. It also forwards the request to the appropriate command process ("btnCommandProc" or "mnCommandProc") which encapsulates the event or the command by storing it in a stack. The stack is used later in managing the undo operation.

The stack is implemented in a process, that is imported to the program as a library. In the current design of Erasmus the compiler does not allow generic type handling in a process, but this is to be changed in later versions of the compiler.

8.2.3 Implementation

The library that implements the stack is shown in Figure 20. The Figure shows two processes, one processes integer values, and the other one processes data of type Text. As mentioned above, this implementation is not generic, for the current limitation of the compiler.

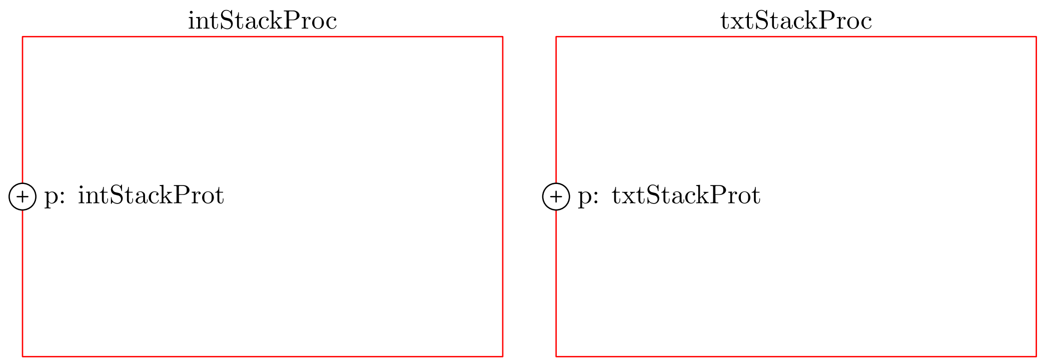

Figure 20: Stack Library

The design of the Command Pattern makes it possible to extend the current application into processing new commands. The "Invoker" is extended to delegate the new request to a new "Receiver", and the "Receiver" sends a command to an appropriate "Command" process to store and manage the command.

8.2.4 Code

The starting point as mentioned above is the "client" process that issues the requests. The requests are sent to the invoker. The invoker, "invokerProc", is implemented as follows:

```
invokerProc = process p: +appProt; q: −btnProt; r: −mnProt|
  loopselect
    || str: Text := p.btnTxt;
      q.btnTxt := str;
    || p.btnUndo;
      q.undo;
    || i: Integer := p.mnInt;
      r.mnInt := i;
    || p.mnUndo;
      r.undo;
    || p.stop; q.stop; r.stop; exit
  end
end
```

The "invokerProc" behaves as a server in respect to the data sent from the "client" process. It breaks down the input message into messages that comply to the request handlers, i.e. receivers. It then forwards, if any, the messages to the appropriate receiver. The connection between the invoker and the receivers is shown in the cell "ctl" code.

```
...

q: btnProt;
r: mnProt;

...

invokerProc(p, q, r);
  buttonReceiverProc(q);
  menuReceiverProc(r);
...
```

The code of a receiving process, the "buttonReceiverProc" process, is as follows:

```
buttonReceiverProc = process p: +btnProt |
  q: −btnProt;
  r: +txtProt;
  btnCommandProc(q, r);
  loopselect
  ||  str: Text := p.btnTxt;
      q.btnTxt := str;
      sys.out := "button Text: " + str + '\n';
  ||  p.undo;
      q.undo;
      aStr: Text := r.txt;
     sys.out := "Button Undo result: " + aStr + "\n";
  ||  p.stop; q.stop; exit
  end
end
```

The “buttonReceiverProc” process instantiates the corresponding command process, which is in this case “btnCommandProc”. It sends the command or the request as a message of protocol type “btnProt” and receives back the data that results from the “undo” action back to the receiver to be handled as a message of protocol type “txtProt”, in the case an “undo” operation is requested.

The command process “btnCommandProc”, for example, has two ports of protocol type “btnProt” and “txtProt”. The port of type “btnProt” receives the requested event data and stores it in a stack. When an undo event is requested, the “btnCommandProc” process sends back the top data of the stack. The data type that is stored in the stack in this example is of type Text. The “btnCommandProc” process instantiates the stack process “txtStackProc”, which is used to store the requested events.

```
btnCommandProc = process p: +btnProt; q: −txtProt|
  t: −txtStackProt;
  txtStackProc(t);
  loopselect
    || str: Text := p.btnTxt;
      t.pushVal := str; t.push;
    || p.undo;
      t.pop;
      aStr: Text := t.popVal;
      q.txt := aStr;
    || p.stop; t.stop; exit;
  end
end
```

The stack process implementation is shown in the following piece of code. The protocol “txtStackProt” is composed of a data of type Text, to be stored, a signal to store (“push”), another signal to pop the value from the stack (“pop”), and the popped value (“popVal”) that is sent back to the requester (in this case the process “btnCommandProc”).

The stack process expects a message of type Text followed by a “push” signal to store the data into the stack or data structure. To pop the data from the top of the stack, a “pop” signal is needed, then the stack replies by sending back the value from the top of the stack to the calling process.

```
...

txtStackProt = protocol
  ∗(pushVal: Text | push | pop; ↑popVal: Text);
  stop
end

...

txtStackProc = process p: +txtStackProt |
  stack: Integer indexes Text;
  count: Integer := −1;
  loopselect
    || txt: Text := p.pushVal; p.push;
      count +=1;
      stack[count] := txt;
    || p.pop;
      if count < 0
      then sys.out := "Stack Overflow!\n"
      else
        txt: Text := stack[count]; count −= 1;
        p.popVal := txt;
      end
    || p.stop; exit
  end
end
```

8.3 Summary

In this chapter we have shown how to implement the Command Pattern in Erasmus. The Erasmus implementation differs from the OOP implementation in that Erasmus does not need to specify a uniform method name (or a process) such as “execute()” to carry out the action or the functionality. The fact that each process carries out an action or a functionality by nature in the implementation of Erasmus's abstraction, makes defining a unified method or entity to carry the action unneeded. So the advantage of the Erasmus version when compared to the OOP version lies in the abstraction of the Erasmus language that makes implementing the Command Pattern intuitive. However, the disadvantage of the current version of µE is in lacking generic type handling, which requires coding a stack process for each type of data that is stored in the stack.

Chapter 9

Behavioral Pattern: Interpreter Design Pattern

9.1 Interpreter Design Pattern in OOP

Recurring problems can be expressed by languages that have their own grammar. To process the language defined by the grammar, the Interpreter Design Pattern is a software blueprint in which the grammar is represented and translated. As we will see in the following sections, the Interpreter pattern is very much like the Composite Pattern. Both Patterns share the part-whole representation, although it is more specific in the Interpreter Pattern because the composition represents the grammar of the language. The following discussion is based on [GHJV95].

9.1.1 Definition

"Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language" [GHJV95].

9.1.2 Background

The Interpreter Pattern represents all the elements of a grammar such as expressions and symbols. Each grammar element is represented by a class. The design of a grammar follows the Composite Pattern, in the sense that an expression is seen as a composition of subexpressions, and symbols represent leaves. This design makes it possible to plug new rules in a grammar and is an efficient representation of a tree structure. However, if the grammar is complicated or big in size, it is recommended not to use this pattern.

An important distinction between the Composite Pattern and the Interpreter is that the interpreter translates an expression and returns a simple result.

9.1.3 Structure and Actors

As it is shown in Figure 21, the Interpreter Pattern is the Composite Pattern if the "AbstractExpression" is replaced by "Component", "TerminalExpression" by "Leaf", and "NonterminalExpression" by "Composite". The new class that is not included in the Composite Pattern is the "Context" class. The "Context" class carries information that is used globally by the Pattern. The "Context" class usually stores information in a data structure.

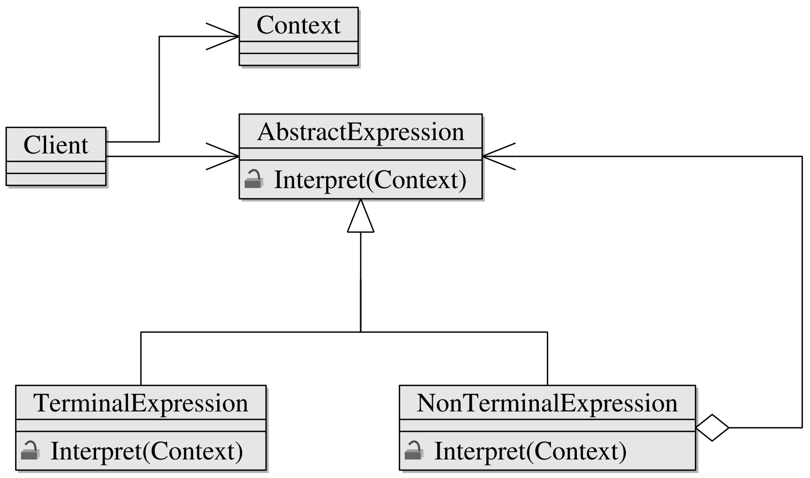

Figure 21: Interpreter Design Pattern in Object-Oriented Programming

The following are the actors of the Interpreter Pattern:

- Client: invokes the Interpreter with a syntax tree of the language that is represented by this Interpreter. The syntax is composed of the elements of that grammar: expressions and symbols.
- Context: stores information that represents the syntax of the grammar and that is related to the Interpreter.
- AbstractExpression: is the abstract representation of the expressions that are the elements of a given grammar. This abstract works here like the "Component" Class in the Composite Pattern.
- TerminalExpression: represents the symbols of the grammar. It works like a "Leaf" in the Composite Design Pattern. It is the indivisible unit in the tree (for example

"+", "-", "and", "or", etc.).

- NonterminalExpression: is in other words the "Composite" Class in the Composite Pattern. It is composed, in turn, of other components (subcomponents).

9.1.4 Implementation Decisions

This Pattern is not used if:

- The grammar is complex, because the class hierarchy becomes big in size, and difficult to manage or maintain.

- Efficiency is a concern, because if efficiency is required a translation of the parse tree to another form is needed first (the parse tree is a tree that represents the syntactic structure of an expression. The grammar defines the set of all possible parse trees).

The benefit of having this Pattern is that it makes it easy to change or add more features to the grammar. This could be done by adding new expression classes. Adding or changing expressions does not need refactoring of the design.

The implementation of a grammar is not that difficult. Each expression is represented by a class, symbols are the end of the recursion or the tree of the language grammar. In general a nonterminal expression is a composition of one or two expressions and a symbol (terminal expression). The symbol acts (depending on its logic) on the expression(s). For example "x + y" is translated to adding the content of the variable x to the content of the variable y (let us assume that both are of numeric type). In this example both variables "x" and "y" are nonterminal expressions, and "+" is a symbol that behaves as it is described in the grammar. That is why this Pattern becomes unmanageable when the grammar is complex and big in size: maintaining many classes (that represent the grammar) is not easy.

Defining new ways of interpretation of the given expressions of a grammar is made easier by using this Pattern. So new evaluations or translations could be plugged into the grammar representation. However, if the grammar can have many translations, it is better to incorporate other design patterns such as the Visitor Design Pattern, which is not covered in this document.

In the Composite Pattern, the "Component" is well defined in the sense of how to design the tree, where to allocate the pointers to the parent and to the children. However, the Interpreter leaves the implementation free to the choice of the programmer.

In some cases it is wiser to separate the representation part of the grammar from the interpretation of it. This is done when the grammar can have many interpretations.

9.1.4.1 Java Implementation

In Java, the Interpreter Pattern follows very much the same way the [GHJV95] describes the design. However, in the given example of the document [Coo00], some packages which are supported in Java were used in building the Pattern:

- Using the Stack class which is an extension of the Vector data structure. The Stack Class, that is part of the "Context" class, allows the parsing tree elements (or expressions) to be stored following the parsing tree structure of the formal language.
- Using the StringTokenizer class. StringTokenizer scans the input string that represent the phrase of the formal language into tokens, each of these tokens represent an expression.

In [Coo00], the objects that construct the interpreter incorporate other design patterns, the parse tree, for example, is represented by a Chain Of Responsibility Pattern. The stack is passed through the chain until an appropriate handler of the expression is found. Also the parser class is an implementation (inherits the interface) of the Command Pattern.

9.2 Interpreter Design Pattern in Erasmus

Similarly to the Interpreter Pattern in object-oriented paradigm where each expression is represented by a class, the Interpreter Pattern in Erasmus represents each expression by a process. The Interpreter Pattern of this design represents and interprets the following grammar:

Expression ::= *Number* | *Variable* | *Expression Operation Expression.*
Variable ::= ('a' | 'b' | ... | 'z')∗.
Number ::= (0 | 1 | 2 | ... | 8 | 9)∗.
Operation ::= *Plus* | *Minus* | *Multiplication* | *Division*.
Plus ::= '+'.
Minus ::= '−'.
Multiplication ::= '∗'.
Division ::= '/'.

Each of the above expressions is represented by a process, or if more than one process is involved in handling the request, a cell is used to encapsulate the handling processes instead. Figure 22 shows the Interpreter Design Pattern. The program that demonstrates the Interpreter Pattern in Erasmus supports the following functionalities:

- Scanner: gets a language expression as a string. It breaks the expression into valid tokens as defined by the grammar. Each valid token is interpreted into an expression that consists of a “kind” (that is an element type declared in an enumeration named “States”) and a “value” that represents the value of the token.

- Parser: analyses the tokens following the rules of the grammar. It delegates the tokens to the appropriate process depending on the type (“kind”) of the token.
- Expressions representations: in which the expressions “Number”, “Variable”, and “Operation” are encapsulated in a process (or in a cell). Each representation of an expression handles what the rules of the grammar specify.
- Calculator: gets the left hand operand, the right hand operand, and the operation, and returns back the partial result.
- Variable Replacer: replaces variables, based on the name of the variable, with their corresponding values.

9.2.1 Structure and Actors

The Interpreter consists, as shown in Figure 22, of the following:

- Protocols
  - tokProt: is a protocol that breaks down the given expression into valid tokens. A token consists of a “value” and a “kind”. A “kind” is a variable of enumeration type “States”. The kind of a token that is sent to the parser could either be a “number”, “name”, “op”, or “stop”.
  - interpretProt: is used by the processes “numberProc” and “variableProc” to get a “token” of type Text and returns the “result” value of type Integer.
  - varValProt: includes two messages: a “name” of type Text, and a “value” of type Integer.
  - operationProt: sends in one way as many times as wanted, an “operand” of type Integer or an operation “op” of type Char. It sends back (in the other way) a value that is named “result” of type Integer.
  - calculationProt: sends in one way an operation value “op” of type Char, left hand side operand “lhs” of type Integer, and a right hand side operand “rhs” also of type Integer, and it sends in the other way the result of the operation on both operands “result” of type Integer.
- Processes

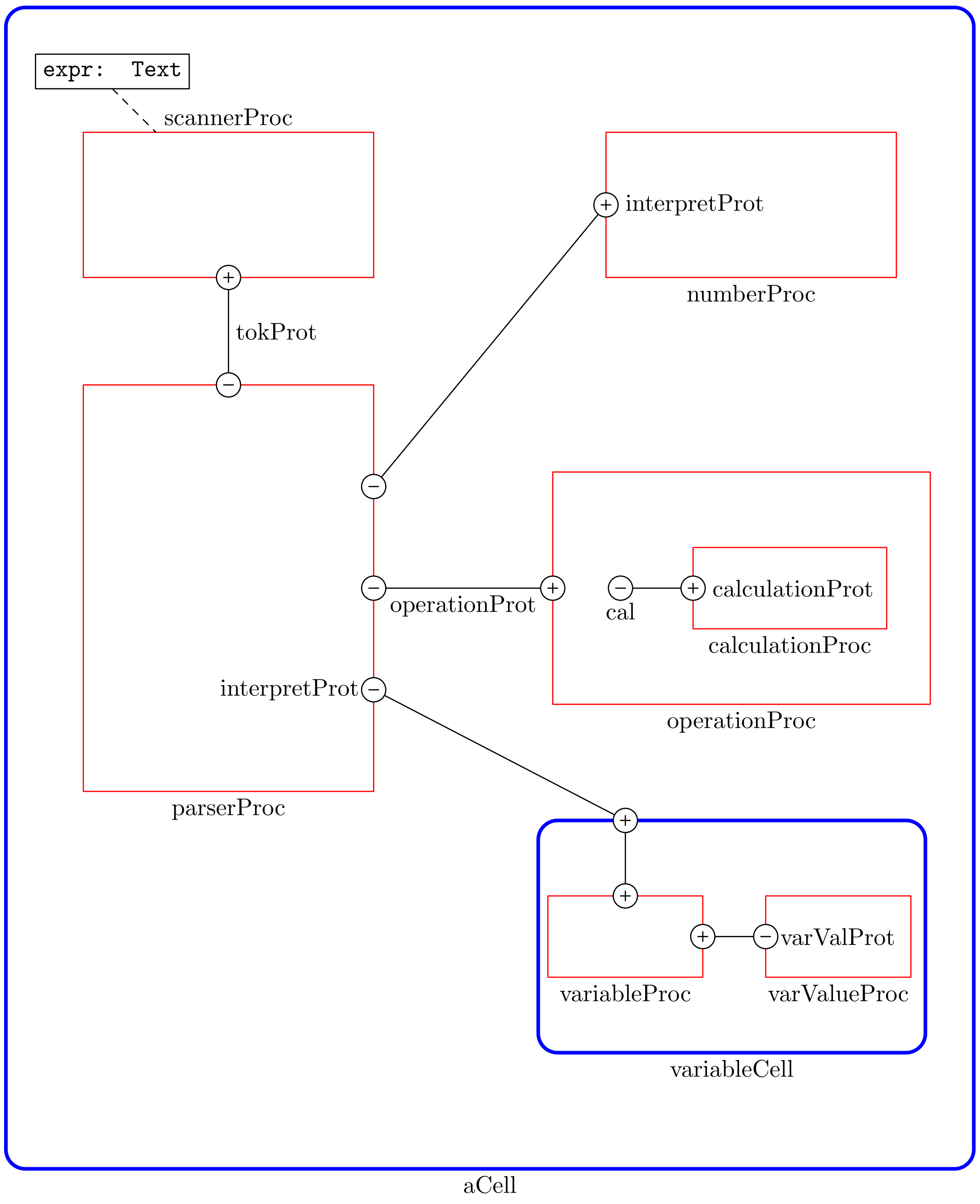

Figure 22: Interpreter Design Pattern in Erasmus

  - scannerProc: scans the input and converts it into tokens, if valid, or prints an error message and exits, if the string does not follow the rules of the grammar.
  - parserProc: gets the tokens from the process “scannerProc”, passes them as messages (either of “interpretProt” or “operationProt”) according to their types (“kind”) to the appropriate process to convert them into Integer values and to calculate the partial result.
  - numberProc: gets token values that have a “kind” equivalent to “number”. It converts the values from a Text type to an Integer type.
  - variableProc: gets tokens that have a “kind” equivalent to “name”. It gets the values of each variable from the process “varValueProc”.
  - varValueProc: contains the variables and their corresponding integer values. This process exchanges messages with the process “variableProc” to pass the value of each variable that appears in the input sentence.
  - operationProc: gets the operands and the operation from the “parserProc” process. Once the left hand side operand and the right hand operand and the operation symbol are ready, it sends their values to the process “calculationProc” to send back the partial result. This process holds the accumulation of the partial results in it, and it sends the final result to the process “parserProc”. The right hand side operand holds the accumulation of the partial results.
  - calculationProc: receives two operands and an operation; depending on the operation, the result of applying the operation on the operands is sent back to the process “operationProc”.

