INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

UMI[®]
800-521-0600

Performance of Object Oriented Design with Patterns

by

Ileana Constantinescu

B.Eng.(Electrical Engineering), Polytechnical Institute Bucharest

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for degree of

Master of Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6

September 1999

Copyright 1999, Ileana Constantinescu

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-48448-3

Canada

Abstract

This thesis intent was to bring a contribution to the effort of introducing performance measurements and analysis in the software development process, and also to bring a better understanding to the performance of design patterns as object-oriented advanced features.

The thesis presents results from studies on the analysis and performance measurements of the collection of twenty three design patterns presented in the Design Pattern Elements of Reusable Object-Oriented Software, by Erich Gamma, Richard Helm, Ralph John, and John Vlissides.

After studying the implementation and behavior of the patterns, results of performance measurements are presented and interpreted. This thesis makes three important contributions.

First, it takes a pragmatic approach by providing working examples for all patterns in the catalog, which were then measured and analysed.

Second, it provides conclusions on the performance analysis for the individual patterns, and for each type of design pattern presented in the catalogue (Creational, Structural, and Behavioral). A good understanding of patterns, together with formulas for the calculation of their overhead, are presented.

Third, the thesis concludes with an analysis of various C++ object oriented overheads (creation/ destruction of objects, subclassing, associations) versus patterns mechanism performance issues. For the major issues, several solutions are proposed

The undersigned recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis:

“Performance of Object Oriented Design with Patterns”

submitted by Ileana Constantinescu, B.Eng.
in partial fulfillment of the requirements for
the degree of Master's of Engineering

Chair, Department of Systems and Computer Engineering

Thesis Supervisor
Professor D. Petriu

Carleton University
September, 1999

Acknowledgment

An important key point of every research is to have someone to talk about problems throughout the research, in short, to have a guardian angel as a supervisor. I was lucky to have Prof. Dr. Dorina Petriu as my supervisor and I would like to express my sincere thanks to her for direction, guidance and advice. Her time and patience committed to communicate her knowledge, judgement, and experience in Performance Measurements and Computer Engineering has been rewarded with the results of the research presented in this thesis.

I would like to offer my appreciation to Mirela Anuichi, a Software designer in Nortel Networks for her support and advice with lab measurements. I experienced her as a friend and remember our discussions about patterns and their possible applications in our projects

The support and assistance from secretaries in the Department of System and Computer Engineering are also very much appreciated.

And finally and most importantly, I would like to thank my mother and my husband for their patience, understanding and help throughout the course of this thesis elaboration.

Table of Contents

Chapter 1 - Introduction	1
Objectives	1
Contributions	2
Chapter 2 - Problem Context	4
Overview	4
Object-Oriented Technology	4
Patterns in general	5
Architectural Patterns	8
Design Patterns	9
Idioms	10
Chapter 3 - Performance Evaluation	11
Object Oriented Performance Evaluation	11
Object Oriented Overheads	12
Design Patterns Performance	17
The Tool	18
Performance Evaluation results	20
Chapter 4 - Creational Patterns	23
Factory Method	24
Abstract Factory	29
Builder	35
Prototype	40
Singleton	45
Conclusions	50
Chapter 5 - Structural Patterns	52
Adapter	53
Bridge	61
Composite	66
Decorator	70

Facade	74
Flyweight	78
Proxy	83
Conclusion on Structural Patterns	89
Chapter 6 - Behavioural Patterns	91
Chain of Responsibility	92
Command	97
Interpreter	102
Iterator	108
Memento	112
Observer	117
Mediator	125
State	126
Strategy	131
Template Method	136
Visitor	139
Conclusions for Behavioral Patterns	146
Chapter 7 - Idioms	150
Chapter 8 - Conclusions and Future Research	151
Conclusions	151
Appendix: UML Conventions	154
Bibliography	155

List of Figures

Factory Method: UML diagram	24
Factory Method pattern: event sequence chart	26
Execution time diagram of the Factory Method pattern example	28
Abstract Factory Pattern: UML diagram	29
AbstractFactory pattern example: UML diagram	31
Abstract Factory: event sequence chart	32
Execution time diagram of the AbstractFactory pattern example	34
Builder pattern UML diagram	35
Builder Pattern: event sequence chart	37
Builder pattern example - Execution time measurements	39
Prototype pattern: UML diagram	40
Prototype: event sequence chart	41
Product Prototype example: UML diagram	42
Prototype pattern example: call graph	44
Singleton pattern: UML diagram	45
Singleton pattern: event sequence chart	46
Singleton with no subclassing: UML diagram	46
Execution time diagram of the Singleton example	49
Adapter Class: UML diagram	53
Object Adapter: UML diagram	54
Adapter pattern: event sequence chart	55
Class Adapter UML diagram	56
Object Adapter: UML diagram	57
Standard implementation of the Target-Adapter	57
Class Adapter example call graph	60
Object Adapter example: call graph	60
Bridge pattern: UML diagram	61

Bridge pattern: event sequence diagram	62
Bridge pattern example: UML diagram	63
Bridge pattern example: call graph	65
Composite pattern: UML diagram	66
Composite pattern: event sequence chart	67
Composite pattern example: call graph	69
Decorator pattern: UML diagram	70
Decorator pattern: event sequence diagram	71
Non-pattern implementation of the Component example	72
Decorator pattern example: call graph	73
Facade pattern functional diagram	74
Facade pattern: event sequence chart	75
Facade example UML diagram	76
Facade pattern example: call graph	77
Flyweight pattern: UML diagram	78
Flyweight pattern: event sequence chart	79
Flyweight pattern example: call graph	82
Proxy pattern: UML diagram	83
Proxy example: event sequence chart	84
Proxy example: UML diagram	86
Proxy example: call graph	88
Chain of Responsibility pattern: UML diagram	92
Chain pattern: event sequence chart	93
Chain of Responsibility example: UML diagram	94
Event sequence chart for request handled by the first element of the Chain.	95
Event sequence chart for the request handled by the second element of the Chain	96
Chain pattern example: call graph	96
Command pattern: UML diagram	97

Command pattern: event sequence chart	98
Command example: UML diagram	99
Command pattern example: call graph	101
Interpreter pattern: UML diagram	102
Interpreter pattern: event sequence chart	103
Case study: UML diagram	105
Interpreter pattern: call graph	107
Iterator pattern: UML diagram	108
Iterator pattern: event sequence chart.	109
Iterator example: call graph	111
Memento pattern: UML diagram	112
Memento pattern: event sequence chart	113
Memento case study: UML diagram	114
Memento pattern: call graph	116
Observer pattern: UML diagram	117
Observer pattern: event sequence chart	118
Memento case study: UML diagram	119
Observer pattern example: call graph	121
Mediator pattern: UML diagram	122
Mediator pattern: event sequence chart	123
Mediator case study: UML diagram	124
Mediator example: call graph	125
State pattern: UML diagram	126
State pattern: event sequence chart	127
State pattern example: UML diagram	128
State pattern example: call graph	130
Strategy pattern: UML diagram	131
Strategy pattern: event sequence chart	132
Case study: UML diagram	133

Strategy pattern example: call graph	135
Template pattern: UML diagram	136
Template Method example: call graph	138
Visitor pattern: UML diagram	140
Standard Visitor: event sequence chart	142
Standard Visitor case study: UML diagram	143
Visitor example: call graph	145

List of Tables

Factory Method example execution time distribution	27
Abstract factory example - execution time distribution	33
Execution time diagram of the Factory Method pattern example	38
Builder example execution time distribution	39
Prototype - execution time distribution	44
Singleton example - execution time distribution	48
Performance of Creational Patterns	50
Adapter - execution time distribution	58
ExternalPolymorphism - execution time distribution	59
Bridge pattern execution time distribution	65
Composite pattern execution time distribution	69
Decorator example execution time distribution	73
Facade- execution time distribution	77
Flyweight - execution time distribution	82
Proxy - execution time distribution	87
Performance of Structural Patterns	90
Chain pattern execution time distribution	96
Command pattern execution time distribution	100
Interpreter execution time distribution	106
Iterator execution time distribution	111
Memento execution time distribution	115
Observer example execution time distribution	120
Mediator example execution time distribution	125
State example execution time distribution	130
Strategy example execution time distribution	134
Visitor pattern variants	144
Performance of Behavioral Patterns	147

Chapter 1

1.0 Introduction

1.1 Objectives

The object-oriented technology has many benefits such as reusability, extensibility, and scalability. Lately, design patterns have received a lot of attention in object-oriented community. Many people in the software engineering community recognize the importance of patterns; the reason of their interest is not the novelty of the subject, but mostly the potential of introducing patterns in software design. Some of the main benefits of the patterns are: they organize thinking communicating useful ideas effectively, they help to manage software complexity, and provide solution to recurring design problems that arise in specific design. With patterns, design reuse becomes effective and concrete, patterns being used to record and reuse the “best practices”. Software developers must confront recurring design challenges involving robustness, efficiency, and extensibility. Successful developers resolve these challenges by applying appropriate design patterns. Traditionally, these patterns have been locked in the minds of expert software developers or buried in complex system source code. The idea of collecting this information was implemented by Eric Gamma with the first and most recognized catalogue of design patterns [Gamma95].

Design Patterns versus Implementation

Despite the theoretical and practical progress made in the design pattern field, there are still issues not addressed, as pointed out by [Cop196]. One of these unsolved issues is the performance evaluation of the software designed with patterns. Although a pattern determines the basic structure of the solution to a particular design problem, it does not specify all the implementation details. A pattern provides a scheme for a generic solution, and not a prefabricated module that can be used “as is”. This scheme should be implemented according to the specific needs of the concrete system. Therefore, both the pattern and its implementation(s) have an impact on performance. As part of the effort to include performance evaluation into the design process, the main objective of this research is to identify the performance issues caused by the generic solution described by the pattern. To achieve this goal the directions followed in the thesis are:

- measure the execution overheads of the patterns published in Gamma's catalogue,
- understand the causes of the overheads and find solutions when possible.