- Cells

  - variableCell: it encapsulates the processes that handle token of type “name” which are variables. It instantiates the two processes “variableProc” and “varValueProc”, and creates a channel “r” of protocol type “varValProt” to connect both processes.
  - aCell: creates channels “p” of protocol type “tokProt”, “qN” and “qV” of protocol type “interpretProt”, and “qO” of protocol type “operationProt”. It instantiates the processes “scannerProc”, “parserProc”, “numberProc”, and “operationProc” and the cell “variableCell” and connects them together using the created channels. This cell runs the program with an input string (sentence) “temp+53*var-20/y” that follows the grammar of the program.

9.2.2 Consequences

Because each expression is represented in a separate process, extending the grammar is possible. The only possible restriction would be maintainability if the grammar supports many expressions: the process “parserProc” will have an interface with many parameters, which makes it difficult at some point to maintain the code.

Supporting more operations in the expression “Operation” that is represented by the process “operationProc” is also possible and is made easy, because the process “calculationProc” encapsulates the possible operations that are listed in the grammar. An example of an operation that could be added is the mod operation represented by “%”.

Finally the process “varValueProc” encapsulates the variables names’ and their values. Consequently the content of this process must be updated according to the input string (sentence) that is passed to the process “scannerProc”.

9.2.3 Implementation

The scanner that is encapsulated in the process “scannerProc” breaks the input expression into tokens. Tokens are sent as messages of protocol type “tokProt”. The “tokProt” protocol is composed of a variable “value” that is of type Text, and a variable “kind” that is an element of enumeration type “States”. The “States” enumeration contains the following elements: “commence”, “number”, “name”, “op”, and “stop”. The element “commence” declares that scanning has started, “number” represents the expression “Number” in the given grammar, “name” represents “Variable”, “op” represents “Plus”, “Minus”, “Multiplication” and “Division” operations, and finally “stop” indicates that the input language expression has reached its end.

The order in which the variables appear in the input sentence must be the same as the order in which the variables and their corresponding values appear in the process “varValueProc”. If the variables that appear in the input sentence do not match in sequence or name with the variables listed in the “varValueProc” process, the program halts, because the exchange of messages between both processes “variableProc” and “varValueProc” blocks the whole program. The piece of code that produces the blockage is:

```
...

variableProc = process p: +interpretProt; r: +varValProt|
  name: Text;
  value: Integer;
  loopselect
    || name := p.token;
      tempName: Text := r.name;
      if tempName = name then
```

```
            p.result := r.value;
          end;
      || p.stop; exit;
      end;
    end

    ...
```

The process “variableProc” requests a message “name” from the process “varValueProc” and then performs a test on the received value in comparison to the received token from the parser. If the name is similar, the process “variableProc” requests the value through the message “value” from the process “varValueProc”. However, if the names are not similar, the process “varValueProc” is still waiting for its message “value” to be sent although the process “variableProc” is not requesting it. This blocks the processes and as a result the whole program.

To solve the sequence issue, we could change the protocol that is exchanged between the processes “variableProc” and “varValueProc” to the following:

```
    ...

    varValProt = protocol
      ∗(name: Text; ↑value: Integer);
      stop
    end

    ...
```

As a result we change the signs of the ports in both processes to meet the new changes in the protocol “varValProt”. Also we change both processes to allow the sequence in which the variables and their values appear differently from the order in which the variables appear in the input string. Figure 23 demonstrates the Interpreter pattern after applying the new changes.

```
    ...

    variableProc = process p: +interpretProt; r: −varValProt|
      name: Text;
      value: Integer;
      loopselect
          || r.name := p.token;
          || p.result := r.value;
          || p.stop; r.stop; exit;
      end;
    end

    varValueProc = process p: +varValProt|

      loopselect
```

```
        || name: Text := p.name;
          cases name
            |"temp"| p.value := 100;
            |"var"| p.value := 20;
            |"y"| p.value := 4;
          end;
        || p.stop; exit;
      end;
    end

    ...
```

This change of code does not change the fact that mismatching variables between what is in the input string and the provided list of variables in the process “varValueProc” block the program. But this time the piece of code that produces the blockage is in the process “variableProc”:

```
    ...

      || p.result := r.value;

    ...
```

If there is no matching variable name in the process “varValueProc”, then the value “r.value” is not sent. As a result, the result “p.result” is also not sent. This blocks the program because the process “variableProc” is expected to send a result.

9.2.4 Code

The full program is found in Appendix F. I list here the most important parts of the code.

This is the declaration of the enumeration “States”.

```
    States = < commence, number, name, op, stop >
    ...
```

The protocol that constructs a token that consists of a “States” enumeration element and a value of type Text. Note that the direction of the flow of the messages is opposite due to the “↑” symbol that appears before both variables “kind” and “value”.

```
    ...
    tokProt = protocol *( ↑kind: States; ↑value: Text ) end
    ...
```

The process “parserProc” forwards the token to the appropriate expression that is represented by a process depending on the value of the “kind” message. In the grammar the expressions are: “Variable”, “Number”, and “Operation”. These expressions are represented by the processes (or cell) “variableCell”, “numberProc”, and “operationProc” respectively.

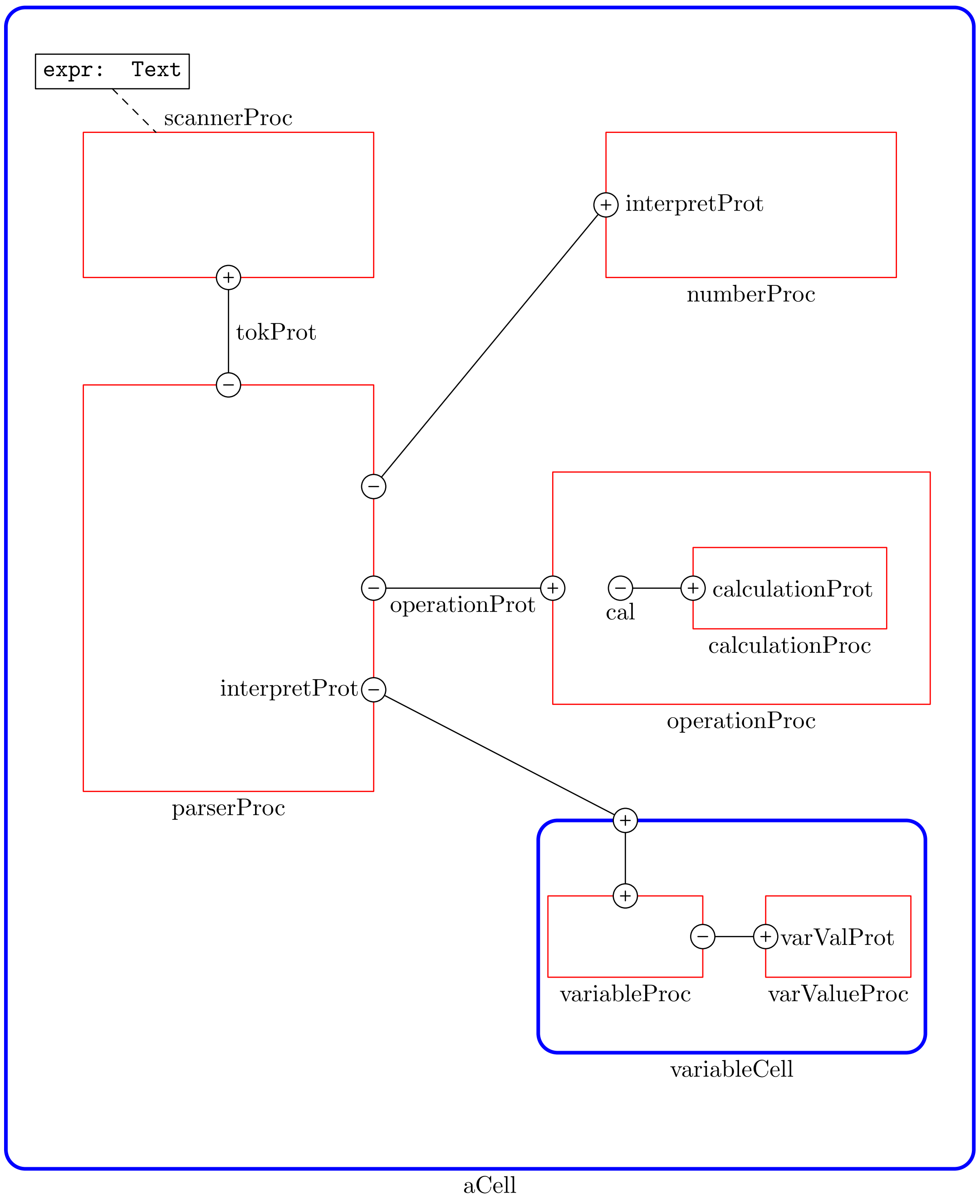

Figure 23: A modified version of the Interpreter Design Pattern in Erasmus

The terminal expressions “Plus”, “Minus”, “Multiplication”, and “Division” are encapsulated together in the process “calculationProc”. After handling the whole input string, the process “parserProc” displays the final result.

```
...

parserProc = process p: −tokProt; qNum, qVar: −interpretProt; qOp: −operationProt|
  tempVal: Integer := 0;
  first : Integer := 1;
  loop
    k: States := p.kind;
    val: Text := p.value;
    cases
      |k = number| qNum.token := val;
                   tempVal := qNum.result;
                   qOp.operand := tempVal;
                   sys.out := "Number: " // tempVal // "\n";
      |k = name|   qVar.token := val;
                   tempVal := qVar.result;
                   qOp.operand := tempVal;
                   sys.out := "Varible: " // val // " has Value: " // tempVal // "\n";
      |k = op|     qOp.op := char val;
                   sys.out := "Opeartion: " // val // "\n";
      |k = stop|   qNum.stop; qVar.stop; exit;
    end
  end;
  sys.out := "\nFinal Result: " // qOp.result // "\n";
  qOp.stop;
end

...
```

The protocol that is exchanged between the “parserProc” process on one hand and the process “numberProc” and the cell “variableCell” on the other hand is of protocol type “interpretProt”. It consists of a variable that is passed from the “parserProc” process, that is a token of type Text, and another variable “result” of type Integer, that returns the integer value of the number or the variable (after substituting it with an integer value). The “stop” signal is used to exit the loop select statement.

```
...

interpretProt = protocol
  *(token: Text| ↑result: Integer);
  stop
end

...
```

The “numberProc” process is simple, it just converts the token to an integer type, and sends it back to the “parserProc” through the message “result”.

```
numberProc = process p: +interpretProt|
  loopselect
    || p.result := int p.token;
    || p.stop; exit;
  end
end
```

The "variableCell" cell encapsulates both processes "variableProc" and "varValueProc". The process "variableProc" gets a token of type Text, and after handling it by exchanging the name of the variable with its value, it returns the result message "result" to the "parserProc" process.

```
...

variableProc = process p: +interpretProt; r: +varValProt|
  name: Text;
  value: Integer;
  loopselect
    || name := p.token;
      tempName: Text := r.name;
      sys.out := "name: " // name // " and gotten Name: " // tempName // "\n";
      if tempName = name then
        p.result := r.value;
      end;
  || p.stop; exit;
  end;
end

...

...

variableCell = cell p: +interpretProt|
  r: varValProt;
  variableProc(p, r);
  varValueProc(r);
end

...
```

Finally, the operation expression is represented by the process "operationProc". This process gets the right hand side and the left hand side operands and the operation sign, it sends them, when all are ready, as messages to the instantiated process "calculationProc", and then gets back the partial result. This process keeps the up-to-date result.

```
...

operationProc = process q: +operationProt;|
  operation: Char;
  sum, operand: Integer := 0;
   first : Integer := 1;
```

```
lhsFlag, rhsFlag: Bool := false;
loopselect ordered
  || operand := q.operand;

    if first = 1 then
      sum := operand;
      first += 1;
    else
      cal: −calculationProt;
  calculationProc(cal);

  cal.op := operation;
  cal.lhs := sum;
  cal.rhs := operand;
  sum := cal.result;
  cal.stop;
    end;
  || operation := q.op;
  || q.result := sum ;
  || q.stop; exit;
end
end

...
```

9.3 Summary

In this chapter we have shown how to implement the Interpreter Pattern in Erasmus. The Erasmus implementation differs from the OOP implementation in that Erasmus represents the expressions of the grammar in a process (or a cell if more than one process is involved in providing the functionality), while in OOP expressions are represented by classes. The Erasmus version has the advantage of low coupling between processes, which makes designing individual processes easier. However, the design of protocols and messages to be exchanged among processes requires good study and attention, because not understanding the sequence and flow of message exchange between processes may result in deadlocks. Even though designing parameters of methods in OOP classes appears not to be as difficult as designing parameters of processes in Erasmus, thoughtless design may cause in high coupling between classes.

Chapter 10

Behavioral Pattern: Iterator Design Pattern

10.1 Iterator Design Pattern in OOP

The idea of the Iterator Pattern is to ensure a uniform interface for traversing aggregates or collections without having to know the internal implementation or the mechanism of achieving the iteration. The implementation of how and what to iterate is left to the application and to the programmer. The Pattern ensures as well a uniform interface for creating iterators in aggregates.

This chapter discusses the Iterator Pattern based on the references [GHJV95], [Met02], [Coo00], and [FFBS04].

10.1.1 Definition

“Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation” [GHJV95].

10.1.2 Background

The Iterator Pattern is normally a feature built into most modern object-oriented programming languages. However, sometimes programmers need to build their own customized iterators, and that is where the Iterator Pattern comes in handy.

The main idea of the Iterator Pattern is to separate a collection or an aggregate object from traversing it, by putting the traversing or iterating mechanism in another object, which is called the Iterator.

Also having a uniform interface that contains the supported functionalities that iterators provide to all types of aggregates or collections simplifies using packages or libraries that are related to collections and their corresponding iterators.

The Iterator and the Aggregate are coupled, so is the user (or the Client object we will see later in Section 10.1.3) with both. A better idea would be to give more flexibility to the Aggregate object and the Iterator for future changes, without having to change the code of the Client object. This is done by introducing polymorphic Iterator and polymorphic Aggregate and that is how the Iterator Pattern is implemented.

Also the design gives Aggregate objects the responsibility of creating their corresponding Iterator. A "CreateIterator()" method is introduced in the Aggregate interface in which an Iterator object is created. The "CreateIterator()" is implemented as a Factory Method (See Factory Method Pattern in Chapter 4).

10.1.3 Structure and Actors

The Iterator Design Pattern Class diagram is shown in Figure 24.

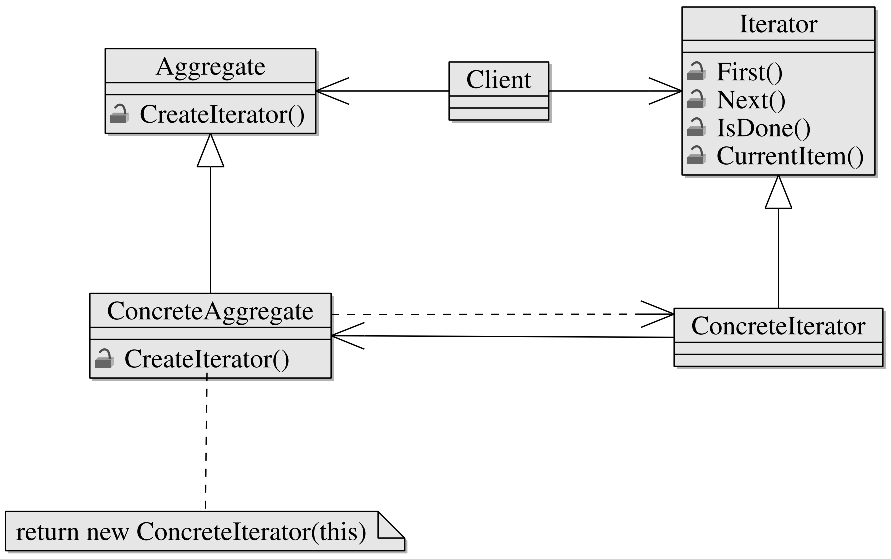

Figure 24: Iterator Design Pattern in Object-Oriented Programming

The actors of this Pattern are:

- Client: is the user of "Aggregate" objects. The "Client" traverses through the Aggregates using Iterators.
- Iterator: is an interface that makes iterating an aggregate uniform to its users.