1.2 Contributions

The main contributions of the thesis are:

- studied the performance of the design patterns presented in the catalogue of patterns [Gamma 95] and measure their execution overheads
- gained understanding of how execution time is spent in each pattern, and proposed solutions to improve pattern performance
- studied the performance of several C++ idioms, and measured their overheads
- gained understanding of how the C++ idioms can impact the performance of the design patterns.

1.3 Content

Chapter 2 surveys the literature on design patterns. It explains the major aims of object-oriented programming, and discusses how design patterns are applied in practice and why they are becoming increasingly important. Finally, the chapter looks at which design patterns can provide greater reusability of software systems.

Chapter 3 describes briefly the measuring tools and techniques used in this thesis to measure the performance of various design patterns implementations and of various C++ features. It also includes discussions of the efficiency of different object-oriented features, such as creation/destruction of objects, inheritance, member function calls.

Chapter 4 describes the structure and behavior of Creational patterns, and concludes with the results of the performance measurements and the analysis of the main causes of the overheads.

Chapter 5 presents the Structural patterns, their structure and their behavior. It concludes with the presentation of the results of the performance measurements and performance analysis of the Structural patterns from Gamma's catalogue.

Chapter 6 presents the **Behavioral patterns**, with their structure and behavior. It also concludes with a summary of the performance measurements and performance analysis of the **Behavioral patterns** of Gamma's catalogue.

Chapter 7 concludes the thesis. It describes the contributions to design patterns, and explains some of my reflections after completing this research. Finally, the last section identifies directions for future research.

Chapter 2

2.0 Problem Context

2.1 Overview

This chapter introduces the notion of Design Patterns. A short history of the object-oriented approach in software development summarizes developments of the last decade. The contribution of the design patterns to today's software industry illustrates the importance of these technologies.

The discussion shows how design patterns are applied in practice and why they are becoming increasingly important for software designers.

2.2 Object-Oriented Technology

With the introduction of object-oriented approaches in computer programming in the 1980's, scientists and programmers thought they found "the solution" to create reusable software. The reality showed that the Object Oriented was just one of the first steps, however a very important one, toward software reusability [Booch94].

During the last decade, dozens of object-oriented programming languages were born and many of them have already died. As a result, so-called "reusable software components" could not be reused because the language, in which components were written in, were not available long enough or the programming language had experienced a "major" release update. In the late 1980's, some object-oriented programming languages such as Smalltalk and C++ became widely known.

After the establishment of some object-oriented programming languages, the foundation for reuse of software components was set. Class libraries with "semifinished" software pieces such as network handling classes and graphical user interface classes have been build and facilitate today's software development process [Booch94]. These Class libraries became well known with the term *framework*. Many software houses distribute frameworks with so-called 'unique' features. In fact, although every implementation of a framework might be different, the features that are supported by most of these frameworks are quite similar [Ellis90].

Another approach to provide software reusability is to describe software components in a way that they can easily be implemented without focusing on specific frameworks or programming languages. Such descriptions explain reusable patterns of software components that occur over and over again. The name *Patterns* has been established for such descriptions over the last few years.

2.3 Patterns in general

Patterns help a designer build on the collective experience of skilled software engineers. They capture existing, well-proven experience in software development and help to promote good design practise. Every pattern deals with a specific, recurring problem in the designer implementation of a software system. Patterns can be used to construct software architectures with specific properties.

The following paragraph provides an in-depth explanation of what patterns for software architecture are, and how they help to build software.

2.3.1 What is a Pattern?

When experts work on particular problem, it is unusual for them to tackle it by reinventing a new solution that is completely distinct from existing ones. They often recall a similar problem they have already solved, and reuse the essence of its solution to solve the new problem. This kind of ‘expert behaviour’, the thinking in problem-solving pairs, is common to many different domains, such as architecture, economics and software engineering.

Extracting from specific problem-solving pairs the common factors, leads to patterns: “*These problem-solution pairs tend to fall into families of similar problems and solutions with each family exhibiting a pattern in both the problems and the solutions*” [Johns94]. The architect Christopher Alexander [Alex79] defines the term pattern as:

Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.

As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.

The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing: both a description of a thing which is alive, and a description of the process which will generate that thing.

Many patterns can be found in software architecture. Software engineering knowledge is based on these patterns from practical experience, experts using them in developing applications to solve design problems effectively and elegantly.

Several properties of the patterns are important for software architecture according to [Busch96].

- *A pattern addresses a recurring design problem that arises in specific design solutions, and presents a solution to it.* Consider an example where the problem is supporting variability in user interface. This problem may arise when developing software systems with human-computer interaction. This problem can be solved by separation of responsibilities: the core functionality of the application is separated from its user interface.

- *Patterns document existing, well-proven design experience.* They are not invented or created artificially. They “provide a means to reuse the design knowledge gained by experienced practitioners”. Those familiar with an adequate set of patterns ‘can apply them immediately to design problems without having to rediscover them’. Instead of knowledge existing only in the heads of a few experts, patterns make it more generally available.

- *Patterns identify and specify abstractions that are above the level of single classes and instances, or of components.* Typically, a pattern describes several components, classes or objects, and details their responsibilities and relationships, as well as their cooperation. All components together solve the problem that the pattern addresses, and usually more effectively than a single component.

- *Patterns provide a common vocabulary and understanding for design principles.* Pattern names, if chosen carefully, become part of a widespread design language. They facilitate effective discussion of design problems and their solution. They remove the need to explain a solution to a particular problem with a lengthy and complicated description. Instead you can use a pattern name, and explain which parts of a solution correspond to which components of the pattern, or to which relationships between them.

- *Patterns are a means of documenting software architectures.* They can describe the vision you have in mind when designing a software system. This helps others to avoid violating this vision when extending and modifying the original architecture, or when modifying the systems code.

- *Patterns support the construction of software with defined properties.* Patterns provide a skeleton of functional behaviour and therefore help to implement the functionality of your application. For example, patterns exist for maintaining consistency between cooperating components and for providing transparent peer-to-peer inter-process communication. In addition, patterns explicitly address non-functional requirements for software systems, such as changeability, reliability, testability or reusability.

- *Patterns help to build complex and heterogeneous software architectures.* Every pattern provides a predefined set of components, roles and relationships between them. It can be used to specify particular aspects of concrete software structures. Patterns '*act as building-blocks for constructing more complex designs*' [Schm95]. Understanding and applying well-written patterns saves time when compared to searching for solutions on your own. At the very least a *pattern system* can help to evaluate and assess design alternatives. However, although a pattern determines the basic structure of the solution to a particular design problem, it does not specify a fully-detailed solution. A pattern provides a scheme for a generic solution to a family of problems, rather than a prefabricated module that can be used 'as is'. You must implement this scheme according to the specific needs of the design problem at hand. A pattern helps with the creation of similar units. These units can be alike in their broad structure, but are frequently quite different in their detailed appearance. Patterns help solve problems, but they do not provide complete solutions.

- *Patterns help to manage software complexity.* Every pattern describes a proven way to handle the problem it addresses: the kinds of components needed, their roles, the details that should be

hidden, the abstractions that should be visible, and how everything works. When you encounter a concrete design situation covered by a pattern there is no need to waste time inventing a new solution to your problem. If you implement the pattern correctly, you can rely on the solution it provides.

2.3.2 Patterns Categories

A closer look at existing patterns reveals that they cover various ranges of scale and abstraction. Some patterns help in structuring a software system into subsystems. Other patterns support the refinement of subsystems and components, or of the relationships between them. Further patterns help in implementing particular design aspects in a specific programming language. Patterns also range from domain-independent ones, such as those for decoupling interacting components, to patterns addressing domain-specific aspects such as transaction policies in business applications, or call routing in telecommunication.

Because patterns can be implemented in almost any programming language, they allow to design code that “*meets both functional and non-functional requirements of an application*” [Busch96]. They are largely used in different domains as for example in telecommunications [Schm95a], application such as ET++ [Gamma95]. However, their main applicability comes with the system of patterns, which connects patterns together. As described in [Busch96], “*a pattern system is a powerful vehicle for expressing and constructing software architecture*”.

To refine the classification, experts group patterns into three categories [Busch96]

- Architectural patterns
- Design patterns
- Idioms or Coding Patterns

Each category consists of patterns having a similar range of scale or abstraction.

2.4 Architectural Patterns

Viable software architectures are built according to some overall structuring principle. We describe these principles with *architectural* patterns

An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationship between them.

Architectural patterns are templates for concrete high-level software architectures. They specify the system-wide structural properties of an application, and have an impact on the architecture of its subsystems. The selection of an architectural pattern is therefore a fundamental design decision when developing a software system.

2.5 Design Patterns

The subsystems of a software architecture, as well as the relationships between them, usually consist of several smaller architectural units. We describe these units using *design patterns*. In the early 90's, design patterns become an important area in Object-Oriented design [Johns94].

Within a design pattern, the *Main class* section, provides a description of the most important classes of the pattern, classes that must be consistent throughout the document. Another section, which belongs to the solution part of a design pattern, is the *Sequence Diagram*, focussing on the main classes, and the collaboration between them.