- ConcreteIterator: is an actual instance that implements the “Iterator” Interface. The mechanism of traversing aggregates varies depending on how the “ConcreteAggregate” is implemented.
- Aggregate: is an interface that represent a collection of objects of the same type. This interface provides a method that creates an “Iterator” object with which the aggregate is traversed.
- ConcreteAggregate: is the concrete instance of the “Aggregate” Interface. The method “CreateIterator()”, which is implemented in “ConcreteAggregate”, returns an Object of type “Iterator”.

10.1.4 Implementation Decisions

The original design of the Iterator interface supported the following methods:

1. First(): this method resets the iterator to start from the beginning of the collection. In other words it resets the index of the aggregate to point to the first item.
2. Next(): it increments the index of the aggregate to point to the following item if it exists.
3. isDone(): checks if the index of iteration refers to an item in the collection or aggregate.
4. CurrentItem(): returns the object or the item, if any, from the aggregate that is pointed at by the index.

As we will see later in Java Implementation, some object-oriented languages substituted these methods by others that meet their requirements and that follow the idea of iteration.

One has to be aware of the consequences of the Iterator Pattern, such as knowing the impact of deleting or adding an item while traversing the collection. Another consequence is to know whether the iteration is single threaded or multi-threaded. Depending on that the Iterator may be synchronized to ensure thread safety or not.

In [GHJV95] the term “internal iterator” and “external iterator” are mentioned. The difference between both types is that the “external iterator” leaves the control of iteration to the user of the iterator object, where as the “internal iterator” is given an operation to perform; it gives back the control to the user after the operation is completed.

10.1.4.1 Java Implementation

Java Implements the Iterator Pattern in two interfaces the old one is called the "Enumeration" and the new one is the "Iterator". In [608a], it is recommended to use the "Iterator" interface instead of the "Enumeration".

The methods that are provided in the "Enumeration" interface are:

1. hasMoreElements(): checks whether the iterator is still pointing to a valid item in the aggregate.

2. nextElement(): returns back the item or element the iterator is pointing to currently.

The aggregates "Vector" and "Hashtable" in Java do not implement the "Enumeration" interface, but they have built in the functionalities that are in the "Enumeration" interface. However, all aggregates in Java including the "Vector" class and the "Hashtable" implement the "Iterator" interface.

The "Iterator" interface came up with new names of methods that are shorter than the ones used in the "Enumeration" interface, but do the same job. In addition the "Iterator" interface introduced a new method called "remove()". The following are the supported methods in the "Iterator" interface [608b]:

1. hasNext(): returns a boolean value, the method is similar in functionality to "hasMoreElements()" in the "Enumeration" interface.

2. next(): returns the item or the element the iterator is pointing at. This method is similar in functionality to the method "nextElement()" in the "Enumeration" interface.

3. remove(): this method removes the current item that is pointed at by the iterator. This operation is the only safe way of modifying the aggregate during iteration [Tut08].

The method "First()" is omitted in Java, because it is implicitly achieved whenever a new "Iterator" (or an "Enumeration") object is created.

The disadvantage of iterators or even enumerations in Java is the fact that the items or elements they return have to be casted to the proper object. This is due to the strong typing of the Java language. So as a result whatever is returned from the method "next()" or "nextElement()" has to be casted before it could be used.

10.2 Iterator Design Pattern in Erasmus

The Iterator Pattern is a built-in feature in Erasmus. It is partly provided by the language by "Comprehension". The supported aggregate type in Erasmus is the map type. The built-in iterators provide the facility of traversing a map either by domain or by range without having to know the internal mechanism of the traversal. The following is the syntax of statements: "Any" and "For" that traverse the aggregate type "Map" [GS09]:

```
Any = any Comprehension do Sequence else Sequence end.
For = for Comprehension do Sequence end.
Comprehension = VarName [ `:' Type ] [ in Set ] [ such that Rvalue ].
Set = Rvalue to Rvalue [ step Rvalue ]
| Rvalue `≤ ' `(' [ Rvalue ] `)' ( `<' | `≤ ' ) Rvalue
| Rvalue `≥ ' `(' [ Rvalue ] `)' ( `>' | `≥ ' ) Rvalue
| [ domain | range | Rvalue
| Type.
```

Based on this above syntax, the following example demonstrates how iterators work on a map in Erasmus.

```
testProc = process |
  map: Integer indexes Integer;
  map[1] := 10;
  map[2] := 100;
  map[3] := 1000;
  map[4] := 10000;
  sys.out := "Iterate  through the domain\n";
  for i: Integer in domain map such that i < 3 do
    sys.out := map[i] // '\n';
  end

  sys.out := "Iterate  through the range\n";
  for i: Integer in range map such that i > 50 do
    sys.out := i // '\n';
  end
end

mainCell = cell |
  testProc();
end

mainCell();
```

The execution of the program produces the following output:

```
Iterate through the domain
10
100
Iterate through the range
```

```
100
1000
10000
```

The two following pieces of code are the key parts that encapsulate iteration by providing the functionality of traversing the aggregate of type map, "map" in the example, either by its index (domain), or by the stored value (range) without having to use any keys to iterate through the domain or the range. The keys are encapsulated or built in the language and implemented internally.

for i: *Integer* **in domain** *map* **such that** $i < 3$ **do**

and

for i: *Integer* **in range** *map* **such that** $i > 50$ **do**

Note that the "Type" that is mentioned in the definition of "Comprehension" could be omitted, because the compiler can deduce the type of the iterator variable. So the "for" statements could be rewritten as:

for i **in domain** *map* **such that** $i < 3$ **do**

and

for i **in range** *map* **such that** $i > 50$ **do**

However, designing an Iterator Pattern for other uses than the ones supported by the language is also possible in Erasmus.

In Section 3.7.7, the point that discusses passing Maps as parameters in Erasmus promoted the idea of having two types of design in implementing the Iterator Pattern in Erasmus. One that does not support sharing aggregates among processes and cells, and the second one that shares aggregates within a cell and its components.

10.2.1 Iterator Pattern in Erasmus without sharing Aggregates

In this approach, the design merges both components Aggregate and Iterator in one. This way the new component that is encapsulated in a process (or as we will see later on in a cell) behaves as a keeper of a collection or aggregate and as an iterator on this collection.

The following discussion of the Pattern is incremental. It starts by a simple demonstration of the design and continues until it reaches its final version of the Pattern.

10.2.1.1 Structure and Actors

The basic representation of the Iterator Pattern is to have one client that uses one Aggregate Iterator component. Figure 25 demonstrates the design.

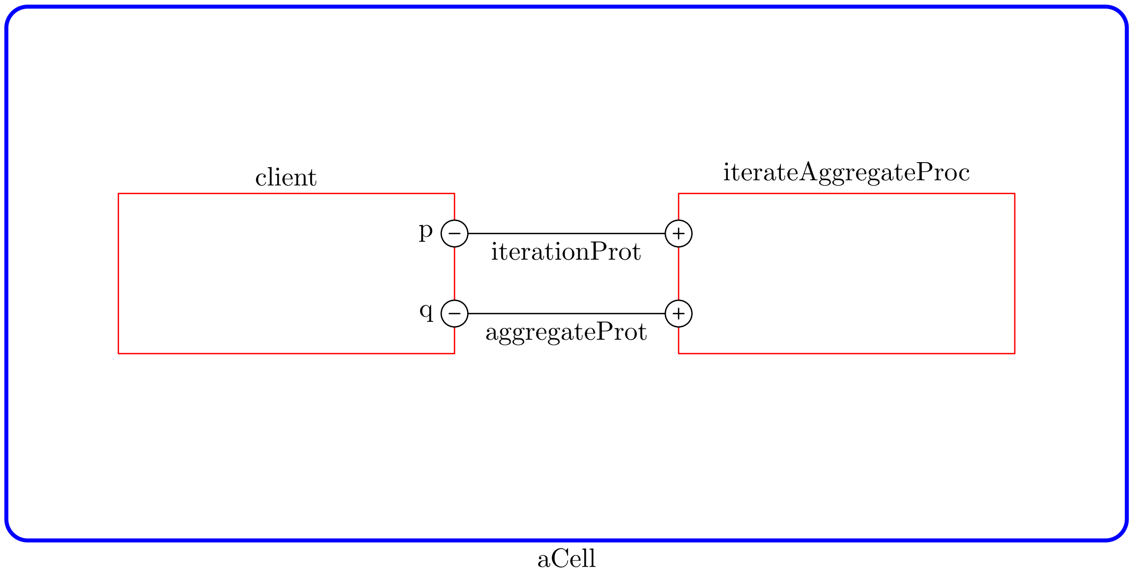

Figure 25: Simplified Iterator Design in Erasmus

The protocols, processes, and cells are explained as follows:

- Protocols:
  - iterationProt: this protocol is concerned with all operations that are related to the Iterator. It supports in this case the following operations:
    1. add: a signal to declare that adding an item is to follow.
    2. next: increments the index by one to move to the following item.
    3. hasNext: a signal to indicate that the following message is to be sent to the client indicating whether we have reached the end of the aggregate or not.
    4. hasNextBool: returns to the client a boolean value; the value is true if the iterator did not reach the end of the aggregate, and false if it did.
  - aggregateProt: this protocol supports functionalities and messages related to storing data into the aggregate. The following are the elements of the protocol:
    1. addTxt: is the item message, that carries the information to be stored in the aggregate.
    2. getNextTxt: is the item that is currently pointed at during iteration. This message sends back the item to the client, when requested (the request is achieved by sending the signal “next” from the “Client” to the “iterateAggregateProc” processes).

3. error: an error signal is sent if the iterator has reached the end of the aggregate.

- Processes:
  - client: this process creates the data, of type Text in this example, to be stored, and initiates requests to the "iterateAggregateProc" process to iterate through the aggregate.
  - iterateAggregateProc: encapsulates both behaviors: the aggregate data keeping and iterating through that aggregate.
- Cells:
  - aCell: instantiates both processes "client" and "iterateAggregateProc", connects the processes, and runs the program.

The program in Figure 25 is quite simple, to extend this program to support many aggregates iterators, Figure 26 demonstrates a possibility of extension.

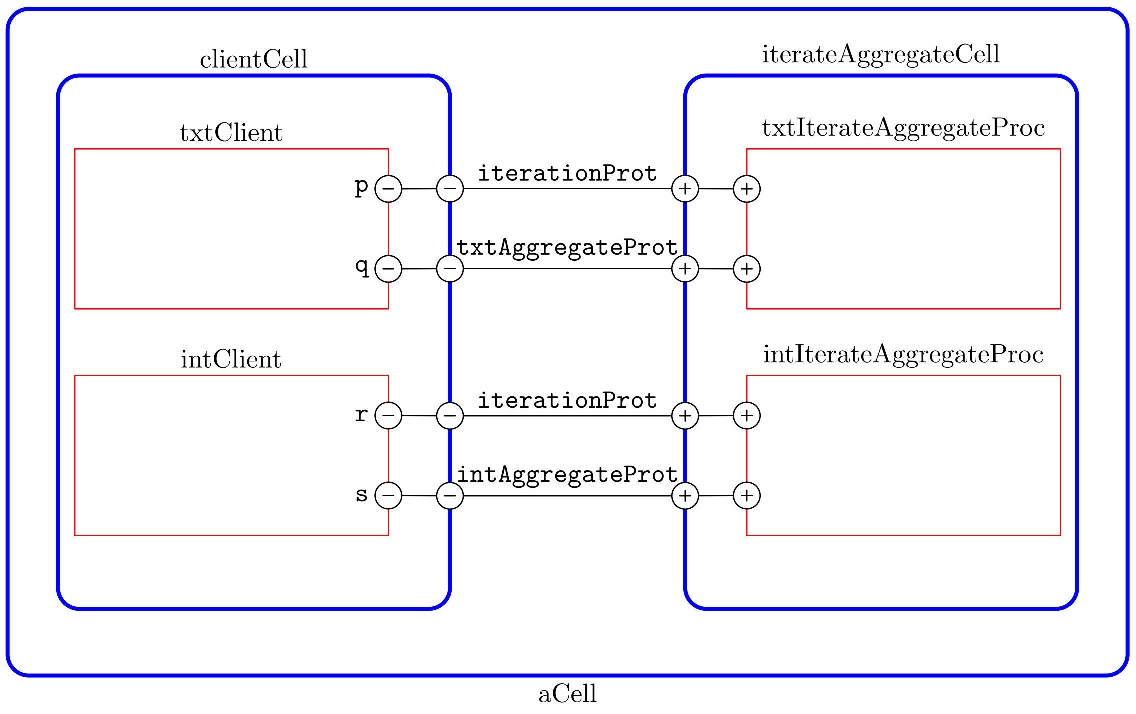

Figure 26: More Elaborate Iterator Design in Erasmus

The components involved in building the design are:

- Protocols:
  - iterationProt: this protocol is concerned with all operations that are related to the Iterator. It supports in this case the following operations:
    1. add: a signal to declare that adding an item is to follow.
    2. next: increments the index by one to move to the following item.
    3. hasNext: a signal to indicate that the following message is to be sent to the client indicating whether we have reached or not the end of the aggregate.
    4. hasNextBool: returns to the client a boolean value; the value is true if the iterator did not reach the end of the aggregate, and false if it did.
  - txtAggregateProt: this protocol supports messages related to storing data of type Text into the aggregate. The following are the elements of the protocol:
    1. addTxt: is the item message, that carries the information to be stored in the aggregate.
    2. getNextTxt: is the item that is currently pointed at during iteration. This message sends back the item to the client, when requested (the request is achieved by sending the signal "next" from the "clientCell" cell to the "iterateAggregateCell" cell).
    3. error: this signal is sent if the iterator reached an index that is beyond the size of the aggregate.
  - intAggregateProt: this protocol supports messages related to storing data of type Integer into the aggregate. The following are the elements of the protocol:
    1. addInt: is the item message of Integer type, that carries the information to be stored in the aggregate.
    2. getNextInt: is the item that is currently pointed at during iteration. This message sends back the item, which is of type Integer, to the client when requested.
    3. error: this signal is sent if the iterator reached an index that is beyond the size of the aggregate.
- Processes:
  - txtClient: this client stores information of type Text at the process "txtIterateAggregateProc" and exchanges messages with it to iterate through the items.

  - intClient: this client stores information of type Integer in the "intIterateAggregateProc" process and sends messages to the same process to iterate through the items.
  - txtIterateAggregateProc: encapsulates both behaviors: the aggregate data keeping and iterating through that aggregate. The aggregate supports data of type Text.
  - intIterateAggregateProc: encapsulates both behaviors: the aggregate data keeping and iterating through that aggregate. The aggregate supports data of type Integer.

- Cells:
  - clientCell: encapsulates clients' processes.
  - iterateAggregateCell: this cell encapsulates processes that combine both behaviors: storing the aggregates and iterating through them.
  - aCell: instantiates both cells "clientCell" and "iterateAggregateCell", connects the processes, and runs the program.

This design is more flexible than the previous one, but it is still not dynamic enough to support more than two aggregate iterators. Further explanation is found in Section 10.2.1.3.

The limitation of the last design is fixed in the following design by introducing a factory process. This process instantiates dynamically aggregate iterators depending on the clients' request. Figure 27 demonstrates the Iterator Pattern without sharing Aggregates among processes. The actors of this Design are:

- Protocols:
  - iterationProt: this protocol is concerned with all operations that are related to the Iterator. it supports in this case the following operations:
    1. add: a signal to declare that adding an item is to follow.
    2. next: increments the index by one to move to the following item.
    3. hasNext: a signal to indicate that the following message is to be sent to the client indicating whether we have reached or not the end of the aggregate.
    4. hasNextBool: returns to the client a boolean value; the value is true if the iterator did not reach the end of the aggregate, and false if it did.
  - txtAggregateProt: this protocol supports messages related to storing data of type Text into the aggregate. The following are the elements of the protocol:

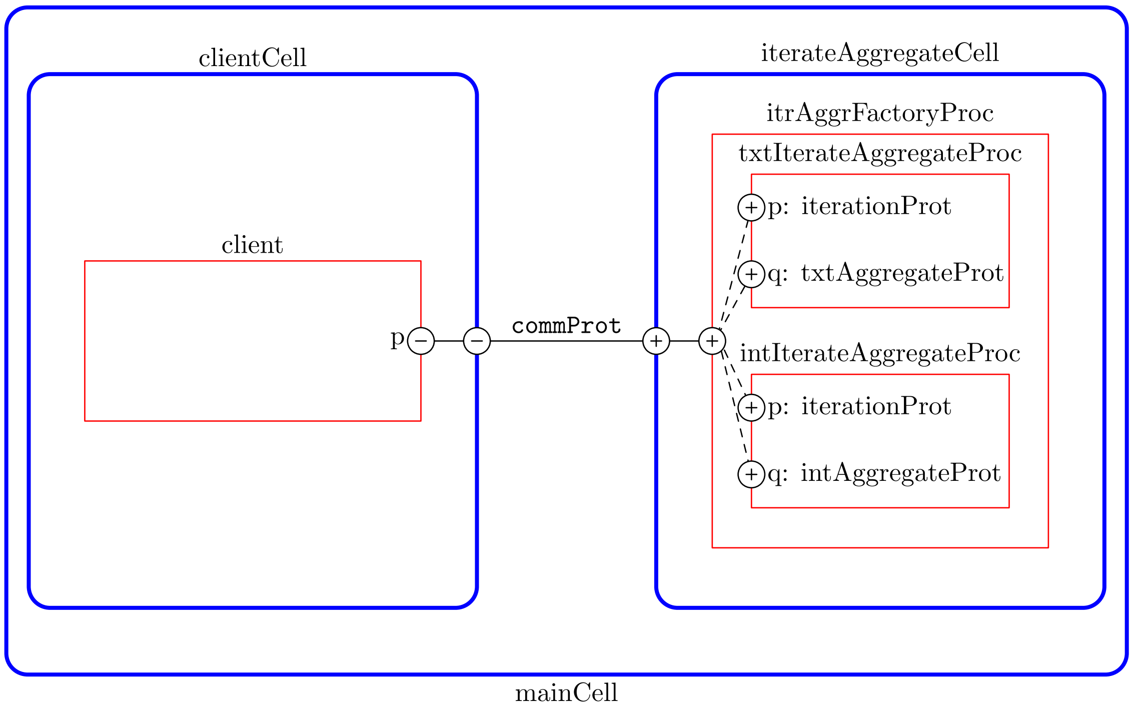

Figure 27: Iterator Pattern without sharing Aggregate in Erasmus

1. addTxt: is the item message, that carries the information to be stored in the aggregate.
2. getNextTxt: is the item that is currently pointed at during iteration. This message sends back the item to the client, when requested.
3. error: this signal is sent if the iterator reached an index that is beyond the size of the aggregate.