A description of the benefits and the drawbacks of the pattern belongs to the *Benefits* and *Drawbacks* section. A critical evaluation of 'features' and 'non-features' of a pattern helps readers understand when a pattern is appropriate for a specific problem and when it is not. If a design pattern is intended to solve a problem in a specific programming language or for a specific operating system, appropriate limits should be specified in the *Drawbacks* of the pattern. Sometimes, there is a need to describe the consequences of adapting a design pattern to application frameworks.

The main benefit of the Design Pattern application should be reusability, but as with the application of the OOD techniques, reusability is not guaranteed. A good design remains the key of the success. Many software projects died before the designs were finished, because of underestimated design time. Other projects died because of wrongly designed interfaces. The use of design patterns could

provide solutions for some of these problems that occur in daily object-oriented software development.

The catalogue of the Design Patterns [Gamma95], provides the following classification of the Design Patterns in :

- creational patterns - abstract the instantiation process
- structural patterns - describe how to compose classes or objects to create larger structures
- behavioural patterns - deal with partitioning of behavioural responsibilities

2.6 Idioms

Idioms or Coding Patterns, deal with the implementation of particular design issues. They represent the lowest-level patterns, addressing aspects of both design and implementation.

Most idioms are language-specific - they capture existing programming experience. Often the same idiom looks different for different languages, and sometimes an idiom that is useful for one programming language does not make sense in another. For example, the C++ community uses reference-counting idioms to manage dynamically-allocated resources: Smalltalk provides a garbage collection mechanism, so has no need for such idioms.

An *idiom* is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language [Copl96].

Chapter 3

3.0 Performance Evaluation

This chapter describes the run-time performance, the evaluation tool and the components of the Performance Evaluation results.

3.1 Object Oriented Performance Evaluation

The strength of the OO approach is the potential to develop flexible, extensible, reusable, maintainable software. The literature largely accepts that this has a price in term of system performance [Panck95], although very few studies have been reported in the literature on this trade-off. Among the top OO distributed systems research issues identified in [Panck95] are scalability (especially that of large-scale applications) and the need to be able to assess the performance of OO systems during the development process by composing performance models of the software as it is built.

Some Object-Oriented authors prefer to ignore performance during design and implementation phases. For example [Auer96] states that early performance optimizations sacrifice the benefits of good design and make the software more difficult to maintain. They prefer to tune performance at a final stage of development, as in a Smalltalk program whose response time was reduced from 15 seconds to less than a second, mainly by caching. However this approach of “fixing performance problems later” does not work for large complex software systems. There is anecdotal evidence that performance fixes may destroy the structure and architecture, and make the system maintainability much worse.

A case study in [Berg95] describes the OO development of an operating system of 2-million lines of C++ code by about 150 engineers over about two years. Performance goals for critical paths were established and then monitored during design and implementation. Even so, the system fell short by a factor of 2.5 in speed when first put together, and a large tuning effort was needed for the critical paths (which were optimized before the whole product was implemented). The conclusion is that early monitoring of performance is necessary in order to ensure that large, complex OO systems will meet their performance goals.

Many OO advantages, such as flexibility, extensibility, reusability come at a price: more classes and abstract interfaces are required, generalization/ specialization and delegation are extensively used. This makes the run-time behaviour more difficult to follow (as it traverses many objects) and may increase the performance costs. Design patterns help to understand the roles and collaboration of different classes (objects), but do not necessarily reduce their number.

3.2 Object Oriented Overheads

3.2.1 Virtual Functions

Polymorphism is one of the pillars of the Object-Oriented programming being defined as - the ability to call a variety of functions using the same interface.

A key issue associated with polymorphism is the timing of the implementation decision. If the system binds the function's call at compile time this is called "*early binding*". If the binding is made dynamically at run-time it is called "*late binding*". "*Early binding*" provides the advantage of execution speed, while "*late binding*" offers the advantage of flexibility and a high level of problem abstraction. Both *early* and *late binding* may support the concept of polymorphism. C++ handles "*late binding*" through virtual functions and "*early binding*" through direct function call.

When the type of an object is known at compile time, there is no need to use virtual call mechanism. Instead, the direct member function call can be used; it is a few memory references, more efficient than a virtual function call. When the type of an object is not known at compile time, then virtual functions should be used. A virtual function is declared in a parent class when it is known that one or more derived classes will overload the virtual function by defining their own implementation of the function.

Virtual functions are selected dynamically whenever a member function is called through a pointer or reference to the base class. A virtual function may be called directly, when it is called through an object, rather than a pointer or reference [Meyer98].

There are advantages and disadvantages of using virtual functions.

Advantages:

- virtual functions are flexible
- they allow applications to be written generically and extended automatically when additional derived classes are developed
- they encourage easy maintenance of related classes manipulated through a common interface

Disadvantages:

- they take space because they are implemented through virtual tables and virtual table pointers. constructed and maintained by the compiler,
- they take time because the code calls the functions indirectly, and virtual function cannot be inlined.
- they are unnecessary. when regular inheritance supplies access to base class functions that do not need to be redefined

From the measurements performed on design patterns, the call of a function made virtual takes only several clock cycles more time than the direct call of the function.

3.2.2 Constructors and Destructors

Constructors are methods with the same name as the class in which they are defined, that provide the automatic initialization of objects at their point of declaration. The creation of an object is made in two steps:

- 1.Memory allocation
- 2.Memory initialization
 - a) Initialize base class objects
 - b) Initialize all member objects
 - c) Execute constructor body

Steps 1 and 2c are always executed. Step 2a is executed in the case of inheritance, and step 2b in the case of composition. *Destructors* are methods that provide the automatic deallocation of the storage occupied by an object when the block containing the object is exited. This is particularly

important when an object contains storage in the heap. Such storage can only be explicitly deallocated by the programmer using the *delete* function unless automatic deallocation using a destructor is provided. The destruction of an object is made also in two steps:

1. Memory deinitialization

- a) Execute destructor body
- b) Deinitialize all member objects
- c) Deinitialize base class objects

2. Memory deallocation

Steps 1a and 2 are always executed. Step 1c is executed in the case of inheritance, step 1b in the case of composition.

Performance

1. In general, for the same object, the construction and destruction time are of the same order of magnitude
2. The overhead for the construction of composed objects can be very high, since it includes the time to create all member objects. The constructor body of the composed class is executed, only after all the member objects have been initialized. The time necessary to construct a composed object includes the time necessary to construct the component object.
3. Initialization of the member data of a class can be done in the constructor either through an initialization list, or through assignment in the constructor body. The difference between the two initialization methods consists in the number of function calls executed for initialization, as well as the type of the function called (copy constructor or default constructor and assignment operator). The measurements show better results with the initialization list, about twice better than with the assignment members in the body of the constructor.
4. In a class hierarchy, the execution cost of the constructors/destructors goes up linearly with the number of levels in the hierarchy.

3.2.3 Operator new, operator delete

Dynamic creation/deletion of objects is implemented with operator new/operator delete.

Memory Performance

1. Default version of the operator *new*, being a general-purpose operator, may allocate a lot more memory than needed.
 - To communicate the size of the allocated memory block, the operator *new* prepends to the memory some additional data containing this information.
 - Operator *delete* uses this information to deallocated the block of memory allocated by its pair operator *new*.
 - For small objects, this additional information can more than double the amount of memory needed for each dynamically allocated object

Run-time Performance

1. Default operator *new* may also be very expensive; it handles memory requests of different sizes and memory fragmentation
2. Static and automatic creation of an object is a lot less expensive then dynamic object creation with *new*. The difference is about 2 orders of magnitude.
3. Destruction of an object with *delete* takes less time than its creation with *new*.

3.2.4 Member Function Calls

The overhead of a member function call is directly dependent upon the number and the type of parameters and returned results. A member function can accept an object as a parameter and also can returns an object from a function. The best performance can be achieved when the design is passing and returning parameters by reference [Meyer98]. A simple comparison of the execution steps for a call-by-value-return-by-value function, versus a call-by-reference-return-by-reference follows. Call-by-value means that an object is passed by value, which is defined by the Copy Constructor of that object's class. The usage of the Copy Constructor can be very expensive. To void this cost it is recommended to pass the objects by reference. The following execution steps describe the call-by-value-return-by-value implementation

1. First, the copy constructor is invoked to initialize the argument with the passed value
2. The function takes control
3. The function calls the copy constructor to copy the data member into the temporary object returned by the function

4. The destructor is invoked automatically to destruct the object containing the data member

5. The object returned by the function is destroyed in *main*.

In the case of a call-by-reference-return-by-reference are similar to the steps above, except that there is no constructor/destructor invoked, which makes such a call more efficient.

The cost of calling and executing the call-by-reference-return-by-reference function is very small compared with a call and execution time of call-by-value-return-by-value function.

3.2.5 Multiple Inheritance

Introduces complexity that does not exist under Single Inheritance.

- The most basic is **ambiguity**. When a derived class inherits a member name from more than one class, any reference to that name is inherently ambiguous.
- Inheritance hierarchy that starts with MI hierarchy has the tendency to evolve into the diamond-shaped inheritance hierarchy from: to:

```
class B { ... };
class C { ... };
class D: public B, public C { ... };
```

```
class A { ... };
class B: virtual public A { ... };
class C: virtual public A { ... };
class D: public B, public C { ... };
```

Performance issues

- The diamond-shaped inheritance hierarchy imposes an additional cost in both space (memory) and time on clients of those classes
- Virtual functions, in this case, have a big impact on performance

3.2.6 Copy Constructor

To support cloning, C++ provides the *Copy Constructor*, which constructs an object by copying an existing one. Default copy constructor in C++ does shallow copy, i.e. the pointers will be shared between the copy and the original. Cloning objects with complex structures requires copy constructors using deep-copy, to ensure the clone and the original are independent.