- intAggregateProt: this protocol supports messages related to storing data of type Integer into the aggregate. The following are the elements of the protocol:

  1. addInt: is the item message of Integer type, that carries the information to be stored in the aggregate.
  2. getNextInt: is the item that is currently pointed at during iteration. This message sends back the item, which is of type Integer, to the client when requested.
  3. error: this signal is sent if the iterator reached an index that is beyond the size of the aggregate.

- commProt: is a channel that passes through ports. The channel allows the following ports and signal to pass through:

  1. txtSignal: this signal is sent to announce that the ports "iterationProt" and "txtAggregateProt" are sent next respectively.
  2. txtItr: is a port that is of type "iterationProt".
  3. txtAgg: is a port that is of type "txtAggregateProt".
  4. intSignal: this signal is sent to announce that the ports "iterationProt" and "intAggregateProt" are sent next respectively.
  5. intItr: is a port that is of type "iterationProt".
  6. intAgg: is a port that is of type "intAggregateProt".
  7. stop: a signal to stop looping and that allows exiting the process.

• Processes:

  - client: this process creates the data to be stored to all aggregate iterators it needs, and initiates requests to the "itrAggrFactoryProc" process that instantiates the corresponding aggregate iterators.
  - txtIterateAggregateProc: encapsulates both behaviors: the aggregate data keeping and iterating through that aggregate. The aggregate supports data of type Text.

  - intIterateAggregateProc: encapsulates both behaviors: the aggregate data keeping and iterating through that aggregate. The aggregate supports data of type Integer.
  - itrAggrFactoryProc: this process follows the Factory Pattern (the Factory Pattern is discussed in Chapter 4) in the sense it takes the responsibility of instantiating the right process in runtime. This process instantiates the "txtIterateAggregateProc" and "intIterateAggregateProc" processes.

- Cells:
  - clientCell: instantiates the "client" process.
  - iterateAggregateCell: instantiates the "itrAggrFactoryProc". This cell encapsulates the aggregate iterators and the factory processes representing the Iterator Pattern in Erasmus.
  - mainCell: instantiates both cells "clientCell" and "iterateAggregateCell", connects them, and runs the program.

10.2.1.2 Consequences

The Pattern demonstrated in Figure 27 is an External Iterator Pattern because it is controlled by the client, i.e., it passes back the control to the client after every iteration.

This Pattern is extendable: it is possible to introduce new types of aggregate iterators that may add more functionalities other than just iterating through the aggregate incrementally. The iterator may, e.g., iterate through only odd or even numbers, or it could iterate following a defined algorithm.

This design also overcomes the missing Generic feature in Erasmus (see Section 3.7.4), by allowing the creation of aggregate iterators that walk through items of different data types without refactoring or changing the original design. Let us assume that we need to introduce an aggregate iterator that stores and handles data of Boolean type. The Pattern makes it possible by allowing the creation of a new protocol "boolAggregateProt" and by allowing the addition of the new protocol to the communication channel "commProt". An aggregate iterator Process "boolIterateAggregateProc" is to be created and an entry to this process in the Factory Process "itrAggrFactoryProc" is to be added.

10.2.1.3 Implementation

Since this Pattern merges both functionalities in one process: storing data in an aggregate and iterating through the aggregate, the pattern is designed to ensure coherence and encapsulation by separating the messages into two groups of protocols: protocols that deal with populating the aggregate ("aggregateProt", "txtAggregateProt", or "intAggregateProt") and protocols that deal with iterating the aggregate (in this demonstrative program there is only one: "iterationProt").

The protocol that is related to the aggregate depends on the data type that is stored in the aggregate itself. The items' data types that could be stored could either be of primitive data types or even protocols. On the other hand, the protocol that is related to the iterator does not change that often, because it represents the interface of the supported operations in the "Iterator" entity, such as moving to the next item ("next"), checking if the iterator did not reach the end of the aggregate ("hasNext"), and checking whether the iterator has reached the end of the aggregate or not.

The design of the program that is demonstrated in Figure 26 is prone to become unmaintainable if extended. Let us assume that we add more than the two client-to-iterateAggregate pairs that are already in the program ("intClient"-to-"intIterateAggregateProc" and "txtClient"-to-"txtIterateAggregateProc"). For each added pair of processes client-to-iterateAggregate, two new ports have to be created in both cells: "clientCell" and "iterateAggregateCell". At some point, the code becomes difficult to maintain and upgrade.

A possible solution to the program in Figure 26 could be by encapsulating each pair of processes client-to-iterateAggregate in one cell (that could be named as clientIterateAggregateCell). This way, the two needed ports in the previous design are discarded here.

On the other hand, the Pattern demonstrated in Figure 27 is extendable because of the "commProt" channel that makes it possible to create ports dynamically at the client side. Also, the "itrAggrFactoryProc" process makes it possible to create processes dynamically when requested by the client. Finally, adding new iterateAggregate processes is made easy by plugging it to the code without having to refactor the original code (other parts that are not related, e.g., other already existing iterateAggregate processes).

10.2.1.4 Code

The protocols and communication channel that are used to demonstrate the design are:

```
iterationProt = protocol
  *(add |next | hasNext; ↑hasNextBool: Bool)
end
...
```

```
txtAggregateProt = protocol
  *(addTxt: Text | ↑getNextTxt: Text | error);
  stop
end

intAggregateProt = protocol
  *(addInt: Integer | ↑getNextInt: Integer | error);
  stop
end

commProt = protocol
  *(txtSignal; txtItr: iterationProt; txtAgg: txtAggregateProt|
  intSignal; intItr: iterationProt; intAgg: intAggregateProt);
  stop
end
```

An example of an aggregate iterator is the one that handles items of data type Text:

```
 ...
txtIterateAggregateProc = process p: +iterationProt; q: +txtAggregateProt|
  map: MapTxt;
  addPosition: Integer := 0;
  loopPosition: Integer := 0;
  loopselect
    || p.add; map[addPosition] := q.addTxt; addPosition += 1;
    || p.next;
      if loopPosition ≤ addPosition
        then
          q.getNextTxt := map[loopPosition]; loopPosition += 1;
        else
          q.error
      end
    || p.hasNext;
            p.hasNextBool := loopPosition < addPosition;
    || q.stop; exit;
  end
end
 ...
```

The Factory Process looks like:

```
itrAggrFactoryProc = process p: +commProt|
  loopselect
    || p.txtSignal; txtIterateAggregateProc(p.txtItr, p.txtAgg);
        || p.intSignal; intIterateAggregateProc(p.intItr, p.intAgg);
        || p.stop; exit;
  end
end
```

The main cell that instantiates and connects the client cell to the aggregate iterator cell looks like:

```
mainCell = cell
```

```
  p: commProt;
  clientCell (p);
  iterateAggregateCell(p);
end
```

10.2.2 Iterator Pattern in Erasmus by sharing Aggregates

The principle idea of this design is to encapsulate concrete instances of iterators in a cell. These iterators share an aggregate that is also encapsulated in the cell.

Sharing the aggregate among the iterator processes increases the decoupling. But for some types of applications, this design is needed. In cases in which an aggregate is needed to be traversed in different ways or using certain algorithms, this design meets the requirements.

In this Section also, the discussion is incremental. The Design Pattern is demonstrated in the last version of the program.

10.2.2.1 Structure and Actors

Figure 28 demonstrated the simplified version of the Design. The involved actors are:

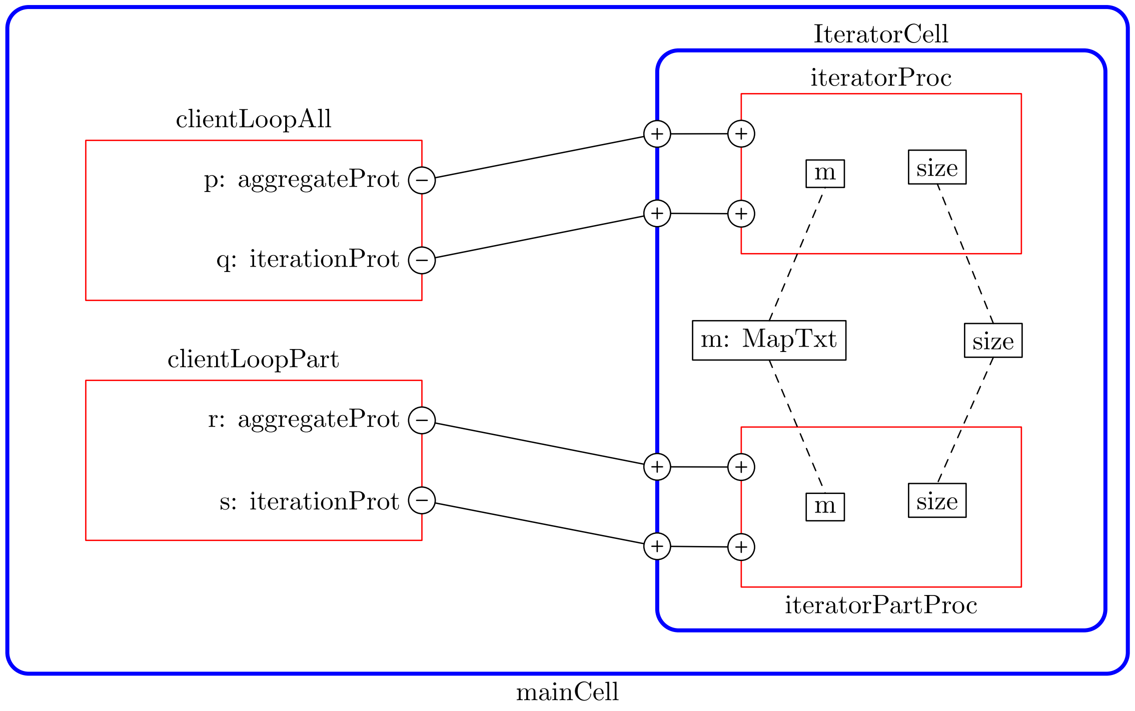

Figure 28: Simplified Design of the Iterator by sharing Aggregates in Erasmus

- Protocols
  - iterationProt: carries the signals and data that are related to traversing an aggregate. This protocol is extendable according to the needs or functionalities of iteration.
  - aggregateProt: is a protocol that deals with the data stored in the aggregate. In this example, the protocol consists of a variable, "val", of type Text. This means that the data to be stored in the aggregate is of type Text. This protocol is also extendable to support other data types.
- Processes
  - iteratorProc: is a concrete process of an iterator, that loops through the whole aggregate incrementally.
  - iteratorPartProc: is another concrete iterating process; this one loops through even indexes in the aggregate.
  - clientLoopAll: is a client process that sends messages to the process "iteratorProc".
  - clientLoopPart: is a client process that sends messages to the process "iteratorPartProc".
- Cells
  - IteratorCell: this cell encapsulates the different iterating processes and the shared aggregate the iterating processes traverse.
  - mainCell: instantiates the processes "clientLoopAll" and "clientLoopPart", connects them to the instance of the "IteratorCell"cell, and finally runs the program.

As we will see in Section 10.2.2.3, extending this program makes it inflexible and handling or managing it becomes difficult. The following design is more general for this type of iterators in which iterators share the same aggregate. Figure 29 demonstrates the Design Pattern, and the actors involved in building the Pattern are explained here:

- Protocols
  - iteratorProt: for simplicity, the signals and data that are related to the iterator, and the signals and data that are related to the aggregate are merged in this protocol.

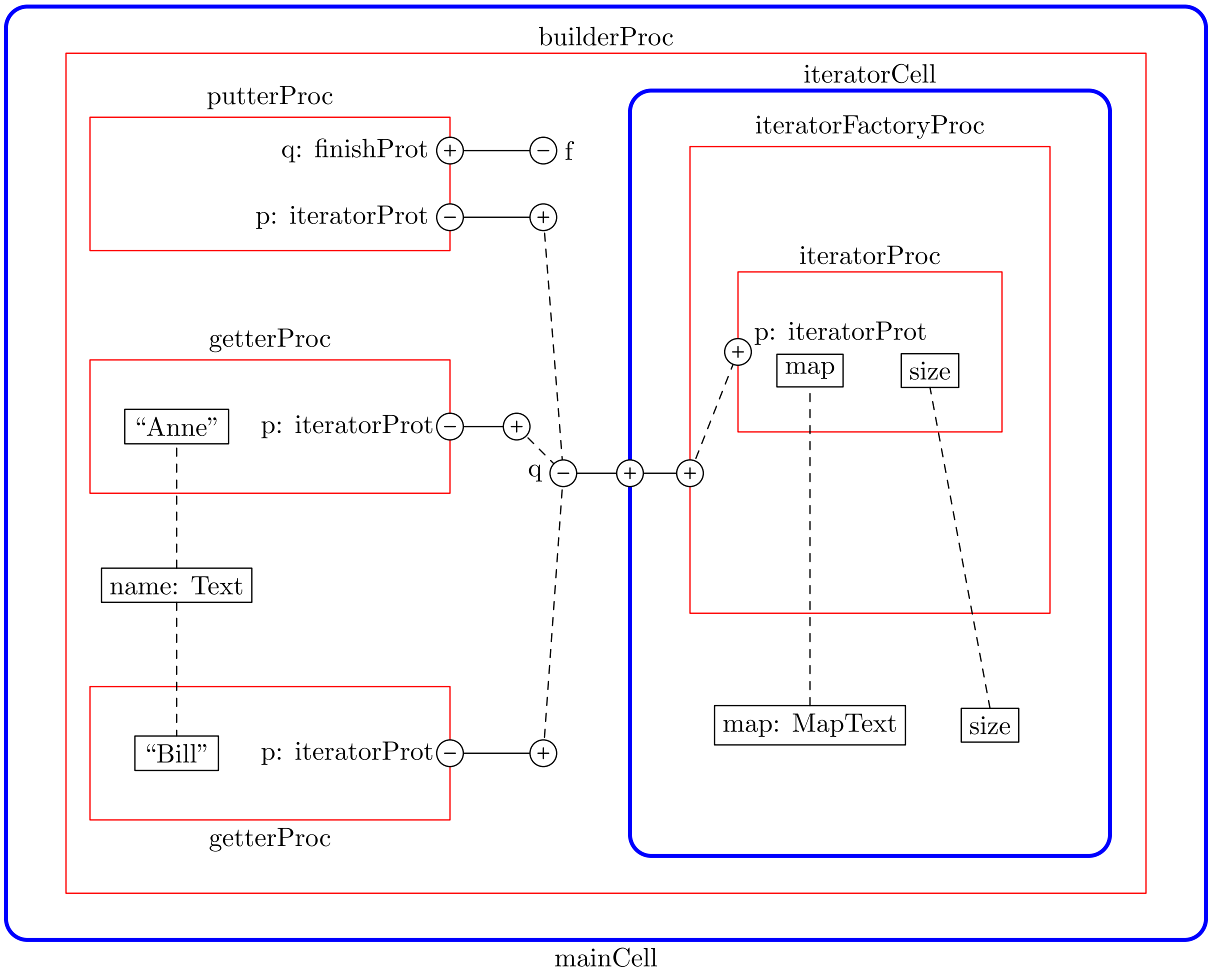

Figure 29: Iterator Design Pattern in Erasmus

- MakeIterProt: is a communication channel that carries ports of type "iteratorProt". It also sends a signal "stop", to indicate the end of sending ports.
- finishProt: contains a signal "finished" that indicates the end of the storing data into the aggregate.

- Processes
  - builderProc: instantiates the processes that are the clients of the iterator and that create data and store them into the aggregate. It also instantiates the "iteratorCell" cell, and connects the processes and the cell through channels.
  - putterProc: is a client process that creates data and sends them to the aggregate as messages to be stored.
  - getterProc: gets back the result of the iteration through the aggregate. In this example, this process is instantiated twice with different variable names as demonstrated in Figure 29.
  - iteratorFactoryProc: instantiates iterator processes when requested.
  - iteratorProc: is the concrete instance of the iterator.