Performance

- Deep-copy is expensive, time and space wise
- Shallow copy is less expensive, but if improperly used, it may create dangling pointers
- Initialization of the clone is also more expensive. It requires an Initialize procedure that takes initialization parameters as arguments to set the Clone's internal states to predefined values.

3.3 Design Patterns Performance

The intention of the first catalog of design patterns [Gamma95] is to enhance one or more software quality characteristics such as flexibility, extensibility, modularity, and reusability. Improving performance is not the goal of any of these patterns, although performance concerns are expressed sometimes.

Performance concerns become much stronger in design patterns for distributed applications. For example, the patterns introduced in the of distributed applications for communications, combine the concern for flexibility, extensibility, reusability and portability with the concern for efficiency [Schm94], [Schm95a], [Schm95b], [Schm96a]. [Islam95] describes how the "Distributed Observer" pattern is used in distributed applications. The authors state that the description of patterns for distributed systems should highlight the performance, fault detection and recovery aspects of their solutions, in order to help the designers in their choices.

Larger-scale architectural patterns are presented in [Shaw96], [Monro97] and [Busch96]. The last does more than describing individual patterns: it proposes a new approach to designing large-scale OO software based on a "pattern system". The patterns presented in [Busch96] span several levels of abstraction from high-level architectural patterns and medium-level design patterns to

low-level idioms. This helps designers to shift the focus from high-level to detailed design, always guided by patterns. Again, the concern is more on how to build flexible, extensible systems, than on how to build fast systems.

However, it is generally recognized that the architecture of a software system plays an important role not only in defining its quality, but also in defining its performance [Spitz98], [Willi98]. Even if sometime is very difficult, the developers should be able to assess and understand the performance effects of various architectural decisions, starting at an early stage, when changes are easy and less expensive, and continuing throughout the software life cycle. This can be achieved by constructing and analyzing quantitative performance models that capture the interactions between the main components and point to the performance trouble spots. In [Petri98] and [Petri99] is presented a formal approach to building Layered Queueing Network (LQN) performance models from descriptions of the high-level architecture of a system, and more exactly from the architectural patterns used in the system. The transformation from the high-level architecture to the LQN performance model is done pattern by pattern and is based on the graph-grammar formalism.

Although a pattern determines the basic structure of the solution to a particular design problem, it does not specify all the implementation details. What a pattern provides is a scheme for a generic solution, rather than a prefabricated module that can be used "as is". This scheme should be implemented according to the specific needs of the concrete system. Therefore, both the pattern and its implementation(s) have an impact on performance. One of the challenges of this research is to identify precisely which performance effects are due to the generic solution described by the pattern itself, and which are decided by the implementation.

Performance evaluation in software design is a controversial subject. Some experts support the idea of Late Optimization, others support the idea of Early Optimization. This research agrees with the supporters of early Optimization, i.e. of using performance evaluation from the first stages of the design to ensure well structured, flexible, and performant software in the end.

The method applied in this research was to evaluate the performance of the patterns through measurements of typical examples. It can be replied that the specific implementation of the pattern may have a big impact on the performance. This is true, however the intent is to find the weak points of the pattern mechanism and to evaluate their impact on performance. The method applied is described in paragraph 3.3.

3.4 The Tool

To measure the run-time performance of the patterns, a run-time performance tool was used. The name of the tool is Quantify [Pure95]. Quantify provides accurate execution time measurements for sequential programs written in C or C++, and measures the execution time for functions defined in the program and for the time spent in operating system calls. The records, which exclude any Quantify overhead, can have a minimum and a maximum value. They were measured in machine instruction cycles and converted the elapsed time according to the clock rate of the system. Quantify determines the system type and the clock rate at the instrumentation time.

For code compiled under the tool, Quantify provides very accurate records: for code with the source code unavailable, as for example system calls or third-party library function calls, the measurements are less accurate. For the code compiled under the tool, Quantify uses the output provided by the compiler, and the data collected at run time. The output provided by the compiler consists of:

- division of the code in basic blocks, which are sequences of instructions executed in succession
- calculation of each basic block execution time based on the number of instructions executed by the block, multiplied by the number of machine cycles each instruction requires (*N machine cycles per basic block*)

The run time information consists of the number of times each basic block is executed (*N of basic block executions*). The total execution time is calculated by

$$N \text{ machine cycles per basic block} * N \text{ of basic block executions}$$

For each function there are two records:

- one, for the time spent inside the function
- another one, for the time spent in descendent function calls

For the non-available code (system calls), Quantify can measure the elapsed time (wall-clock) or the kernel time. To measure the elapsed time, Quantify calls the *gettimeofday* utility. if a processor context switch (from user state to kernel state) occurs during the call to *gettimeofday*, the measured time of the system call is larger. This is because Quantify cannot detect a context switch occurrence. To measure the actual time used by a process, Quantify uses *getrusage* utility. *Getrusage* returns information about the resources used by the current process, mainly about the CPU time spent in user mode, or in system mode. This information is stored by the operating system. It samples the state of the system with every tick of the clock, the accuracy being dependent on the clock resolution. The accuracy is higher for larger code blocks. To increase the accuracy of the measurements for smaller blocks of code a designer can use one of the following:

1. Calculate the average of the CPU time measurements on a large number of copies of the code under measurement. Disadvantage: in real applications is not possible to have multiple, consecutive copies of the code.
2. Measure the CPU time for every execution of the code and calculate the average. In this case, it is required to have many executions of the code, to obtain a statistically significant average. Disadvantage: it can be used only if the CPU time for the code execution is significantly larger than the resolution of the CPU timers.
3. A combination of alternatives 1 & 2. Repeat the execution time measurements for a large number of replications of the code.

Pitfalls of using *getrusage*:

- The measurements may be affected by an insufficient resolution of the machine.
- For small pieces of code the overhead caused by the starting/stopping timing, the loop control or the *getrusage* itself, may be significant. It may be necessary to subtract it from the measured value, in order to have a correct estimation of the CPU time of the code under measurement.
- Competing processes may distort CPU time.

In this research the second alternative was used. The values outside the $[a/1.5 \text{ to } 1.5a]$ range, are discarded (here a is an average value).

3.5 Performance Evaluation results

The data was collected by using an UltraSparc machine, with a clock rate of 143 MHz. The execution time of each pattern was calculated in clock cycles, excluding Quantify overhead and memory effects.

The total execution time and the overhead formulas use the following abbreviations:

T	The total execution time of the implementation with patterns
O	The pattern's overhead represents the increase in the execution time caused by the mechanism of the pattern. The reference is the total execution time of the OO implementation, i.e. the total execution time T is: $T = T_{OO} + O$

Total Execution Time Components:

N =	$\sum_n t_n$	Execution time of the <i>new</i> functions. Most of the time being consumed by memory allocation.
D =	$\sum_n t_d$	Execution time of the delete function.
V =	$\sum_v t_v$	Execution time of virtual functions.
F =	$\sum_f t_f$	Execution time of all the functions of the pattern.
C =	$\sum_c t_c$	Execution time of constructors.

Overhead Components:

G =	$\sum_g t_g$	The <i>GetProduct</i> function time (used only in the Builder pattern) contained in the pattern overhead.
-----	--------------	---

T	The total execution time of the implementation with patterns
$I = \sum_i t_i$	The Instance function time (used only in the Singleton pattern) contained in the pattern overhead.
N_{Name}	Execution time of <i>new</i> functions used in the main class of the pattern.
C_{Name}	Execution time of pattern's constructor.
V_{Name}	Execution time of the virtual function of the main class of the pattern; in general it has the name of the pattern.
F_{Name}	Execution time of other functions defined in the main class of the pattern.

where:

n represents the number of calls of the function “new”

t_n represents the time in the function *new* indexed by n

f represents the number of non-virtual functions

t_f represents the execution time of non-virtual function f

v represents the number of virtual functions

t_v represents the execution time of a virtual function v

c represents the number of constructors

t_c represents the execution time of a constructor c

g represents the number of *GetProduct* function

t_g represents the execution time of a *GetProduct* function g

i represents the number of Instance functions

t_i represents the execution time of a function Instance i

Chapter 4

4.0 Creational Patterns

In general, systems tend to evolve from relying on inheritance to relying on object composition, by defining a set of component behaviors that can be combined in different ways. Thus, the creational process becomes more complex than instantiating a class. Creational Patterns deal with initialization and configuring classes and objects capable to:

- abstract the instantiation process
- encapsulate knowledge about concrete classes in the system
- hide how instances of the classes are created and connected
- permit flexibility in what, how, when it gets created, and who creates it
- permit flexibility in configuring the system
- allow for static or dynamic configuration

In a design based on Creational Patterns, the system knowledge on the objects is based on their interfaces, defined by abstract classes. The list of creational patterns as defined in [Gamma95] is:

- Builder** - designed as a factory for creating complex objects incrementally
- Factory Method** - a factory where subclasses implement the creational process
- Abstract Factory** - designed to create related/dependent objects
- Prototype** - a factory for cloning new instances
- Singleton** - designed to create unique instances

These patterns are closely related, their structure presents only small variations from pattern to pattern. For this reason it was simple to study all of them, using a common example to illustrate their implementation and functionality.

4.1 Factory Method

4.1.1 Background

In a parallel subclassed hierarchy of Products and Creators, the Factory Method is an interface for creating Products objects. It defines the connection between these two class hierarchies of Products and Creators, encapsulating the knowledge of which classes belong together.

The Creator class knows when to create a new Product, but it does not know what kind of Product to create. This problem is solved by the ConcreteCreator subclass which defines a CreateProduct operation on the Creator, to return the appropriate Product subclass. The CreateProduct operation is implemented with a virtual Factory Method function.