- Cells
  - IteratorCell: this cell encapsulates the iterator(s), the Iterator Factory producer (for more information about the Factory Pattern go to Chapter 4), and the shared aggregate in it.
  - mainCell: instantiates the "iteratorCell" cell, the "putterProc" process, and two instances of the "getterProc" process, it connects all these processes and cell through channels, and it runs the program.

10.2.2.2 Consequences

The Iterator Design Pattern demonstrated in Figure 29 is extendable in the sense it could support different data types to be stored in an aggregate. For this we need to create an aggregate that supports storing, for example, data of type Integers. The protocol is extendable, by changing the original version of the "iteratorProt" protocol to:

```
iteratorProt = protocol
  *(
    add: Text | addI: Integer|
    reset |
    ( ↑hasNext: Bool ) |
    ( next; (( ↑valTxt: Text | error ) |  ( ↑valInt: Integer | error )))
  ); stop
end
```

The Iterator Factory "iteratorFactoryProc" process can also support more Iterator processes, but for this we need to indicate in the code which Iterator process is requested to be instantiated.

A way of representing the extension could be:

```
MakeIterProt = protocol
  *(cond:Integer; chan: iteratorProt );
  ↑stop
end

iteratorFactoryProc = process q: +MakeIterProt; alias map: MapText; alias size: Integer|
  loopselect
    || cond: Integer := q.cond;
       cases
         |cond = 1| iteratorProc(q.chan, map, size);
         |cond = 2| iteratorTwoProc(q.chan, map, size);
         |cond = 3| iteratorThreeProc(q.chan, map, size);
       end;
    || q.stop; exit
  end
end
```

10.2.2.3 Implementation

The first version that is demonstrated in Figure 28 is possible to extend, but difficult to handle and to maintain. For each pair client-iterator processes that is created or added, another two ports are needed to be created (ports of protocol type "aggregateProt" and "iterationProt"). Managing and maintaining so many ports becomes confusing once the number of client-iterator pairs grows. Additionally, the creation of iterator processes is not dynamic, this part adds to the restriction of the design.

However, the Design Pattern that is demonstrated in Figure 29 is more flexible, because it basically allows the client to create, under request, the iterator dynamically. This is done through the "iteratorFactoryProc" that does the dynamic creation of the Iterator process.

10.2.2.4 Code

The Iterator protocols merge, as mentioned before, the signal and data of both, the iterator and the aggregate.

```
iteratorProt = protocol
  *(
    add: Text |
    reset |
    ( ↑hasNext: Bool ) |
    ( next; (( ↑val: Text | error ) | ( ↑valInt: Integer | error )))
  ); stop
end
...
```

The Factory Process “iteratorFactoryProc” shares the aggregate with its parent cell (this is indicated by the keyword “alias”). It instantiates the “iteratorProc” process or quits as required in the message of protocol type “MakeIterProt”. The “alias” keyword allows the processes to change the content of the aggregate.

```
...
iteratorFactoryProc = process
  q: +MakeIterProt;
  alias map: MapText;
  alias size: Integer
|
  loopselect
  || iteratorProc(q.chan, map, size);
  || q.stop; exit
  end
end
...
```

The iterator process, also shares the aggregate with its parent cell “iteratorCell”, and its parent process “iteratorFactoryProc”.

```
...
iteratorProc = process
  p: +iteratorProt;
  alias map: MapText;
  alias size: Integer
|
  index: Integer := 0;
  loopselect
  || map[size] := p.add; size += 1
  || p.reset; index := 0
  || p.hasNext := exists map[index]
  || p.next;
     if exists map[index]
       then p.val := map[index]; index += 1
       else p.error
     end
  || p.stop; exit
  end
end
...
```

10.3 Summary

In this chapter we have shown how to implement the Iterator Pattern in Erasmus. The Erasmus implementation differs from the OOP implementation in that the design of the Pattern in Erasmus can only be dynamic if incorporated with the Factory Design Pattern. Also, the point that is different in both paradigms is the interface: OOP implementation emphasizes on programming for an interface and not for an implementation. In Erasmus, on the other hand, there is no concept of process as an interface, but the parameters of a process are the interface of the process. As a result, parameters have to be well thought of during the design phase. Finally, the implementation of Erasmus seems at first to have high coupling between the “Iterator” and the “Aggregate“ entities in comparison to the OOP implementation. However, the coupling between these two entities is also high in OOP because a concrete “Aggregate” object is passed by reference to the “ConcreteIterator” object, the thing that allows the iterator and the aggregate itself to change the contents of the aggregate if not restricted in the implementation. The Erasmus version has a disadvantage over the OOP. In the Erasmus implementation, for each representation of a data type that is stored in the aggregate, an iterator or an aggregate-iterator process must be created. This affects the flexibility and maintainability of the code.

Chapter 11

Conclusion and Future Work

11.1 Conclusion

Since design patterns proved to be very important to solve common and recurring problems in OOP, we demonstrate in this document that these problems also arise in Erasmus, the emerging programming language. We discuss these patterns in terms of the current version of Erasmus, µE; we also introduce our designs to these patterns following the guidelines of elements of documentation that appear in the book [GHJV95].

We have developed some design patterns that were inspired from some design patterns that appear in the book [GHJV95]. These patterns are: *Factory Pattern*, *Protocol Adapter Pattern*, *Process Adapter Pattern*, *Cell Level Composition*, *Protocol Level Composition*, *Process Level Composition*, *Chain of Responsibility Pattern*, *Command Pattern*, *Interpreter Pattern*, *Iterator Pattern without sharing Aggregates*, and *Iterator Pattern by sharing Aggregates*.

Each pattern is demonstrated with diagrams, discusses the possibilities of extension, and its limitations. Some patterns are composed of other patterns in their implementation. The patterns were chosen carefully to represent the different categories that appear in the [GHJV95] and we tried to preserve the essential concept of the original design in our own design. This is done to prove that our conclusion we draw about Erasmus is a fair statement.

The conclusion that appears in the article [BLR96] that patterns analysis indicates the importance of some constructs and mechanisms that should be provided by new object-oriented languages, can be generalized to the important constructs and mechanisms supported by process-oriented languages, precisely in this case study Erasmus. With this, we can conclude that by being able to express patterns in Erasmus we prove that Erasmus as a language has at least some validity.

This also demonstrates and proves that POP like OOP is a viable paradigm, and in fact the implementations of some design patterns demonstrate that in some cases, the abstraction of POP itself supports these design patterns, especially the ones that belong to the *Behavioral* category.

The shift towards concurrent programming and computing that was driven by multicore processors, networks and distributed architectures encouraged a paradigm shift that will at some point replace OOP. One of these paradigms is the process-oriented programming (POP). The basic element of this paradigm is the process. Processes communicate with each other by exchanging messages through ports. This concept if compared with OOP reduces coupling. Unlike OOP where an object has two interfaces, the one it provides services from to other objects and another one from which it gets services, a process in POP has only one type of interface that is composed of its ports. Hence, the dependencies of a process in POP are bounded by its ports, while in OOP a dependency of an object is unbounded: they are the transitive closure of the object's calls [GS07].

Coming from an OOP programming background, working with POP is not very different apart from the part that is concerned with message exchange. Message exchange requires a good knowledge of the direction of the flow of data. At the beginning, the decision of how and what types of ports to design may seem difficult, but once the concept and the rules are learned, things become smoother and easier. However, incautious design of message exchange in POP may lead to deadlocks.

Designing some patterns in POP was more difficult than its counterpart in OOP, e.g., the *Interpreter* and the *Iterator by sharing Aggregation* patterns. However, in other cases designing a pattern was easy and appeared intuitive such as the *Command* and *Chain of Responsibility* patterns. Since µE, the current version of Erasmus, is still evolving, there are some limitations in implementing some design patterns.

11.2 Future Work

This is just the start, otherwise, there is a lot of work that is to be done in building new design patterns for Erasmus. First there are many other patterns that are defined in [GHJV95] and that are not yet explored in this thesis. Also, there are other patterns related to concurrent programming that are defined and explored in [Lea99] and [San97], and others for parallel programming such as the ones that appear in [MSM04]. Add to that once well defined, Erasmus will have for sure its own design patterns that could be deduced from the language itself.

Finally, once the language is stable and most of its features are already developed, apart from the development of patterns, there is some work related to paradigm performance comparison and readability comparison to be done in relation to other paradigms.

Bibliography

[608a] Java Platform Standard Ed. 6. Enumeration, 2008. `http://java.sun.com/javase/6/docs/api/`.

[608b] Java Platform Standard Ed. 6. Iterator, 2008. `http://java.sun.com/javase/6/docs/api/`.

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, 1977.

[BC87] Kent Beck and Ward Cunningham. Using pattern languages for object-oriented programs. Technical Report No. CR-87-43, Submitted to the OOPSLA-87, September 1987.

[BLR96] Gerald Baumgartner, Konstantin Laufer, and Vincent F. Russo. On the interaction of object-oriented design patterns and programming languages. Technical Report CSD-TR-96-020, 1996.

[Coo00] James W. Cooper. *Java Design Patterns: A Tutorial.* Addison-Wesley, 2000.

[Dij65] E.W. Dijkstra. Cooperating sequential processes. Technical Report Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, 1965. Reprinted in [Gen68].

[FFBS04] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns.* O'Reilly Media, Inc., 2004.

[Gea03] David Geary. Adopt adapter understand how adapters let disparate systems work together, 2003. `http://www.javaworld.com/javaworld/jw-09-2003/jw-0926-designpatterns.html?page=2`.

[Gen68] F. Genuys, editor. *Programming Languages (NATO Advanced Study Institute).* Academic Press, 1968.

[GHJO09] Erich Gamma, Richard Helm, Ralph Johnson, and Larry O'Brien. Design Patterns 15 Years Later: An Interview with Erich Gamma, Richard Helm, and Ralph Johnson, October 2009. `http://www.informit.com/articles/article.aspx?p=1404056`.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GS07] Peter Grogono and Brian Shearing. Modular concurrency: a new approach to manageable software. Technical Report TR E-02, Department of Computer Science and Software Engineering, Concordia University, April 2007.

[GS08] Peter Grogono and Brian Shearing. MEC Tests. Technical Report TR E-07, Department of Computer Science and Software Engineering, Concordia University, February 2008.

[GS09] Peter Grogono and Brian Shearing. MEC Reference Manual. Technical Report TR E-06, Department of Computer Science and Software Engineering, Concordia University, May 2009.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[Hol06] Steve Holzner. *Design Patterns For Dummies*. Wiley Publishing, Inc, 2006.

[KG08] Foutse Khomh and Yann-Gael Gueheneuce. Do design patterns impact software quality positively? In *CSMR '08: Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, pages 274–278, Washington, DC, USA, 2008. IEEE Computer Society.

[Lea99] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Prentice Hall, 1999.

[Lee06] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, May 2006.

[Mag89] Boris Magnusson. Process oriented programming. *SIGPLAN Not.*, 24(4):34–36, 1989.

[Met02] Steven J. Metsker. *Design Patterns Java Workbook*. Addison-Wesley, 2002.

[MSM04] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming.* Addison-Wesley, 2004.

[San97] Bo I. Sandén. Concurrent design patterns for resource sharing. In *TRI-Ada '97: Proceedings of the conference on TRI-Ada '97*, pages 173–183, New York, NY, USA, 1997. ACM.

[Sin01] Tony Sintes. Speaking on the observer pattern: How can you use the observer pattern in your java design?, 2001. `http://www.javaworld.com/javaworld/javaqa/2001-05/04-qa-0525-observer.html`.

[SMC74] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005.

[Tut08] The Java Tutorials. The collection interface, 2008. `http://java.sun.com/docs/books/tutorial/collections/interfaces/collection.html`.

[Zen92] Steven Ericsson Zenith. *Process Interaction Models, Université Pierre et Marie Curie - PARIS VI.* PhD thesis, Ecole Nationale Supérieure des Mines de Paris, FRANCE, 1992.

Appendix A

Factory Design Pattern in Erasmus

The complete code of the demonstrative program of the Factory Pattern in Erasmus and the output of the execution are listed here:

A.1 Code

```
CR: Text = "\n";

txtProt = protocol
  *(txt: Text);
  stop
end

intProt = protocol
  *(num: Integer);
  stop
end

boolProt = protocol
  *(val: Boolean);
  stop
end

factoryProt = protocol
  *(val1: txtProt| val2: intProt| val3: boolProt);
  stop
end

product1Proc = process p: +txtProt|
  conct: Text := "";
  loopselect
    || temp: Text := p.txt; conct += temp;
    || p.stop; exit;
  end;
  sys.out := "Server One: " // conct // CR;
```

```
end

product2Proc = process p: +intProt|
  sum: Integer := 0;
  loopselect
    || temp: Integer := p.num; sum += temp;
    || p.stop; exit;
  end
  sys.out := "Server Two sum: " // sum // CR;
end

product3Proc = process p: +boolProt|
  loopselect
    || temp: Boolean := p.val;
      sys.out := "This is " // temp // CR;
    || p.stop; exit;
  end;
end

factoryProc = process p: +factoryProt |
  loopselect
    || product1Proc(p.val1);
    || product2Proc(p.val2);
    || product3Proc(p.val3);
    || p.stop; exit;
  end
end

clientProc = process p: −factoryProt|
  q1: −txtProt;
  p.val1 := q1;
  q1.txt := "Hello!";
  q1.txt := " How are you?";
  q1.stop;

  q2: −intProt;
  p.val2 := q2;
  q2.num := 100;
  q2.num := 210;
  q2.num := 400;
  q2.stop;

  q3: −boolProt;
  p.val3 := q3;
  q3.val := true;
  q3.val := false;
  q3.stop;

  p.stop;
end

mainCell = cell
```

```
  p: factoryProt;
  clientProc(p);
  factoryProc(p);
end

mainCell()
```

A.2 Results

Running the Factory Design program produces the following output:

```
product One: Hello! How are you?
product Two sum: 710
This is true
This is false
```

Appendix B

Adapter Design Pattern in Erasmus

B.1 Protocol Adapter Pattern

B.1.1 Code

The following is the complete code of the implementation of the Protocol Adapter Pattern in Erasmus I have developed:

```
lineProt = protocol
  point: Integer
end;

triangleProt = protocol
  line1: lineProt;
  line2: lineProt;
  line3: lineProt;
  stop
end;

targetProt = protocol
  line1: lineProt;
  line2: lineProt
end;

serverCompositeProc = process p: +triangleProt |
  loopselect
    || line1: +lineProt := p.line1;
      sys.out := "Line1: point: " + text line1.point + "\n";
    || line2: +lineProt := p.line2;
      sys.out := "Line2: point: " + text line2.point + "\n";
    || line3: +lineProt := p.line3;
      sys.out := "Line3: point: " + text line3.point + "\n";
    || $p.stop$; **exit**;
  **end**;
**end**;

$adapterProc$ = **process** $p$: $+targetProt$ |
  $q$: $-triangleProt$;
  $serverCompositeProc(q)$;
  $q.line1$ := $p.line1$;

  $q.line2$ := $p.line2$;
  $q.stop$;
**end**

$clientProc$ = **process** $p$: $-targetProt$|
  $q1$: $-lineProt$;
  $p.line1$ := $q1$;
  $q1.point$ := 1;

  $q2$: $-lineProt$;
  $p.line2$ := $q2$;
  $q2.point$ := 2;

**end**;

$ctl$ = **cell**
  $p$: $targetProt$;
  $clientProc(p)$;
  $adapterProc(p)$;
**end**

$ctl()$
```

B.1.2 Results

Running the program produces the following output:

```
Line1: point: 1
Line2: point: 2
```

B.2 Process Adapter Pattern

This Section includes two programs, one that implements a post-fix grammar, and the other one that implements a tree-based grammar.