Structure

FIGURE 1. Factory Method: UML diagram

The pattern can be used when it is difficult to predict the concrete class objects to create, or when a class wants to delegate to subclasses the responsibility of choosing which objects to create.

The Factory Method pattern is very often used in Object Oriented design.

Main Classes

Product - an abstract class interface of objects created by the Factory Method. The Client knows only the Product class.

Concrete Product - a subclass implementing a Product object (when the Product is subclassed)

Creator - a class which defines the Factory Methods. Each Factory Method returns a Product or a Part object. i.e the default implementation of the Factory Method returns a Concrete Product object.

ConcreteCreator - a subclass of the Creator with a Factory Method which overrides the Factory Method of the Creator and returns an instance of a ConcreteProduct. Creating objects inside a Class with a Factory Method is more flexible than creating an object directly. It gives subclasses a chance to provide an extended version of an object.

There are several variations of the Factory Method pattern:

- Creator as an abstract class, when the Factory Method is implemented by the Creator's subclasses
- Creator as a concrete class, when the Factory Method is implemented by the Creator itself
- Parametrized Factory Methods, when the Factory Method takes the product as a parameter

Sequence Diagram

FIGURE 2.Factory Method pattern: event sequence chart

4.1.2 Case Study

There are two main variations of the pattern: one when the Creator is an abstract class and does not provide an implementation of its Factory Method. the other, when the Creator is a concrete class and provides a default implementation of its Factory Method.

Our case study refers to the second case and illustrates the structure of the pattern from Fig.1.

Pattern Application - Roles played by case-study classes:

The **Creator** Class is an abstract class having several Factory Methods: `Make Product`, `Make Part1`, `Make Part2`. The real implementation has no `ConcreteCreator`, the abstract Creator fills in the role of abstract and concrete Creator.

Each of the Factory Methods returns a part of the Product, to the Creator; for example the MakePart1 returns a Part1. The MakeProduct returns the Product. The CreateProduct method invokes all the Factory Methods, and returns the Product. To create different types of Products a design can subclass the Creator class or parameterize the FactoryMethods [Gamma+95].

4.1.3 Performance Evaluation

The total execution time and the overhead of the Product-Creator framework using the Factory Method pattern can be calculated with the following general formulas:

$$\begin{array}{ll} \text{Total execution time} & T = F + V + C + N \\ \text{Overhead} & O_{FactoryMethod} = (F + V)_{FactoryMethod} \end{array}$$

These formulas are general enough to be applied to any of the implementation variants mentioned above.

When the Creator is subclassed, the Factory Method is implemented as a virtual function, to allow flexible implementations of the ConcreteCreators. Subclassing, however, will introduce a higher overhead. A solution to subclassing is to use templated Creators to create the appropriate Product object. In this case, there is no need to use virtual functions; therefore, the virtual function overheads do not exist.

The case study example does not use a subclassed Creator, therefore, the Factory Method is implemented as a non-virtual function. For this scenario as well as for templated Creator scenario, the total execution time and the overhead formulas become:

$$\begin{array}{ll} \text{Total execution time} & T = F + C + N \\ \text{Overhead} & O_{FactoryMethod} = F_{FactoryMethod} \end{array}$$

The real time performance evaluation is based on the real-time measurements shown in Fig. 6. The total cost is 3539 cycles, and the overhead is 134 cycles. The execution time distribution of the case

4.2 Abstract Factory

4.2.1 Background

The Abstract Factory pattern provides an interface that makes a system independent of how its objects are created. It is a mechanism to create groups of related objects without specifying their concrete classes. The pattern can be used for example when a set of related Products has to support multiple standards, or when a system needs to be independent of how the products are created. To achieve this, the pattern does not allow the application to hard code its products and allows the client to see only the products' interfaces, and not their implementation.

Structure

FIGURE 4. Abstract Factory Pattern: UML diagram

Main Classes

AbstractFactory - an interface for operations that create **AbstractProduct** objects through Factory Methods specifically defined for each Product.

ConcreteFactory - a Class which creates its products by overriding the Abstract Factory class. It requires a new factory subclass for each product it builds. For many product families, the ConcreteFactory can be implemented using the Prototype pattern. However because an application needs only an instance of a ConcreteFactory per product family it is usually best implemented as a Singleton pattern.

AbstractProduct - created by the operations defined in the AbstractFactory. The AbstractFactory usually defines a different operation for each kind of AbstractProduct.

ConcreteProduct - creates the products through the Factory Methods defined in the Concrete Factory classes.

Director - creates different products objects by using different ConcreteFactories.

At run time an AbstractFactory class defers the creation of products objects to its ConcreteFactory subclasses. ConcreteProducts are single instances of ConcreteFactory subclasses created at run time.

Sequence Diagram

4.2.2 Case Study

The case study example is based on the AbstractFactory structure shown in Fig. 4. The AbstractFactory pattern is used here to build a Product with two Parts. The UML diagram of the

example is presented in Fig. 5. A detailed description of a similar example can be found in [Gamma95]. The most important characteristic of the AbstractFactory class is the structure of the class which contains a collection of Factory Methods. The AbstractFactory class of this example acts as both an AbstractFactory and a ConcreteFactory.

FIGURE 5. AbstractFactory pattern example: UML diagram

Pattern Application - Roles played by case-study classes

Director class defines CreateProduct method to construct the Product.

AbstractFactory - a Factory class (Abstract and Concrete) which creates the parts of the Product, the Part1, the Part2. It consists of four Factory Methods one for each part of the product and one for the product: MakePart1, MakePart2, MakeProduct.

The AbstractFactory defines the interface for the Product creation. The Product and its parts are created with Factory Methods (MakeProduct, MakePart1, MakePart2), all defined in the AbstractFactory class.

At run time the AbstractFactory creates the products objects, with the correspondent Factory Methods. The Factory Method invokes the new operator to create the parts of the Product (Part1, Part2) and the Product itself.

Sequence Diagram

FIGURE 6. Abstract Factory: event sequence chart

4.2.3 Performance Evaluation

This section presents performance measurements on the example shown in Fig. 9. In this example the AbstractFactory class is not subclassed; therefore, there are no ConcreteFactory classes.

The AbstractFactory defines all Factory Methods (MakeProduct, MakePart1, MakePart2), as virtual functions to allow later expansion of their functionality through AbstractFactory class subclassing.

For example, new type of Products can be added: they require a specific ConcreteFactory, the ConcreteProductFactory, defined as a subclass of the AbstractFactory.

The call graph of the AbstractFactory example is presented in Fig. 10, and the total execution time and overhead formulas are:

Total execution time $T = F + V + C + N$

Overhead $O_{AbstractFactory} = (C + D + N)_{AbstractFactory} + (F + V)_{FactoryMethods}$

The total execution time components are:

F - represents the execution time of all standard functions including the execution time of the Factory Methods and of their descendents. It does not include the overhead of the virtual attribute used with the Factory Methods. This overhead was considered separately and represented by V.

N - represents the cost of dynamic creation with *new* of the Product product as well as of the Part1, Part2.

C - represents the execution time of all the constructors, the constructors of the Product, Part1, Part2, and Product.

The result of the real time measurements are summarized in Table 3.

TABLE 2. Abstract factory example - execution time distribution

T	F	V	N	C	O
3566	96	48	3280	144	164
100%	> 2%	2%	91.5%	4.5%	4%

This table shows clearly that the highest cost of the execution time is due to dynamic objects creation with *new* $N = 91.5\%$. The analysis of dynamic object creation with *new*, and some solution to this problem are presented in chapter 3. The overhead, is very small. It is mainly the result of the static (instead of dynamic) AbstractFactory object creation ($C_{Factory} = 7$ clock cycles), of using virtual attribute for the Factory Methods ($V_{FactoryMethods} = 157$ clock cycles), and of increased execution time for the CreateProduct function due to parametrization. The overhead introduced by CreateProduct is really minimal, because the argument is passed by reference. To decrease even more the overhead added by this pattern, a design can use the following techniques: avoid dynamic instantiation of the AbstractFactory object, there is no need; use return value optimization to compensate for the overhead introduced by the temporary references created by the Factory Methods [Meyer96]; use passed-by-reference instead of passed-by-value.

FIGURE 7. Execution time diagram of the AbstractFactory pattern example

4.3 Builder

4.3.1 Background

The pattern is used to build complex objects incrementally. It isolates the algorithm implementation from the objects that depend on the algorithm, hides the internal representation of the products, and encapsulates the construction of the products in the ConcreteBuilders. This characteristic to isolate the code implementing the construction of an object from the object itself, makes it applicable when an algorithm needs to be extended or optimized.

Structure

FIGURE 8.Builder pattern UML diagram

Main Classes

ProductBuilder Class - This class does not create Products itself; it is a general abstract interface for the creation of the Product objects for all types of ConcreteProductBuilders.

This class hides the internal structure and the representation of the Product and contains pure virtual methods (BuildPart1, BuildPart2, GetProduct), empty methods, to allow subclasses to override certain operations only.

To change the internal representation of the Products without affecting the existing code, it is required to define a new type of Builder.

ConcreteProductBuilder Class - A class used to construct and retrieve the Product. It manages all the representations of the Product, used by different Directors to build different Products. The methods owned by this class are overriding the ProductBuilder methods.

Director Class - Is a constructor of the Product object through the ProductBuilder interface. The Construct() method owned by this Class takes the ConcreteProductBuilder as a parameter.

Product variants are possible through new Directors.

Product - Is the object under construction which includes classes defining the constituent parts of the Product, and the interfaces for assembling the parts.

Parts - Is the abstract interface to access different parts.

Part1, Part2 - Are Concrete Parts used in the process of building the Product.