B.2.1 Code

This program implements post-fix grammar. The process that provides the post-fix grammar functionality is the “evaluateProc” process.

```
States = < commence, number, name, op, stop >
tokProt = protocol *( ↑kind: States; ↑value: Text ) end

tokValProt = protocol *( kind: States; value: Text ); stop end

expressionProt = protocol
  *( number: Integer| minus| plus| multiplication );
  stop
end

commProt = protocol
  prot: expressionProt
end

scannerProc = process expr: Text; out: +tokProt |
  sys.out := 'Scanning: ' + expr + '\n';
  expr += '$';
  ix: Integer := 0;
  buffer: Text := '';
  state: States := commence;
  loop
    ch: Text := expr[ix];
    isDigit: Bool := '0' ≤ ch and ch ≤ '9';
    isLetter: Bool := 'a' ≤ ch and ch ≤ 'z';
    cases

      |state = commence and isDigit|
        buffer := ch;
        state := number;
        ix += 1

      |state = commence and isLetter|
        buffer := ch;
        state := name;
        ix += 1

      |state = commence and ch = '$'|
        out.kind := stop;
        out.value := '';
        exit

      |state = commence|
        out.kind := op;
        out.value := ch;
        ix += 1

      |state = number and isDigit|
        buffer += ch;
        ix += 1
      |state = number|
        out.kind := number;
        out.value := buffer;
        state := commence

      |state = name and (isLetter or isDigit)|
        buffer += ch;
        ix += 1
      |state = name|
        out.kind := name;
        out.value := buffer;
        state := commence

      ||
        sys.err := 'Scanner error!\n';
        exit
    end
  end
end;

evaluateProc = process p: +commProt|
  q: +expressionProt := p.prot;
  stack: Integer indexes Integer;
  sum: Integer := 0;
  counter: Integer := −1;
  loopselect
      || number: Integer := q.number; counter +=1; stack[counter]:= number;
        sys.out := "Number Received: " + text number + "\t counter: "
        + text counter + "\n";
      || q.minus; sys.out := "Minus Sign\n";
        sum −= stack[counter];
        sys.out := "Counter Value: " + text counter + "\t stack"
        + text stack[counter] + "\n";
        counter −= 1; sum += stack[counter];
        sys.out := "Counter Value: " + text counter + "\t stack"
        + text stack[counter] + "\n";
        stack[counter] := sum;
        sys.out := "Counter Value: " + text counter + "\t stack"
        + text stack[counter] + "\n";
        sum := 0;

      || q.plus; sys.out := "Plus Sign\n";
        sum += stack[counter];
        sys.out := "Counter Value: " + text counter + "\t stack"
        + text stack[counter] + "\n";
        counter −= 1; sum += stack[counter];
        sys.out := "Counter Value: " + text counter + "\t stack"
        + text stack[counter] + "\n";
        stack[counter] := sum;
        sys.out := "Counter Value: " + text counter + "\t stack"
        + text stack[counter] + "\n";
        sum := 0;
    || q.stop;  sys.out := "Final Sum: " + text stack[0] + "\n"; exit;
  end
end

expressionBuilderProc = process inp: −tokProt; out: −tokValProt |
  sn: States indexes Text;
  sn[commence] := 'commence';
  sn[number] := 'number';
  sn[name] := 'name';
  sn[op] := 'op';
  sn[stop] := 'stop';
  loop
    kind: States := inp.kind;
    out.kind := kind;
    out.value := inp.value;
    until kind = stop;
  end
  out.stop;
end;


converterProc = process pt: +tokValProt|
  p: −commProt;
  evaluateProc(p);
  q: −expressionProt;
  p.prot := q;

  loopselect
    || state: States := pt.kind;
       val: Text := pt.value;
       cases
         | state = number|
           q.number := int val;
         | state = op and val = "+"|
           q.plus;
         | state = op and val = "−"|
           q.minus;
         | state = op and val = "*"|
           q. multiplication
       end;
    || pt.stop; exit;
  end;
  q.stop;
end

aCell = cell
  p: tokProt;
  q: tokValProt;
  scannerProc('42 4 2 − +', p);
  converterProc(q);
  expressionBuilderProc(p, q);
end;
```

```
aCell();
```

B.2.2 Results

Running the program produces the following output:

```
Scanning: 42 4 2 - +
Number Received: 42      counter: 0
Number Received: 4       counter: 1
Number Received: 2       counter: 2
Minus Sign
Counter Value: 2         stack2
Counter Value: 1         stack4
Counter Value: 1         stack2
Plus Sign
Counter Value: 1         stack2
Counter Value: 0         stack42
Counter Value: 0         stack44
Final Sum: 44
```

B.2.3 Code

This program implements tree-based grammar. The process that provides the tree-based grammar functionality is the "evaluateProc" process.

```
States = < commence, number, name, op, stop >
tokProt = protocol *( ↑kind: States; ↑value: Text ) end

tokValProt = protocol *( kind: States; value: Text ); stop end

expressionProt = protocol
  *( number: Integer| minus| plus| multiplication );
  stop
end

commProt = protocol
  prot: expressionProt
end

scannerProc = process expr: Text; out: +tokProt |
  sys.out := 'Scanning: ' + expr + '\n';
  expr += '$';
  ix: Integer := 0;
  buffer: Text := '';
  state: States := commence;
  loop
    ch: Text := expr[ix];
    isDigit: Bool := '0' ≤ ch and ch ≤ '9';
    isLetter: Bool := 'a' ≤ ch and ch ≤ 'z';
    cases

      |state = commence and isDigit|
        buffer := ch;
        state := number;
        ix += 1

      |state = commence and isLetter|
        buffer := ch;
        state := name;
        ix += 1

      |state = commence and ch = '$'|
        out.kind := stop;
        out.value := '';
        exit

      |state = commence|
        out.kind := op;
        out.value := ch;
        ix += 1

      |state = number and isDigit|
        buffer += ch;
        ix += 1
      |state = number|
        out.kind := number;
        out.value := buffer;
        state := commence

      |state = name and (isLetter or isDigit)|
        buffer += ch;
        ix += 1
      |state = name|
        out.kind := name;
        out.value := buffer;
        state := commence

      ||
        sys.err := 'Scanner error!\n';
        exit
    end
  end
end;

evaluateProc = process p: +commProt|
  q: +expressionProt := p.prot;
  stack: Integer indexes Integer;
  sum: Integer := 0;
  counter: Integer := −1;
  loopselect
    || number: Integer := q.number; counter +=1;
      stack[counter]:= number;
      sys.out := "Number Received: " + text number + "\t counter: "
      + text counter + "\n";
    || q.minus; sys.out := "Minus Sign\n";
      sum := stack[counter];
      sys.out :=  "Counter Value: " + text counter + "\t stack"
      + text stack[counter] + "\n";
      number:Integer := q.number;
      sum −= number;
      stack[counter] := sum;
      sys.out :=  "Counter Value: " + text counter
      + "\t stack" + text stack[counter] + "\n";
      sum := 0;

    || q.plus;  sys.out := "Plus Sign\n";
      sum := stack[counter];
      sys.out :=  "Counter Value: " + text counter + "\t stack"
      + text stack[counter] + "\n";
      number:Integer := q.number;
      sum += number;
      stack[counter] := sum;
      sys.out :=  "Counter Value: " + text counter + "\t stack"
      + text stack[counter] +  "\n";
      sum := 0;
    || q.multiplication;  sys.out := "Multiplication  Sign\n";
      sum := stack[counter];
      sys.out :=  "Counter Value: " + text counter + "\t stack"
      + text stack[counter] + "\n";
      number:Integer := q.number;
      sum ∗= number;
      stack[counter] := sum;
      sys.out :=  "Counter Value: " + text counter + "\t stack"
      + text stack[counter] +  "\n";
      sum := 0;
    || q.stop;  sys.out := "Result: "  + text stack[0]  + "\n"; exit;
  end
end

expressionBuilderProc = process inp: −tokProt; out: −tokValProt |
  sn: States indexes Text;
  sn[commence] := 'commence';
  sn[number] := 'number';
  sn[name] := 'name';
  sn[op] := 'op';
  sn[stop] := 'stop';
  loop
    kind: States := inp.kind;
    out.kind := kind;
    out.value := inp.value;
    until kind = stop;
  end
  out.stop;
end;


converterProc = process pt: +tokValProt|
  p: −commProt;
  evaluateProc(p);
  q: −expressionProt;
  p.prot := q;

  loopselect
    || state: States := pt.kind;
      val: Text := pt.value;
      cases
        |state = number|
          q.number := int val;
        |state = op and val = "+"|
          q.plus;
        |state = op and val = "−"|
          q.minus;
        |state = op and val = "*"|
          q.multiplication
      end;
    || pt.stop; exit;
  end;
  q.stop;
end

aCell = cell
  p: tokProt;
  q: tokValProt;
  scannerProc('42−4+2', p);
  converterProc(q);
  expressionBuilderProc(p, q);
end;

aCell();
```

B.2.4 Results

Running the program produces the following output:

```
Scanning: 42-4+2

Number Received: 42      counter: 0

Minus Sign

Counter Value: 0         stack42

Counter Value: 0         stack38
```

```
Plus Sign
Counter Value: 0         stack38
Counter Value: 0         stack40
Result: 40
```

Appendix C

Composite Design Pattern in Erasmus

C.1 Cell Level Composition

C.1.1 Code

The complete code that demonstrates the Cell Composition is as follows:

```
namePort = protocol
  fName: Text ; lName: Text
end

ageProt = protocol
  dayOfBirth: Integer;  monthOfBirth: Integer; yearOfBirth: Integer
end

statusProt = protocol
  nationality:  Text
end

personProt = protocol
  *(name; aName: nameProt| age; anAge: ageProt| status; aStatus: statusProt);
  stop
end

nameProc = process p: +nameProt |
  temp: Text;
  temp := "First Name: " + p.fName;
  temp += "\t";
  temp += " Last Name: " + p.lName + "\n";
  sys.out := temp;
end
```

```
nameCell = cell p: +nameProt |
  nameProc(p);
end

ageProc = process curYr: Integer; p: +ageProt |
  temp: Text := "Your are born in: " + text p.dayOfBirth + "/";
  temp += text p.monthOfBirth + "/";

  year: Integer := p.yearOfBirth;
  sys.out :=  temp + text year;

  sys.out := "\tYou are now: " + text (curYr − year);
end

ageCell = cell p: +ageProt|
  curYr: Integer := 2009;
  ageProc(curYr, p)
end

nationalityProc = process p:+statusProt|
  sys.out := "\nYou are: " + p.nationality;
end

nationalityCell  = cell p:+statusProt|
  nationalityProc(p)
end

clientProc  = process p: −personProt|
  sys.out := "First  Name: ";
  fName: Text := sys.inp;

  sys.out := "Last Name: ";
  lName: Text := sys.inp;

  sys.out := "Day of Birth: ";
  dBirth: Integer := int sys.inp;

  sys.out := "Month of Birth: ";
  mBirth: Integer := int sys.inp;

  sys.out := "Year of Birth: ";
  yBirth: Integer := int sys.inp;

  sys.out := "Nationality: ";
  nationality: Text := sys.inp;

  sys.out := "\n\n";

  p1: −nameProt;
  p.name;
  p.aName := p1;
  p1.fName := fName;
  p1.lName := lName;


  p2: −ageProt;
  p.age;
  p.anAge := p2;

  p2.dayOfBirth := dBirth;
  p2.monthOfBirth := mBirth;
  p2.yearOfBirth := yBirth;

  p3: −statusProt;
  p.status;
  p.aStatus := p3;

  p3.nationality := nationality;

  p.stop;

  sys.out := "\n";

  q: −nameProt;
  nameCell(q);
  q.fName := fName;
  q.lName := lName;

  sys.out := "\n";
end

compositeProc = process p: +personProt|
  loopselect
     || p.name; nameCell(p.aName);
     || p.age; ageCell(p.anAge);
     || p.status; nationalityCell(p.aStatus);
     || p.stop; exit;
  end
end


compositeCell = cell p: +personProt|
  compositeProc(p)
end

ctl = cell
  p: personProt
  clientProc(p);
  compositeCell(p);
end

ctl()
```

C.1.2 Results

Running the program produces the following output:

```
First Name: Laura

Last Name: Smith

Day of Birth: 10

Month of Birth: 10

Year of Birth: 1980

Nationality: Canadian



First Name: Laura        Last Name: Smith

Your are born in: 10/10/1980    You are now: 29

You are: Canadian


First Name: Laura        Last Name: Smith
```

C.2 Protocol Level Composition

C.2.1 Code

The complete code that demonstrates the Protocol Composition is as follows:

```
lineProt = protocol
  point: Integer
end;

triangleProt = protocol
  line1: lineProt;
  line2: lineProt;
  line3: lineProt;
  stop
end;

ServerCompositeProc = process p: +triangleProt |
  loopselect
     || line1: +lineProt := p.line1;
       sys.out := "Line1: point: " + text line1.point + "\n";
     || line2: +lineProt := p.line2;
       sys.out := "Line2: point: " + text line2.point + "\n";
     || line3: +lineProt := p.line3;
       sys.out := "Line3: point: " + text line3.point + "\n";
     || p.stop; exit;
  end;
end;

serverLeafProc = process p: +lineProt|
  sys.out := "\nLeaf point: " + text p.point + "\n";
end;

clientProc = process |

  p: −triangleProt;
  ServerCompositeProc(p);
  q1: −lineProt;
  p.line1 := q1;

  q1.point := 1;

  q2: −lineProt;
  p.line2 := q2;
  q2.point := 2;

  q3: −lineProt;
  p.line2 := q3;
  q3.point := 3;

  q4: −lineProt;
  serverLeafProc(q4);
  q4.point := 4;

  p.stop;
end;

ctl = cell
  clientProc();
end

ctl()
```

C.2.2 Results

Running the program results outputs the following:

```
Line1: point: 1
Line2: point: 2
Line2: point: 3


Leaf point: 4
```

C.3 Process Level Composition

C.3.1 Process Composite Pattern Version One

C.3.1.1 Code

The complete code that demonstrates the first version of the Process Composition is as follows:

```
personProt =
protocol
  (fName: Text|
  lName: Text|
  dayOfBirth: Integer|
  monthOfBirth: Integer|
  yearOfBirth: Integer|
  nationality: Text);
  stop
end

commProt = protocol
  channel: personProt
end

serverProc = process p: +personProt |
  loopselect
    || sys.out := "Your First Name is: " + p.fName + "\n";
    || sys.out := "Your Last Name is: " + p.lName + "\n";
    || sys.out := "Your Day of Birth: " + text p.dayOfBirth + "\n";
    || sys.out := "Your Month of Birth: " + text p.monthOfBirth + "\n";
    || sys.out := "Your Year of Birth: " + text p.yearOfBirth + "\n";
    || sys.out := "Your Nationality: " + p.nationality + "\n";
    || p.stop; exit;
  end
end

componentProc = process p: +commProt |
  sys.out := "componentProc is Running\n";
  q: +personProt := p.channel;
  serverProc(q);
end

clientProc = process |
  sys.out := "First Name: ";
  fName: Text := sys.inp;

  sys.out := "Last Name: ";
  lName: Text := sys.inp;

  sys.out := "Day of Birth: ";
  dBirth: Integer := int sys.inp;
  sys.out := "Month of Birth: ";
  mBirth: Integer := int sys.inp;

  sys.out := "Year of Birth: ";
  yBirth: Integer := int sys.inp;

  sys.out := "Nationality: ";
  nationality: Text := sys.inp;

  sys.out := "\n\n";

  p: −commProt;
  componentProc(p);
  q: −personProt;
  p.channel := q;

  q.fName := fName;
  q.lName := lName;
  q.dayOfBirth := dBirth;
  q.monthOfBirth := mBirth;
  q.yearOfBirth := yBirth;
  q.nationality := nationality;
  q.stop;

  sys.out := "\n";

end

ctl = cell
  clientProc();
end

ctl()
```

C.3.1.2 Results

Running the program produces the following output:

```
First Name: Laura
Last Name: Smith
Day of Birth: 10
Month of Birth: 10
Year of Birth: 1980
Nationality: Canadian
```

```
componentProc is Running
Your First Name is: Laura
Your Last Name is: Smith
Your Day of Birth: 10
Your Month of Birth: 10
Your Year of Birth: 1980
Your Nationality: Canadian
```

C.3.2 Process Composite Pattern Version Two

C.3.2.1 Code

The complete code that demonstrates the second version of the Process Composition is as follows:

```
personProt =
protocol
  (fName: Text|
  lName: Text|
  dayOfBirth: Integer|
  monthOfBirth: Integer|
  yearOfBirth: Integer|
  nationality: Text);
  stop
end

commProt = protocol
  channel: personProt
end

server1Proc = process p: +personProt |
  loopselect
    || sys.out := "Your First Name is: " + p.fName + "\n";
    || sys.out := "Your Last Name is: " + p.lName + "\n";
    || sys.out := "Your Nationality: " + p.nationality + "\n";
    || p.stop; exit;
  end
end

server2Proc = process p: +personProt |
  loopselect
    || sys.out := "Your Day of Birth: " + text p.dayOfBirth + "\n";
    || sys.out := "Your Month of Birth: " + text p.monthOfBirth + "\n";
    || sys.out := "Your Year of Birth: " + text p.yearOfBirth + "\n";
    || p.stop; exit;
  end
end

componentProc = process p: +commProt |
  sys.out := "componentProc is Running\n";
  q: +personProt := p.channel;
  server1Proc(q);
  server2Proc(q);
end

clientProc = process |

  sys.out := "First Name: ";
  fName: Text := sys.inp;

  sys.out := "Last Name: ";
  lName: Text := sys.inp;

  sys.out := "Day of Birth: ";
  dBirth: Integer := int sys.inp;

  sys.out := "Month of Birth: ";
  mBirth: Integer := int sys.inp;

  sys.out := "Year of Birth: ";
  yBirth: Integer := int sys.inp;

  sys.out := "Nationality: ";
  nationality: Text := sys.inp;

  sys.out := "\n\n";

  p: −commProt;
  componentProc(p);
  q: −personProt;
  p.channel := q;

  q.fName := fName;
  q.lName := lName;
  q.dayOfBirth := dBirth;
  q.monthOfBirth := mBirth;
  q.yearOfBirth := yBirth;
  q.nationality := nationality;
  q.stop;
  q.stop;

  sys.out := "\n";

end

ctl = cell
  clientProc();
end

ctl()
```

C.3.2.2 Results

Running the program produces the following output:

```
First Name: Laura
Last Name: Smith
Day of Birth: 10
Month of Birth: 10
Year of Birth: 1980
Nationality: Canadian



componentProc is Running
Your First Name is: Laura
Your Last Name is: Smith
Your Day of Birth: 10
Your Month of Birth: 10
Your Year of Birth: 1980
Your Nationality: Canadian
```

C.3.3 Process Composite Pattern Version Three

C.3.3.1 Code

The complete code demonstrating the third version of the Process Composition is as follows:

```
personProt =
protocol
  (fName: Text|
  lName: Text|
  dayOfBirth: Integer|
  monthOfBirth: Integer|
  yearOfBirth: Integer|
  nationality: Text);
  stop
end

commProt = protocol
  channel: personProt
end

server2Proc = process p: +personProt |
  loopselect
    || sys.out := "Your Day of Birth: " + text p.dayOfBirth + "\n";
    || sys.out := "Your Month of Birth: " + text p.monthOfBirth + "\n";
    || sys.out := "Your Year of Birth: " + text p.yearOfBirth + "\n";
    || p.stop; exit;
  end
end

server1Proc = process p: +personProt |
  q: −personProt;
  server2Proc(q);
  loopselect
    || sys.out := "Your First Name is: " + p.fName + "\n";
    || sys.out := "Your Last Name is: " + p.lName + "\n";
    || sys.out := "Your Nationality: " + p.nationality + "\n";
    || q.dayOfBirth := p.dayOfBirth;
    || q.monthOfBirth := p.monthOfBirth;
    || q.yearOfBirth := p.yearOfBirth;
    || p.stop; q.stop; exit;
  end
end

componentProc = process p: +commProt |
  sys.out := "componentProc is Running\n";
  q: +personProt := p.channel;
  server1Proc(q);
end

clientProc = process |

  sys.out := "First Name: ";
  fName: Text := sys.inp;

  sys.out := "Last Name: ";
  lName: Text := sys.inp;

  sys.out := "Day of Birth: ";
  dBirth: Integer := int sys.inp;

  sys.out := "Month of Birth: ";
  mBirth: Integer := int sys.inp;

  sys.out := "Year of Birth: ";
  yBirth: Integer := int sys.inp;

  sys.out := "Nationality: ";
  nationality: Text := sys.inp;

  sys.out := "\n\n";

  p: −commProt;
  componentProc(p);
  q: −personProt;
  p.channel := q;
```

```
    q.fName := fName;
    q.lName := lName;
    q.dayOfBirth := dBirth;
    q.monthOfBirth := mBirth;
    q.yearOfBirth := yBirth;
    q.nationality := nationality;
    q.stop;

    sys.out := "\n";

  end

  ctl = cell
    clientProc();
  end

  ctl()
```

C.3.3.2 Results

The execution of the program outputs:

```
First Name: Laura
Last Name: Smith
Day of Birth: 10
Month of Birth: 10
Year of Birth: 1980
Nationality: Canadian



componentProc is Running
Your First Name is: Laura
Your Last Name is: Smith
Your Day of Birth: 10
Your Month of Birth: 10
Your Year of Birth: 1980
Your Nationality: Canadian
```

Appendix D

Chain of Responsibility Design Pattern in Erasmus

D.1 Code

The full code of the Chain of Responsibility Pattern in Erasmus is demonstrated in the following section.

```
prot = protocol
  *( i: Integer; txt: Text );
  stop
end

handler1 = process i:Integer; p: +prot; q: −prot |
  loopselect
    || tempVal: Integer := p.i;
      str: Text := p.txt;
      if tempVal = 1
      then
        sys.out := "Round" + text i + " Chain1 has processed " + str + "\n";
      else
        q.i := tempVal;
        q.txt := str;
      end;
    || p.stop; q.stop; exit
  end
end

handler2 = process i:Integer; p: +prot; q: −prot |
  loopselect
    || tempVal: Integer := p.i;
      str: Text := p.txt;
      if tempVal = 2
      then
        sys.out := "Round" + text i + " Chain2 has processed " + str + "\n";
      else
        q.i := tempVal;
        q.txt := str;
      end;
    || p.stop; q.stop; exit
  end
end

handler3 = process i:Integer; p: +prot |
  loopselect
    || p.i; str: Text := p.txt;
      sys.out := "Round" + text i + " Chain3 has processed " + str + "\n";
    || p.stop; exit
  end
end

client = process i:Integer; p: −prot |
  sys.out := "\n";
  p.i := 3;
  p.txt := "Three";

  p.i := 1;
  p.txt := "One";

  p.i := 2;
  p.txt := "Two";

  p.stop
end

chain1Cell = cell
  p1, p2, p3: prot;
  client (1, p1);
  handler1(1, p1, p2);
  handler2(1, p2, p3);
  handler3(1, p3);
end

chain2Cell = cell
  p1, p2: prot;
  client (2, p1);
  handler1(2, p1, p2);
  handler3(2, p2);
end

ctl = cell
  chain1Cell();
  chain2Cell();
 end

ctl()
```

D.2 Results

The program produces the following output:

```
Round2 Chain3 has processed Three
Round2 Chain1 has processed One
Round1 Chain1 has processed One
Round1 Chain3 has processed Three
Round2 Chain3 has processed Two
Round1 Chain2 has processed Two
```

Appendix E

Command Design Pattern in Erasmus

E.1 Command Pattern

E.1.1 Code

The full code of the demonstrative program of the Command Pattern is:

```
import stack_ver_2

appProt = protocol
  *( btnTxt: Text | btnUndo | mnInt: Integer | mnUndo ) ;
  stop
end

btnProt = protocol
  *( btnTxt: Text | undo) ;
  stop
end

txtProt = protocol
  *( txt: Text)
end

mnProt = protocol
  *(mnInt: Integer | undo);
  stop
end

intProt = protocol
  *( num: Integer)
end
```

```
btnCommandProc = process p: +btnProt; q: −txtProt|
  t: −txtStackProt;
  txtStackProc(t);
  loopselect
    || str: Text := p.btnTxt;
      t.pushVal := str; t.push;
    || p.undo;
      t.pop;
      aStr: Text := t.popVal;
      q.txt := aStr;
    || p.stop; t.stop; exit;
  end
end


mnCommandProc = process p: +mnProt; q: −intProt|
  t: −intStackProt;
  intStackProc(t);
  loopselect
    || num: Integer := p.mnInt;
      t.pushVal := num; t.push;
    || p.undo;
      t.pop;
      val: Integer := t.popVal;
      q.num := val;
    || p.stop; t.stop; exit;
  end
end

buttonReceiverProc = process p: +btnProt |
  q: −btnProt;
  r: +txtProt;
  btnCommandProc(q, r);
  loopselect
  ||  str: Text := p.btnTxt;
      q.btnTxt := str;
      sys.out := "button Text: " + str + '\n';
  ||  p.undo;
      q.undo;
      aStr: Text := r.txt;
      sys.out := "Button Undo result: " + aStr + "\n";
  ||  p.stop; q.stop; exit
  end
end

menuReceiverProc = process p: +mnProt |
  sum: Integer := 0;
  q: −mnProt;
  r: +intProt;
  mnCommandProc(q, r);
  loopselect
  ||  num: Integer := p.mnInt;
```

```
      q.mnInt := num;
      sum += num;
      sys.out := "menu total Number: " + text sum + '\n';
   || p.undo;
      q.undo;
      aNum: Integer := r.num;
      sum -= aNum;
      sys.out := "Menu After Undo Total: " + text sum + "\n";
   || p.stop; q.stop; exit
  end
end

invokerProc = process p: +appProt; q: -btnProt; r: -mnProt|
  loopselect
     || str: Text := p.btnTxt;
        q.btnTxt := str;
     || p.btnUndo;
        q.undo;
     || i: Integer := p.mnInt;
        r.mnInt := i;
     || p.mnUndo;
        r.undo;
     || p.stop; q.stop; r.stop; exit
  end
end

client = process p: -appProt |
  p.btnTxt := "Button";
  p.btnTxt := "OK";
  p.btnTxt := "Cancel";
  p.btnUndo;
  p.btnUndo;

  p.mnInt := 3;
  p.mnInt := 2;
  p.mnInt := 4;
  p.mnUndo;

  p.stop;
end

ctl = cell
  p: appProt;
  q: btnProt;
  r: mnProt;
  client (p);
  invokerProc(p, q, r);
  buttonReceiverProc(q);
  menuReceiverProc(r);
end

ctl ()
```

E.1.2 Results

Running the program gives the following output:

```
button Text: Button
button Text: OK
button Text: Cancel
Button Undo result: Cancel
menu total Number: 3
menu total Number: 5
Button Undo result: OK
menu total Number: 9
Menu After Undo Total: 5
```

E.2 Stack Library

The Stack Library includes a process stack that stores data of type Integer and another process stack that stores data of type Text. The Stack Library implementation is as shown below:

E.2.1 Code

```
intStackProt = protocol
  *(pushVal: Integer | push | pop; ↑popVal: Integer);
  stop
end

txtStackProt = protocol
  *(pushVal: Text | push | pop; ↑popVal: Text);
  stop
end

intStackProc = process p: +intStackProt |
   stack: Integer indexes Integer;
   count: Integer := −1;
   loopselect
      || num: Integer := p.pushVal; p.push;
        count +=1;
        stack[count] := num;
      || p.pop;
        if count < 0
          then sys.out := "Stack Overflow!\n"
        else
          num: Integer := stack[count]; count −= 1;
          p.popVal := num;
        end
      || p.stop; exit
    end
end

txtStackProc = process p: +txtStackProt |
    stack: Integer indexes Text;
    count: Integer := −1;
    loopselect
      || txt: Text := p.pushVal; p.push;
        count +=1;
        stack[count] := txt;
      || p.pop;
        if count < 0
          then sys.out := "Stack Overflow!\n"
        else
          txt: Text := stack[count]; count −= 1;
          p.popVal := txt;
        end
      || p.stop; exit
    end
end
```

Appendix F

Interpreter Design Pattern in Erasmus

F.1 Interpreter Design Pattern in Erasmus

The complete code of the demonstrative program of the Interpreter Pattern in Erasmus and the results of executing the program are listed here:

F.1.1 Code

The code is as listed below:

```
States = < commence, number, name, op, stop >

tokProt = protocol
  *( ↑kind: States; ↑value: Text )
end

interpretProt = protocol
  *(token: Text| ↑result: Integer);
  stop
end

varValProt = protocol
  *(name: Text; value: Integer);
  stop
end

operationProt = protocol
  *(operand: Integer| op: Char);
  ↑result: Integer;
  stop
end
```

```
calculationProt = protocol
  op: Char;
  lhs: Integer;
  rhs: Integer;
  ↑result: Integer;
  stop
end

scannerProc = process expr: Text; out: +tokProt |
  sys.out := 'Scanning: ' // expr // '\n';

  expr += '$';
  ix: Integer := 0;
  buffer: Text := '';
  state: States := commence;
  loop

    ch: Text := expr[ix];
    isDigit: Bool := '0' ≤ ch and ch ≤ '9';
    isLetter: Bool := 'a' ≤ ch and ch ≤ 'z';
    cases

      |state = commence and isDigit|
        buffer := ch;
        state := number;
        ix += 1

      |state = commence and isLetter|
        buffer := ch;
        state := name;
        ix += 1

      |state = commence and ch = '$'|
        out.kind := stop;
        out.value := '';
        exit

      |state = commence|
        out.kind := op;
        out.value := ch;
        ix += 1

      |state = number and isDigit|
        buffer += ch;
        ix += 1
      |state = number|
        out.kind := number;
        out.value := buffer;
        state := commence

      |state = name and (isLetter or isDigit)|
        buffer += ch;
        ix += 1
        |state = name|
          out.kind := name;
          out.value := buffer;
          state := commence

        ||
          sys.err := 'scannerProc error!\n';
          exit
      end
    end
end;

numberProc = process p: +interpretProt|
    loopselect
      || p.result := int p.token;
      || p.stop; exit;
    end
end

calculationProc = process p: +calculationProt |
    lhs, rhs: Integer;
    loopselect
      || op: Char := p.op;
        cases op
          |"+"| p.result := p.lhs + p.rhs
          |"−"| p.result := p.lhs − p.rhs
          |"∗"| p.result := p.lhs ∗ p.rhs;
          |"/"| p.result := p.lhs / p.rhs;
        end;
      || p.stop; exit;
    end
end

operationProc = process q: +operationProt;|
    operation: Char;
    sum, operand: Integer := 0;
    first : Integer := 1;
    lhsFlag, rhsFlag: Bool := false;
    loopselect ordered
      || operand := q.operand;

        if first = 1 then
          sum := operand;
          first += 1;
        else
          cal: −calculationProt;
      calculationProc(cal);

      cal.op := operation;
      cal.lhs := sum;
      cal.rhs := operand;
      sum := cal.result;
      cal.stop;
      end;
    || operation := q.op;
    || q.result := sum ;
    || q.stop; exit;
  end
end

parserProc = process p: −tokProt; qNum, qVar: −interpretProt; qOp: −operationProt|
  tempVal: Integer := 0;
   first : Integer := 1;
  loop
    k: States := p.kind;
    val: Text := p.value;
    cases
      |k = number| qNum.token := val;
                   tempVal := qNum.result;
                   qOp.operand := tempVal;
                   sys.out := "Number: " // tempVal // "\n";
      |k = name|   qVar.token := val;
                   tempVal := qVar.result;
                   qOp.operand := tempVal;
                   sys.out := "Varible: " // val // " has Value: " // tempVal // "\n";
      |k = op|     qOp.op := char val;
                   sys.out := "Opeartion: " // val // "\n";
      |k = stop|   qNum.stop; qVar.stop; exit;
    end
  end;
  sys.out := "\nFinal Result: " // qOp.result // "\n";
  qOp.stop;
end

variableProc = process p: +interpretProt; r: +varValProt|
  name: Text;
  value: Integer;
  loopselect
    || name := p.token;
      tempName: Text := r.name;
      sys.out := "name: " // name // " and gotten Name: " // tempName // "\n";
      if tempName = name then
        p.result := r.value;
      end;
  || p.stop; exit;
  end;
end

varValueProc = process p: −varValProt|

  p.name := "temp";
  p.value := 100;

  p.name := "var";
  p.value := 20;
  p.name := "y";
  p.value := 4;
end

variableCell = cell p: +interpretProt|
  r: varValProt;
  variableProc(p, r);
  varValueProc(r);
end

aCell = cell
  p: tokProt;
  qN, qV: interpretProt;
  qO: operationProt;
  scannerProc('temp+53*var−20/y', p);
  parserProc(p, qN, qV, qO);
  numberProc(qN);
  variableCell(qV);
  operationProc(qO);
end

aCell();
```

F.1.2 Results

The output of the program is listed below. The test input string is "temp+53*var-20/y", where "temp", "var", and "y" are variables with the values: 100, 20, and 4 respectively. The result is calculated mathematically as: (((100+53)*20)-20)/4=760.

```
Scanning: temp+53*var-20/y
Varible: name has Value: 100
Opeartion: +
Number: 53
Opeartion: *
Varible: name has Value: 20
Opeartion: -
Number: 20
Opeartion: /
Varible: name has Value: 4

Final Result: 760
```

F.2 A modified version of the Interpreter Design Pattern in Erasmus

After applying modifications to handle the sequencing of variables, the code of of the Interpreter Pattern is shown in the following section. A demonstration of the output results after running the program follows.

F.2.1 Code

The code is as listed below:

```
States = < commence, number, name, op, stop >

tokProt = protocol
  *( ↑kind: States; ↑value: Text )
end

interpretProt = protocol
  *(token: Text| ↑result: Integer);
  stop
end

varValProt = protocol
  *(name: Text; ↑value: Integer);
  stop
end

operationProt = protocol
  *(operand: Integer| op: Char);
  ↑result: Integer;
  stop
end

calculationProt = protocol
  op: Char;
  lhs: Integer;
  rhs: Integer;
  ↑result: Integer;
  stop
end

scannerProc = process expr: Text; out: +tokProt |
  sys.out := 'Scanning: ' // expr // '\n';

  expr += '$';
  ix: Integer := 0;
  buffer: Text := '';
  state: States := commence;
  loop
```

```
    ch: Text := expr[ix];
    isDigit: Bool := '0' ≤ ch and ch ≤ '9';
    isLetter: Bool := 'a' ≤ ch and ch ≤ 'z';
    cases

      |state = commence and isDigit|
        buffer := ch;
        state := number;
        ix += 1

      |state = commence and isLetter|
        buffer := ch;
        state := name;
        ix += 1

      |state = commence and ch = '$'|
        out.kind := stop;
        out.value := '';
        exit

      |state = commence|
        out.kind := op;
        out.value := ch;
        ix += 1

      |state = number and isDigit|
        buffer += ch;
        ix += 1
      |state = number|
        out.kind := number;
        out.value := buffer;
        state := commence

      |state = name and (isLetter or isDigit)|
        buffer += ch;
        ix += 1
      |state = name|
        out.kind := name;
        out.value := buffer;
        state := commence

      ||
        sys.err := 'scannerProc error!\n';
        exit
    end
  end
end;

numberProc = process p: +interpretProt|
  loopselect
    || p.result := int p.token;
    || p.stop; exit;
  end
```

```
end

calculationProc = process p: +calculationProt |
  lhs, rhs: Integer;
  loopselect
    || op: Char := p.op;
      cases op
        |"+"| p.result := p.lhs + p.rhs
        |"−"| p.result := p.lhs − p.rhs
        |"*"| p.result := p.lhs * p.rhs;
        |"/"| p.result := p.lhs / p.rhs;
      end;
    || p.stop; exit;
  end
end

operationProc = process q: +operationProt;|
  operation: Char;
  sum, operand: Integer := 0;
  first : Integer := 1;
  lhsFlag, rhsFlag: Bool := false;
  loopselect ordered
    || operand := q.operand;

      if first = 1 then
        sum := operand;
        first += 1;
      else
        cal: −calculationProt;
    calculationProc(cal);

    cal.op := operation;
    cal.lhs := sum;
    cal.rhs := operand;
    sum := cal.result;
    cal.stop;
      end;
    || operation := q.op;
    || q.result := sum ;
    || q.stop; exit;
  end
end

parserProc = process p: −tokProt; qNum, qVar: −interpretProt; qOp: −operationProt|
  tempVal: Integer := 0;
  first : Integer := 1;
  loop
    k: States := p.kind;
    val: Text := p.value;
    cases
      |k = number| qNum.token := val;
                   tempVal := qNum.result;
                   qOp.operand := tempVal;
```

```
                      sys.out := "Number: " // tempVal // "\n";
        |k = name|    qVar.token := val;
                      tempVal := qVar.result;
                      qOp.operand := tempVal;
                      sys.out := "Varible: " // val // " has Value: " // tempVal // "\n";
        |k = op|      qOp.op := char val;
                      sys.out := "Opeartion: " // val // "\n";
        |k = stop|    qNum.stop; qVar.stop; exit;
      end
    end;
    sys.out := "\nFinal Result: " // qOp.result // "\n";
    qOp.stop;
  end

  variableProc = process p: +interpretProt; r: −varValProt|
    name: Text;
    value: Integer;
    loopselect
      || r.name := p.token;
      || p.result := r.value;
      || p.stop; r.stop; exit;
    end;
  end

  varValueProc = process p: +varValProt|

    loopselect
      || name: Text := p.name;
        cases name
          |"temp"| p.value := 100;
          |"var"| p.value := 20;
          |"y"| p.value := 4;
        end;
      || p.stop; exit;
    end;
  end

  variableCell = cell p: +interpretProt|
    r: varValProt;
    variableProc(p, r);
    varValueProc(r);
  end

  aCell = cell
    p: tokProt;
    qN, qV: interpretProt;
    qO: operationProt;
    scannerProc('var+53∗temp−20/y', p);
    parserProc(p, qN, qV, qO);
    numberProc(qN);
    variableCell(qV);
    operationProc(qO);
  end
```

aCell();