Sequence Diagram

FIGURE 9.Builder Pattern: event sequence chart

4.3.2 Case Study

The Builder pattern example follows the structure described in Fig.8, with the following classes and function roles:

Pattern Application - Roles played by case-study classes

ProductBuilder - abstract class with the main role to define the BuildProduct, BuildPart1, BuildPart2 functions as interfaces for building a Product, and also to define the GetProduct as a function which returns the Product to the Director.

ConcreteProductBuilder - subclass of the ProductBuilder. The ConcreteProductBuilder class keeps track of the products under construction with the variable **_currentProduct**, and implements the BuildPart1, BuildPart2, BuildProduct, GetProduct functions to override the

equivalent **ProductBuilder** functions. Clients can use **CreateProduct** in conjunction with **ConcreteProductBuilder** to create a **Product**.

Director class. Its **CreateProduct** function, a construct function, takes **ProductBuilder** as a parameter. This **ProductBuilder** can be any of the **ConcreteProductBuilder** subclasses.

CreateProduct - a Construct function

4.3.3 Performance Evaluation

The pattern defines the mechanism for encapsulating the operations to construct an object, by using Builder interface. To create different kinds of objects, the **ProductBuilder** uses subclassing, each object being constructed by a **ConcreteProductBuilder**, a subclass of the abstract **ProductBuilder** interface. On one hand this architecture improves the design flexibility, and on the other hand it affects the performance by inserting another level of indirection.

The performance analysis of the Builder example presented in chapter 4.3.2 is summarized by the following formulas:

$$\begin{array}{ll} \text{Total execution time} & T = V + F + N + C + D + G \\ \text{Overhead} & O_{Builder} = (V + F)_{Builder} + (C + N + D)_{Builder} \end{array}$$

The Builder pattern's overhead has three components: a component introduced by the **Builder** class functions, another introduced by the function **GetProduct** and the last component introduced by the **ConcreteProductBuilder** class. Constructor **CreateProduct** function receives **ProductBuilder** as an argument.

This allows to construct different types of products for different types of **ProductBuilders**.

```
Product* Director::CreateProduct (ProductBuider& builder);
```

To improve performance, the **ProductBuilder** class is passed by reference and not by value; when passed by reference only the address of the argument is copied, instead of the entire object. When a large class is passed-by-value, the time and space costs to allocate and copy the class object onto the stack are often too high. Even when using pass-by-reference there is an overhead. This

overhead, being very small, was not included in the overhead formula. According to the call graph of the case study example presented in Fig. 10, the total execution time is 2667 clock cycles, and the overhead is 123 clock cycles.

The distribution of the execution time identifies *new* as the main time consumer, 86% of the total execution time being spent with the dynamic object creation of the Products. The overhead is small representing only 4% of the total execution time.

TABLE 3.Builder example execution time distribution

T	N	G	F	V	C	O
2667	2286	18	158	79	126	123
100%	86%	0.6%	6%	3%	4.4%	4%

FIGURE 10.Builder pattern example - Execution time measurements

4.4 Prototype

4.4.1 Background

This pattern declares an interface, the **Prototype** class, which creates new objects by copying itself. The pattern has many areas of applicability. It can be used when the design requires to specify product classes only at run time. The advantage of the pattern are: isolate the concrete product class from the client, reduce the number of classes, avoid parallel class hierarchy, add or remove ConcreteProducts at runtime, and configure dynamically the application with classes.

The main benefits of using the pattern are: allows to vary the behaviour of a client by changing the value of its delegate, reduces subclassing by cloning a prototype instead of asking a Factory Method to make a new object. It applies primarily to languages like C++.

The main drawback is the **Clone()** operation, which is expensive and may be difficult to implement and change.

Structure

FIGURE 11. Prototype pattern: UML diagram

Main Classes

Prototype - declares an interface for cloning itself. The cloning operation is the sensitive part of Prototype's implementation. The Prototype and its subclasses must implement the Clone() operation. In C++ cloning is supported with the *copy constructor*.

ConcretePrototype - a subclass of the Prototype. It must implement the Clone() operation

Director - creates a new object by asking a prototype to clone itself.

Sequence Diagram

The Client sends a request to the Prototype class to clone itself.

FIGURE 12. Prototype: event sequence chart

4.4.2 Case Study

The Prototype example follows the structure described in Fig. 11. It consists in creating a Product by using prototyping. The intention is to keep the Abstract Factory as a framework and add the Prototype pattern to it. This mechanism requires to implement several changes in the AbstractFactory framework. The first change consists in augmenting the AbstractFactory interface by subclassing it with a PrototypeFactory. In this example, the PrototypeFactory interface is subclassed with a ConcretePrototypeFactory. The ConcretePrototypeFactory is initialized with prototypes of the Product and Product Components, such that there is no need to subclass it again when changes of the Product or Product Components classes are needed.

FIGURE 13. Product Prototype example: UML diagram

Pattern Application - Roles played by case-study classes

PrototypeFactory - a Factory class. It can be used to create a prototypical or a default ProductPrototype by initializing it with Prototypes of basic Product (Prototype) components. It defines the member functions to create the Product (Prototype) and the components of the Products. Each member function clones a prototype (Prototype1, Prototype2, Product) and then initializes it.

ConcretePrototypeFactory - a Factory subclass. The ConcretePrototypeFactory class is a Concrete Factory class which defines pointers to all prototypes (Part1Prototype, Part2Prototype, ProductPrototype), and Factory Methods which implement the cloning of the Prototypes, by invoking the Clone() functions. It should not be forgotten that the Factory Methods (Make ProductPrototype, Make Part1Prototype, Make Part2Prototype, Make ProductPrototype) are all virtual functions to allow the expansion of the Prototypes through subclassing.

The objects of the prototypes, such as instances of Part1Prototype, Part2Prototype, ProductPrototype, must support the Clone() operation. To support Clone operation, each prototype must have a copy constructor.

4.4.3 Performance Evaluation

The case study example uses the AbstractFactory pattern example as a framework for the implementation of the Prototype example. The measurements presented in Fig. 14 below, include the run-time performance of the AbstractFactory and the Prototype pattern, the total execution time representing the sum of the total execution time of both patterns. The total execution time and the overhead formulas for the Prototype pattern are:

Total execution time $T = F + V + C + N$

Overhead $O_{Prototype} = (F + V)_{Clone} + (C + N)_{Clone} + (F + V)_{Initialize}$

F - represents the execution time of all functions invoked during the execution, including the FactoryMethods,

V - represents the overhead introduced by the virtual functions of the Factory Methods,

N - represents the cost of dynamic objects creation with *new*,

F_{Clone} - represents the execution time of the function Clone,

N_{Clone} - represents the overhead of dynamic object creation by cloning itself with *new*,

C_{Clone} - represents the execution time of the cloned Constructor

The execution time distribution presented in Table 4 shows that the overhead of the Prototype pattern is larger than the overhead of other Creational patterns. This overhead is caused primarily by the cloning mechanism of the Prototype. The *new* function is invoked in the copy constructor of the PrototypeFactory, any time an object is created dynamically for Prototype initialization.

TABLE 4. Prototype - execution time distribution

T	F	V	N	C	O
4576	550	275	3426	325	1200
100%	12%	6%	75%	7%	14%

The main benefit of the Prototype pattern is its flexibility: it allows to add or remove products at run time, or to configure an application with classes dynamically. For these features, however, the price to pay is lower performance. There is no other way to add or remove products at run time except by using "dynamic object instantiation". A solution to this problem is to implement and use the *custom operator new* [Meyer98], which has higher performance than *default operator new*.

FIGURE 14. Prototype pattern example: call graph

4.5 Singleton

4.5.1 Background

This pattern ensures a class has one instance, and provides a global point of access to all clients. The Singleton responsible for keeping track of its instances, provides an easy way to access them. The pattern can be used when there is a need to have only one instance of a class, accessible to clients from a well-known access point, or when the sole instance is extensible by subclassing, and clients are able to use the extended instance without modifying their code.

Because the Singleton encapsulates its sole instance, it can have strict control over how and when clients access it. Being an improvement over global variables, it reduces name space size by avoiding global variables that stores sole instances.

When the Singleton needs to be deleted before the end of the program, and when there are multiple Singleton objects in the program, which depend on each other and the order of destruction is important, explicit destruction is the best alternative.

Structure

FIGURE 15.Singleton pattern: UML diagram

Main Classes

Singleton - defines an Instance operation that lets clients access its unique instance. Instance is a member function responsible for creating its own unique instance.

Sequence Diagram

FIGURE 16.Singleton pattern: event sequence chart

4.5.2 Case Study

The implementation assumes no subclassing. It is based on the pattern structure presented in Fig. 15, and uses the AbstractFactory pattern as framework. The UML diagram of the example is shown in Fig. 17.

FIGURE 17.Singleton with no subclassing: UML diagram

Pattern Application - Roles played by case-study classes:

ProductFactory - a concrete Singleton Class. The Instance member function is the only access to the Singleton, it returns the variable - instance which is initialized to zero. Product Factory's constructor is *protected*, to ensure that only one instance of the ProductFactory is created.

Client - The Client of the Singleton cannot instantiate the ProductFactory directly; this will ensure that only one ProductFactory is ever created.

Implementation characteristics:

- The design requires to protect the Constructor. This prevents accidental instantiation, which may lead to more than one instance.
- Only one instance of a *ProductFactory* is needed. The instance is available to code that builds any part of the Product. By making the *ProductFactory* a Singleton we make the Product object globally accessible without using global variables.
- No dependencies can exist between Singletons

4.5.3 Performance Evaluation

This section presents performance measurements for the case study example, which takes an Abstract Factory and makes it a Singleton. Because more than one pattern is used, the total execution time measurements include the execution time of all patterns, in this case the execution time of both patterns, Abstract Factory and Singleton. The overhead, however, is calculated only for the Singleton.

Based on these observations and on the callgraph presented in Fig. 18, the total execution time T and the overhead O are:

$$\text{Total execution time} \quad T = T_{\text{AbstractFactory}} + T_{\text{Singleton}} = F + V + C + N + D$$

$$\text{Overhead} \quad O_{\text{Singleton}} = (V + F)_{\text{Instance}} + (N + C + D)_{\text{Singleton}} + (V + F)_{\text{Register}}$$

$F_{Instance}$ - represents the call and execution time of the `Instance()` function,

F - represents the call and execution time of all non virtual functions,

V - represents the call and execution time of all virtual functions.

N - represents the execution time of all dynamic objects creations. with *new*, including Singleton,

C - represents the call and execution time of all constructors, including Singleton's constructor.

$N_{Singleton}$ - represents the time spent for Singleton object creation with *new*;

$C_{Singleton}$ - represents the execution time of Singleton's constructor.

$D_{Singleton}$ - represents the time to delete the Singleton object.

$(F + V)_{Singleton}$ - represents the execution time of Singleton's virtual functions

The critical path and the details of the measurements are shown in Fig. 18. The total cost was $T = 3705$ clock cycles, and the overhead $O = 885$ clock cycles.

The execution time distribution is summarized in Table 5, and as in the previous examples. it can be seen that most of the time is spent with the dynamic object creation with *new*:

TABLE 5.Singleton example - execution time distribution

T	I	F	V	N	C	O
3705	18	184	184	3077	242	885
100%	0.5%	5%	5%	83%	6.5%	23%

The overhead in this case is large. It includes dynamic Singleton object creation $N_{Singleton} = 853$ clock cycles. As with the Prototype, the Singleton object being created at run time, uses dynamic object instantiation with *new*. A solution to this problem is to use your own *operator new* to block the less efficient *default operator new*.

The detailed description of this solution can be found in [Meyer98].

4.6 Conclusions

The main benefit a designer can get from applying patterns is the flexibility of the design. The drawback is that potentially these patterns will provide more code and more objects.

Each of the five Creational Patterns introduces a new design concept:

- Factory Method: uses virtual functions, instead of constructors.
- Abstract Factory. Builder: pass a creator object as parameter.
- Prototype: parametrize with prototypical objects.
- Singleton: creates one object per program without using global variables.

A summary of Creational Pattern characteristics, and their real time performance, is presented in Table 6:

TABLE 6. Performance of Creational Patterns

Pattern Name	Description	Performance	
		Memory	Execution Time
Factory Method	Defines an interface for Product creation.	Small increase caused by virtual functions	Small overhead $O = (F + V)_{FM}$
Abstract Factory	An interface for creating families of related objects without specifying their concrete classes	Small increase caused by virtual functions and Factory Objects	Small overhead $O = (F + V)_{FM} + (C + D + N)_{AF}$ OR $O = (F + V)_{FM} + C_{AF}$
Builder	An interface that helps to build objects incrementally.	Small increase caused by virtual functions and Builder object	Small overhead $O = G + (F + V + C + N + D)_{Builder}$ OR $O = G + (F + V + C)_{Builder}$
Prototype	Creates new objects by cloning itself.	High mainly when complex structures are cloned	High overhead $O = (V + F)_{Clone} + (N + C)_{Clone} + (V + F)_{Initialize}$
Singleton	Ensures a class has one instance only.	Saves memory by reducing the number of instances of a class to one.	High overhead $O = (V + F)_{Instance} + (N + C + D)_{Singleton} + (V + F)_{Register}$

Performance measurements on Creational Patterns showed that dynamic object creation with *new* is very expensive, a lot more expensive than static (automatic) object creation. When invoked *new* allocates memory first, and then calls one or more constructors. Similarly, when *delete* is invoked, one or more destructors are invoked and then the memory previously allocated with *new*, is deallocated. Two of the Creational Patterns, Prototype and Singleton, have high overhead. They can be used in applications with no time constraints, but where extensibility and flexibility are important. If the Prototype and Singleton patterns are used in high performance applications, a solution to improve their performance, would be to use a customized operator *new* as proposed in [Meyer98].

The other three Creational patterns, the Builder, the Abstract Factory and the Factory Method, have small overheads. They can be used in applications, where performance is a concern simply because dynamic object creation is not absolutely necessary and can be replaced with static object creation.

Note: Dynamic Product object creation was not included in the overhead of the Creational Patterns, mainly because the reference implementation was an OO implementation, using dynamic Product object instantiation.

Chapter 5

5.0 Structural Patterns

Structural patterns define how classes and objects are composed to form larger structures. There are two types of Structural Patterns

- Structural class patterns relying on single or multiple instances providing a static view
- Structural object patterns relying on object composition, at run time, providing a dynamic view

Structural class pattern characteristic is composition through inheritance. The list of structural class patterns as described in [Gamma95]:

- Class version of the **Adapter** makes one interface conform to another by using inheritance.

Structural object pattern characteristics is the composition through objects. The list of structural object patterns is:

- Object version of the **Adapter** - combines classes with different interfaces
- Bridge** - decouples abstraction from its implementation
- Composite** - builds complex structures
- Proxy** - provides a representative for an object
- Decorator** - adds responsibilities to objects dynamically
- Flyweight** - efficiently shares fine-grained objects
- Facade** - provides a simple interface to a complex system

Structural patterns look very similar, because they all rely on the same mechanism: single or multiple inheritance, and object composition. Even if they look very similar, what differentiate them are their intent.

5.1 Adapter

5.1.1 Background

The pattern converts the interface of a Class into an interface needed by the clients. It helps classes with incompatible interfaces work together. There are two types of Adapter patterns: Class Adapter and Object Adapter.

The Class Adapter pattern is used when a design needs to use an existing class, and its interface does not match the one is needed, or when a design needs to create a reusable class that cooperates with classes that don't necessarily have compatible interfaces.

The Object Adapter pattern is used when a design needs to use existing subclasses, but it is impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class. Both types of Adapters inherit from the Target, but they differ in how they forward messages to the Adaptee.

Structure

Class Adapter uses multiple inheritance to adapt one interface to another. On one branch inherits publicly from the target and forwards back to the actual target portion of itself. On the other branch inherits the implementation.

FIGURE 19. Adapter Class: UML diagram

Because the Class Adapter adapts Adaptee to Target by committing to a concrete Adapter calls, a Class Adapter will not work when trying to adapt a Class and its subclasses. In this configuration, Adapter overrides some of Adaptee's behaviour, since Adapter is a subclass of the Adaptee.

Object Adapter relies on object composition: to combine classes with different interfaces.

FIGURE 20.Object Adapter: UML diagram

Main Classes

Target - an interface to the Client

Client - instantiates Adapter objects, and calls operations on this objects

Adaptee - the interface that is adapted

Adapter - adapts the Adaptee to the Target. Adapters vary in the amount of work they do to adapt Adaptee to the target interface. The amount of work Adapter does depends on how similar the Target interface is to Adaptee's.

Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

Sequence Diagram

FIGURE 21. Adapter pattern: event sequence chart

5.1.2 Case Study

The Adapter pattern case study examples are based on the pattern structures described in Fig. 20 and 21.

Class Adapter (Fig. 22)

The Class Adapter, inherits privately from the actual TargetImplementor class and forwards messages to the actual portion of itself. By applying the Adapter pattern in this example two issues are solved: type intrusiveness, and flexibility. The Adapter inherits from the TargetImplementor and the Adaptee.

With this implementation not only the Adaptee does not inherit from TargetImplementor, as in the standard OO implementation, but the complexity of back pointers to the Target is moved inside the Adapter.

The Adapter contains now pointers to each Target and use these pointers to determine which target was requested. Once the Adapter knows this, it can tell the Adaptee directly which target sent the request.

FIGURE 22. Class Adapter UML diagram

Pattern Application - Roles played by case-study classes:

Target and TargetImplementor defined as two distinct classes to allow Adapter to inherit only from the class TargetImplementor class, implementing Target specific functions.

Each Target type needs its own Adapter. But it solves coupling because the Adaptee no longer needs to know about Targets, and Targets know nothing about Adaptee. Also it solves type intrusiveness as the Adapter no longer inherits from the Client, and flexibility as different adapters could adapt the Client interface to call functions or function objects.

An **Object Adapter** example is presented in Fig. 23.

The Object Adapter pattern is also called External Polymorphism. This version of the Adapter uses template Adapter is templated by the type of Adaptee being adapted. By using template, there is no need to write separate adapters for each kind of Adaptee, adapters being automatically generated from the template. This makes maintenance of the Adapter code and adding of new kinds of Adaptees much easier. The addition of this pattern resolved the issue of having to create manually new adapters. The creation of adapters is automated. However, the flexibility is not resolved. No single template is able to call back to functions, function objects, and member

functions. And even more, the coupling is not resolved either. The Adaptee must know now about targets. The main change consists of replacing the Adapter class with ExternalPolymorphism class. The ExternalPolymorphism, is a more general Adapter.

FIGURE 23.Object Adapter: UML diagram

External Polymorphism class - is an Adapter

If there are only one or two types of Clients that need to work with the server without inheriting from the client base class, the Adapter class is the solution. If there are many kinds of Clients that need adapting, ExternalPolymorphism is the solution, as long as all the Clients want the same method called.

5.1.3 Performance evaluation

Standard Object Oriented implementation of the Target-Adapter example is time efficient, but requires extra space to store Client pointers. It is also very limited because the Client, and the Server, are tightly coupled. A solution to this problem is provided by the Adapter pattern

FIGURE 24.Standard implementation of the Target-Adapter

This analysis refers to both Class Adapter and Object Adapter.