F.2.2 Results

The output of the program is listed below. The test input string is “var+53*temp-20/y”, where “temp”, “var”, and “y” are variables with the values: 100, 20, and 4 respectively. The result is calculated mathematically as: (((20+53)*100)-20)/4=1820.

```
Scanning: var+53*temp-20/y
Varible: var has Value: 20
Opeartion: +
Number: 53
Opeartion: *
Varible: temp has Value: 100
Opeartion: -
Number: 20
Opeartion: /
Varible: y has Value: 4

Final Result: 1820
```

Appendix G

Iterator Design Pattern in Erasmus

G.1 Iterator Pattern in Erasmus without sharing Aggregates

G.1.1 Simplified Iterator Design in Erasmus

G.1.1.1 Code

The full code of the program discussed in Section 10.2.1.1 and shown in Figure 25 is:

```
iterationProt = protocol
  ∗(add |next | hasNext; ↑hasNextBool: Bool)
end

MapTxt = Integer indexes Text;

aggregateProt = protocol
  ∗(addTxt: Text | ↑getNextTxt: Text | error);
  stop
end

iterateAggregateProc = process p: +iterationProt; q: +aggregateProt|
  map: MapTxt;
  addPosition: Integer := 0;
  loopPosition: Integer := 0;
  loopselect
    || p.add; map[addPosition] := q.addTxt; addPosition += 1;
    || p.next;
      if loopPosition ≤ addPosition
        then
          q.getNextTxt := map[loopPosition]; loopPosition += 1;
        else
          q.error
      end
    || p.hasNext;
      p.hasNextBool := loopPosition < addPosition;
      || q.stop; exit;
    end
  end

  client = process p: −iterationProt; q: −aggregateProt|
     p.add; q.addTxt := "Student";
     p.add; q.addTxt := "doctor";
     p.add; q.addTxt := "teacher";

     p.next;
     select
        || sys.out := "Get First Value: " + q.getNextTxt + "\n";
        || q.error; sys.out := "End of Aggregate!\n"
     end

     p.hasNext; sys.out := "is there a following element? " + text p.hasNextBool + "\n";

     q.stop
  end

  aCell = cell
    p: iterationProt;
    q: aggregateProt;
    client (p,q);
    iterateAggregateProc(p,q);
  end

  aCell()
```

G.1.1.2 Results

The output of the program is:

```
Get First Value: Student
is there a following element? true
```

G.1.2 More Elaborate Iterator Design in Erasmus

G.1.2.1 Code

A more elaborate program that is discussed in Section 10.2.1.1 and shown in Figure 26 is listed here:

```
  iterationProt = protocol
    *(add |next | hasNext; ↑hasNextBool: Bool)
  end

  MapTxt = Integer indexes Text;
```

```
MapInt = Integer indexes Integer;

txtAggregateProt = protocol
  ∗(addTxt: Text | ↑getNextTxt: Text | error);
  stop
end

intAggregateProt = protocol
  ∗(addInt: Integer | ↑getNextInt: Integer | error);
  stop
end


txtIterateAggregateProc = process p: +iterationProt; q: +txtAggregateProt|
  map: MapTxt;
  addPosition: Integer := 0;
  loopPosition: Integer := 0;
  loopselect
    || p.add; map[addPosition] := q.addTxt; addPosition += 1;
    || p.next;
      if loopPosition ≤ addPosition
        then
          q.getNextTxt := map[loopPosition]; loopPosition += 1;
        else
          q.error
      end
    || p.hasNext;
      p.hasNextBool := loopPosition < addPosition;
    || q.stop; exit;
  end
end

intIterateAggregateProc = process p: +iterationProt; q: +intAggregateProt|
  map: MapInt;
  addPosition: Integer := 0;
  loopPosition: Integer := 0;
  loopselect
    || p.add; map[addPosition] := q.addInt; addPosition += 1;
    || p.next;
      if loopPosition ≤ addPosition
        then
          q.getNextInt := map[loopPosition]; loopPosition += 1;
        else
          q.error
      end
    || p.hasNext;
      p.hasNextBool := loopPosition < addPosition;
    || q.stop; exit;
  end
end

iterateAggregateCell = cell p: +iterationProt; q: +txtAggregateProt; r: +iterationProt; s: +intAggregateI
```

```
  txtIterateAggregateProc(p,q);
  intIterateAggregateProc(r,s);
end


txtClient = process p: −iterationProt; q: −txtAggregateProt|
   p.add; q.addTxt := "Student";
   p.add; q.addTxt := "doctor";
   p.add; q.addTxt := "teacher";



   p.next;
   select
      || sys.out := "Get First Value: " + q.getNextTxt + "\n";
      || q.error; sys.out := "End of Aggregate!\n"
   end

   p.hasNext; sys.out := "is thee a following element? " + text p.hasNextBool + "\n";

   q.stop
end

intClient = process p: −iterationProt; q: −intAggregateProt|
   p.add; q.addInt := 10;
   p.add; q.addInt := 100;
   p.add; q.addInt := 1000;

   p.hasNext;
   hasNext: Bool := p.hasNextBool;
   loop while hasNext;
     p.next;
     select
       || sys.out := "Iterate Through get Value: " + text q.getNextInt + "\n";
       || q.error; sys.out := "End of Aggregate!\n"
   end

   p.hasNext;
   hasNext := p.hasNextBool;
   end

   q.stop
end

clientCell = cell p: −iterationProt; q: −txtAggregateProt; r: −iterationProt; s: −intAggregateProt|
  txtClient(p,q);
  intClient(r,s);
end

aCell = cell
  p: iterationProt;
  q: txtAggregateProt;
  r: iterationProt;
  s: intAggregateProt;
   clientCell (p,q,r,s);
  iterateAggregateCell(p,q,r,s);
end

aCell()
```

G.1.2.2 Results

The output the program is:

```
Get First Value: Student
Iterate Through get Value: 10
is thee a following element? true
Iterate Through get Value: 100
Iterate Through get Value: 1000
```

G.1.3 Iterator Design Pattern in Erasmus

G.1.3.1 Code

The program that demonstrates the Iterator Pattern in Erasmus Without sharing the aggregate that is discussed in Section 10.2.1.1 and shown in Figure 27 is listed here:

```
iterationProt = protocol
  *(add |next | hasNext; ↑hasNextBool: Bool)
end

MapTxt = Integer indexes Text;
MapInt = Integer indexes Integer;

txtAggregateProt = protocol
  *(addTxt: Text | ↑getNextTxt: Text | error);
  stop
end

intAggregateProt = protocol
  *(addInt: Integer | ↑getNextInt: Integer | error );
  stop
end

commProt = protocol
  *( txtSignal; txtItr: iterationProt; txtAgg: txtAggregateProt|
  intSignal; intItr: iterationProt; intAgg: intAggregateProt);
  stop
end
```

```
txtIterateAggregateProc = process p: +iterationProt; q: +txtAggregateProt|
  map: MapTxt;
  addPosition: Integer := 0;
  loopPosition: Integer := 0;
  loopselect
    || p.add; map[addPosition] := q.addTxt; addPosition += 1;
    || p.next;
      if loopPosition ≤ addPosition
        then
          q.getNextTxt := map[loopPosition]; loopPosition += 1;
        else
          q.error
      end
    || p.hasNext;
      p.hasNextBool := loopPosition < addPosition;
    || q.stop; exit;
  end
end

intIterateAggregateProc = process p: +iterationProt; q: +intAggregateProt|
  map: MapInt;
  addPosition: Integer := 0;
  loopPosition: Integer := 0;
  loopselect
    || p.add; map[addPosition] := q.addInt; addPosition += 1;
    || p.next;
      if loopPosition ≤ addPosition
        then
          q.getNextInt := map[loopPosition]; loopPosition += 1;
        else
          q.error
      end
    || p.hasNext;
      p.hasNextBool := loopPosition < addPosition;
    || q.stop; exit;
  end
end

itrAggrFactoryProc = process p: +commProt|
  loopselect
    || p.txtSignal; txtIterateAggregateProc(p.txtItr, p.txtAgg);
    || p.intSignal; intIterateAggregateProc(p.intItr, p.intAgg);
    || p.stop; exit;
  end
end

iterateAggregateCell = cell p: +commProt|
  itrAggrFactoryProc(p);
end


client = process p: −commProt|
  p.txtSignal;
  q: −iterationProt;
  p.txtItr := q;
  r: −txtAggregateProt;
  p.txtAgg := r;

  q.add; r.addTxt := "Student";
  q.add; r.addTxt := "doctor";
  q.add; r.addTxt := "teacher";
  q.add; r.addTxt := "farmer";

  q.hasNext;
  hasNext: Bool := q.hasNextBool;
  loop while hasNext;
   q.next;
   select
      || sys.out := "Iterate Through get Value: " + text r.getNextTxt + "\n";
      || r.error; sys.out := "End of Aggregate!\n"
   end

   q.hasNext;
   hasNext := q.hasNextBool;
  end
  r.stop;

  p.intSignal;
  qq: −iterationProt;
  p.intItr := qq;
  rr: −intAggregateProt;
  p.intAgg := rr;

  qq.add; rr.addInt := 10;
  qq.add; rr.addInt := 100;
  qq.add; rr.addInt := 1000;

  qq.hasNext;
  hasNext := qq.hasNextBool;
  loop while hasNext;
    qq.next;
    select
       || sys.out := "Iterate Through get Value: " + text rr.getNextInt + "\n";
       || rr.error; sys.out := "End of Aggregate!\n"
    end

    qq.hasNext;
    hasNext := qq.hasNextBool;
  end
  rr.stop;
  p.stop
end

clientCell = cell p: −commProt|
    client (p);
end

mainCell = cell
    p: commProt;
    clientCell (p);
    iterateAggregateCell(p);
end

mainCell()
```

G.1.3.2 Results

The output of the program is:

```
Iterate Through get Value: Student
Iterate Through get Value: doctor
Iterate Through get Value: teacher
Iterate Through get Value: farmer
Iterate Through get Value: 10
Iterate Through get Value: 100
Iterate Through get Value: 1000
```

G.2 Iterator Pattern in Erasmus by sharing Aggregates

G.2.1 Simplified Design of the Iterator by sharing Aggregates in Erasmus Code

G.2.1.1 Code

The simple program that is discussed in Section 10.2.2.1 and shown in Figure 28 is listed here:

```
iterationProt = protocol
    *(next; ↑getNextTxt: Text | next; (error | hasNext; ↑hasNextBool: Bool));
    stop
end

aggregateProt = protocol
    *(val: Text)
end

MapTxt = Integer indexes Text;


iteratorProc = process
```

```
  p: +aggregateProt;
  q: +iterationProt;
   alias m: MapTxt;
   alias size: Integer
|
  loopPosition: Integer := 0;
  loopselect
    || txt: Text := p.val;
      m[size] := txt; size +=1;
    || q.next;
      if loopPosition < size
        then
          q.getNextTxt := m[loopPosition]; loopPosition += 1;
      else
        q.error
      end
    || q.hasNext;
      q.hasNextBool := loopPosition < size;
    || q.stop; exit;
  end
end

iteratorPartProc = process
  p: +aggregateProt;
  q: +iterationProt;
  alias m: MapTxt;
  alias size: Integer
|
  loopPosition: Integer := 0;
  loopselect
    || txt: Text := p.val;
      m[size] := txt; size +=1;
    || q.next;
      if loopPosition < size
        then
          q.getNextTxt := m[loopPosition]; loopPosition += 2;
      else
        q.error
      end
    || q.hasNext;
      q.hasNextBool := loopPosition < size;
    || q.stop; exit;
  end
end

IteratorCell = cell
  p: +aggregateProt;
  q: +iterationProt;
  r: +aggregateProt;
  s: +iterationProt
|
  m: MapTxt;
  size: Integer := 0;
  iteratorProc(p, q, m, size);
  iteratorPartProc(r, s, m, size);
end

clientLoopAll = process p: −aggregateProt; q: −iterationProt |
   p.val := "Student";
   p.val := "doctor";
   p.val := "teacher";
   p.val := "hairdresser";
   p.val := "farmer";

   q.hasNext;
   hasNext: Bool := q.hasNextBool;
   loop while hasNext;
     q.next;
     select
        || sys.out := "Iterate  all: " + q.getNextTxt + "\n";
        || q.error; sys.out := "End of Aggregate!\n"
     end
     q.hasNext;
     hasNext := q.hasNextBool;
   end
   q.stop;
end

clientLoopPart = process p: −aggregateProt; q: −iterationProt |
   p.val := " civil  engineer";
   p.val := "computer engineer";
   p.val := " electrical  engineer";
   p.val := "mechanical engineer";

   q.hasNext;
   hasNext: Bool := q.hasNextBool;
   loop while hasNext;
     q.next;
     select
        || sys.out := "Iterate  Part: " + q.getNextTxt + "\n";
        || q.error; sys.out := "End of Aggregate!\n"
     end
     q.hasNext;
     hasNext := q.hasNextBool;
   end
   q.stop;
end

mainCell = cell
  p: aggregateProt
  q: iterationProt;
  clientLoopAll(p, q);

  r: aggregateProt
```

```
    s: iterationProt;
    clientLoopPart(r, s);

    IteratorCell(p, q, r, s)
  end

  mainCell()
```

G.2.1.2 Results

The output of the program is:

```
Iterate Part: civil engineer
Iterate all: civil engineer
Iterate Part: computer engineer
Iterate all: Student
Iterate Part: electrical engineer
Iterate all: computer engineer
Iterate Part: mechanical engineer
Iterate all: doctor
Iterate Part: farmer
Iterate all: electrical engineer
Iterate all: teacher
Iterate all: mechanical engineer
Iterate all: hairdresser
Iterate all: farmer
```

G.2.2 Iterator Design Pattern in Erasmus

G.2.2.1 Code

The program that demonstrates the Iterator Design Pattern by Sharing Aggregates and that is discussed in Section 10.2.2.1 and shown in Figure 29 is listed here:

```
  iteratorProt = protocol
    *(
      add: Text |
      reset |
      ( ↑hasNext: Bool ) |
      ( next; (( ↑val: Text | error ) | ( ↑valInt: Integer | error )))
    ); stop
  end

  MapText = Integer indexes Text
```

```
iteratorProc = process
  p: +iteratorProt;
  alias map: MapText;
  alias size: Integer
|
  index: Integer := 0;
  loopselect
   || map[size] := p.add; size += 1
   || p.reset; index := 0
   || p.hasNext := exists map[index]
   || p.next;
      if exists map[index]
        then p.val := map[index]; index += 1
        else p.error
      end
   || p.stop; exit
  end
end

MakeIterProt = protocol ∗chan: iteratorProt; ↑stop end

iteratorFactoryProc = process
  q: +MakeIterProt;
  alias map: MapText;
  alias size: Integer
|
  loopselect
   || iteratorProc(q.chan, map, size);
   || q.stop; exit
  end
end

iteratorCell = cell q: +MakeIterProt |
  map: MapText;
  size: Integer := 0;
  iteratorFactoryProc(q, map, size)
end

finishProt = protocol finished end

putterProc = process p: −iteratorProt; q: +finishProt |
  p.add := "doctor";
  p.add := "teacher";
  p.add := "hairdresser";
  p.add := "farmer";
  p.stop;
  q.finished;
end

getterProc = process p: −iteratorProt; name: Text |
  for count in 1 to 2 do
    p.reset;
    loop while p.hasNext;
      p.next;
      select
      || val: Text := p.val;
         sys.out := name // ' ' // count // ' ' // val // '\n'
      || p.error; sys.out := "Error!\n"
      end
    end
  end;
  p.stop
end

builderProc = process |
  q: −MakeIterProt;
  iteratorCell (q);

  p0: iteratorProt;
  q.chan := p0;
  f: −finishProt;
  putterProc(p0, f);
  f.finished;

  p1: iteratorProt;
  q.chan := p1;
  getterProc(p1, "Anne");

  p2: iteratorProt;
  q.chan := p2;
  getterProc(p2, "Bill");

  q.stop;
end

mainCell = cell
  builderProc()
end

mainCell()
```

G.2.2.2 Results

The output of the program is:

```
Anne 1 doctor
Bill 1 doctor
Bill 1 teacher
Anne 1 teacher
Bill 1 hairdresser
Anne 1 hairdresser
```

```
Bill 1 farmer
Anne 1 farmer
Bill 2 doctor
Anne 2 doctor
Bill 2 teacher
Anne 2 teacher
Bill 2 hairdresser
Anne 2 hairdresser
Bill 2 farmer
Anne 2 farmer
```