Class Adapter

The Class Adapter decouples the Server from the Client at the expense of some time efficiency. There are two main overheads added with this implementation: one caused by the trading of a direct method call for a polymorphic method call, and the other caused by the addition of another level of indirection. However, the overhead inserted by the class Adapter pattern is minor.

Based on the measurements shown in Fig. 25 the total execution time and the overhead are:

$$\text{Total execution time} \quad T = C + F$$

$$\text{Overhead} \quad O_{\text{ClassAdapter}} = F_{\text{Request}} + C_{\text{Adapter}}$$

In the overhead calculation, F_{Request} represents the polymorphic method execution time, and C_{Adapter} represents the execution time of the Adapter constructor.

The execution time distribution of the Adapter example is shown below

TABLE 7. Adapter - execution time distribution

T	N	F	C	O
113	-	79	34	39
100%	0%	70%	30%	28%

Object Adapter

With class Adapter we have to manually create an Adapter for each kind of Client. The Object Adapter or ExternalPolymorphism provides the mechanism to automate Adapter.

The Object Adapter may have a negative impact on efficiency, because it could cause code bloat from template instantiation. Since the template is small, the efficiency is not affected.

The overhead added by the Object Adapter is still minor. It can be calculated with the following formula

$$\text{Overhead} \quad O_{\text{ObjectAdapter}} = F_{\text{Request}} + C_{\text{ExternalPolymorphism}}$$

The time distribution for the ExternalPolymorphism example is:

TABLE 8. ExternalPolymorphism - execution time distribution

T	N	F	C	O
129	-	93	36	55
100%	0%	71%	29%	42%

The total execution time as well as the overhead are larger in the Object Adapter's case, due to a new level of indirection added by the ExternalPolymorph class when invoking TargetImplementor method. Both pattern variations, Class and Object Adapter, use inheritance to adapt interfaces.

One inheritance branch is used to inherit the target interface and the other branch to inherit the implementation. the Adapter interface. In single inheritance each object whose class declares virtual functions carries a virtual table *-vtbl-*, and a hidden data member *-vpri-* that points to the virtual table for that class.

The cost of virtual function has two components: one component consists of the space to set aside for the virtual table, and the second component consists of the cost to pay for an extra pointer inside each object that is of a class containing virtual functions.

With multiple inheritance the per-class and per-object space overhead for virtual function and the run-time execution cost increases.

In conclusion, multiple inheritance increases the object size, per class data, and slightly the execution time. Multiple inheritance stays at the base of the Class Adapter pattern; for this type of overhead it is difficult to find a solution.

FIGURE 25. Class Adapter example call graph

FIGURE 26. Object Adapter example: call graph

5.2 Bridge

5.2.1 Background

Inheritance binds an implementation to the abstraction permanently; this makes it difficult to modify, extend, and reuse abstractions and implementations independently. The Bridge pattern addresses this problem by putting the abstraction and its implementation in separate Class hierarchies. There is one Class hierarchy for abstraction interfaces and a separate hierarchy for the implementation. The main role of the pattern is to decouple the abstract class from the implementation class, providing the means for the implementation to be configured at run time.

Structure

FIGURE 27. Bridge pattern: UML diagram

Main Classes

Abstraction - an abstract interface, maintaining a reference to Implementor's object. Its role is to forward client's requests to the Implementor object.

RefinedAbstraction - extension of the Abstraction interface.

•**Implementor** - an interface that provides the primitive operations needed by the Abstraction interface to define the high-level operations. It is difficult sometime to have a good design when there are many implementors. The difficulty consists in finding the place and the time when to instantiate the implementor. Changing an Implementor Class doesn't require recompiling the Abstraction Class and its clients.

ConcreteImplementor - implements the Implementor interface and defines Implementor's concrete implementations.

This pattern improves extensibility, by allowing to extend the Abstraction and Implementor hierarchies independently and encourages layering that can lead to a better-structured system.

Sequence Diagram

FIGURE 28. Bridge pattern: event sequence diagram

5.2.2 Case study

In this example the Bridge pattern is used for printing expression trees. The Abstraction Class defines the Implementor abstraction for Client applications. Abstraction class maintains a reference to an Implementor, the abstract Class that declares an interface to the underlying system. Subclasses of Implementor define the different kinds of nodes the application might use.

The Bridge pattern example is presented in Fig. 29.

FIGURE 29. Bridge pattern example: UML diagram

Note how this implementation decouples the Abstraction interface for printing from the Implementor subclass implementations, i.e. the Abstraction interface is fixed whereas the Implementor varies. However, clients need not be concerned about the variation.

5.2.3 Performance Evaluation

The typical method for implementing the Abstraction in C involves the use of struct/union to represent data structures, and of switch statement and recursive function to build and evaluate the Abstraction. This approach has many limitations. Some limitations are: tight coupling between nodes and edges in union representation, any small change ripples through the entire design; waste of space with respect to structs and union (problem which becomes worse the bigger the size of the item becomes); and finally, the overhead is high because of the switch statement. By contrast, the implementation using patterns solves all the problems found with the typical implementation presented above.

The case study example includes two patterns:

- The Factory Pattern to initialize Implementor subclasses,
- The Bridge Pattern to shield the use of inheritance and dynamic binding, which solved the problem of printing subtrees without revealing their types.

The pattern provides a stable, uniform, interface that is closed and open in the same time: closed to prevent direct code changes, and open to allow extensibility. In this example the Bridge pattern is used for printing expression trees.

```
void Abstraction :: print (ostream & os) const
{ this -> implementor -> print (os); }
```

This implementation decouples the Abstraction interface for printing from Implementor subclass implementation, i.e. the Abstraction interface is fixed, whereas the Implementor varies. However, clients need not be concerned with Implementor variations.

The overhead added by the Bridge pattern is essentially caused by the Abstraction class. It is the result of Abstraction's initialization and virtual functions utilization. Overall the Bridge pattern may be more efficient than the large switch statements or if-the-else chains of the typical functional implementation in C.

The total execution time and the overhead are evaluated by the formulas:

Total execution time $T = C + D + F + N$

Overhead $O_{Bridge} = (C + N + D)A_{Abstraction} + F_{Abstraction}$

where N includes the cost of ConcreteImplementor's objects instantiation. In a standard OO implementation, the same cost would have to be paid for the instantiation of the ConcreteImplementor's objects. For this reason this part of N is not included in the overhead. $C_{Abstraction}$ represents the instantiation cost of the Abstraction class, $F_{Abstraction}$ represents the execution cost of all functions of the Abstract class. It includes the cost of using virtual function, and an extra level of indirection.

The overhead components are all the result of adding the Abstraction class in the pattern's structure. The total execution time T and the overhead O are: $T = 3534$ clock cycles, $O = 614$ clock cycles, the overhead representing 17% of the total execution time. The time distribution for the pattern is shown in Table 9.

TABLE 9. Bridge pattern execution time distribution

T	C	D	F	N	O
3534	419	106	36	1972	614
100%	14%	31%	1%	55%	17%

The Bridge pattern mechanism does not rely on dynamic instantiation of the Abstraction object. The overhead is moderate.

FIGURE 30. Bridge pattern example: call graph

5.3 Composite

5.3.1 Background

With Composite pattern the hierarchies are represented by the structures of objects and compositions of objects uniformly. The pattern defines Class hierarchies consisting of primitive objects and composite objects. Primitive objects can be composed into more complex objects, which in turn can be composed. Wherever clients code expects a primitive object, they can also take a composite object.

Structure

FIGURE 31. Composite pattern: UML diagram

The Composite pattern describes how to use recursive composition so that clients don't have to make distinction between primitive and container objects. Composite pattern can be found in almost all object-oriented systems.

Main Classes

Component - is the abstract interface to access and manage all the objects in the composition, all the child component and the component's parent in the recursive structure. It implements default behaviour for the interface common to all classes.

ComponentLeaf- is a Leaf, and represents the objects with no children in the composition and defines behaviour for primitive objects in the composition

CompositeComponent - is the class which stores child components, defining behaviour for components with children.

CompositeComponents - are child components. They are all Leafs.

Client - uses the Component class to interact with the objects in the composition structure. If the recipient is a Leaf, then the request is handled directly. If the recipient is a Composite, then it usually forwards requests to its child components, performing additional operations before and/or after forwarding.

Sequence Diagram

FIGURE 32. Composite pattern: event sequence chart

5.3.2 Case Study

The case study example describes a hierarchy of Components. Each CompositeComponent is composed of simple lower level parts, which are the ComponentComponents. Component class defines an interface for all elements in the hierarchy. It contains methods that return the attributes of an component. The subclasses, the concrete components, implement specific operations for specific types of components. To access its parts, Component class defines a CreateIterator method to create an Iterator that will traverse the list. The CompositeComponent is the base class for the definition access and management of a set of subcomponents.

5.3.3 Performance Evaluation

To store the children the Composite pattern may use different types of data structures: linked lists, trees, arrays, and hash tables. For efficiency reasons, the data structure is selected in function of the amount of information that has to be stored.

The case study example presents a CompositeEquipment with a List of components. A ListIterator is defined to traverse the List. An instance of the ListIterator is returned by the function Component::CreateIterator(). The call graph from Fig. 33, and the total execution time T , include the time of both patterns, Composite and Iterator. The overhead, however refers only to the Composite pattern.

$$\text{Total execution time} \quad T = F + V + C + N$$

$$\text{Overhead} \quad O_{Composite} = (F + V)_{Composite} + (C + N + D)_{Composite}$$

The overhead is small and is mainly created by one more layer of indirection introduced by the Composite class. It has three components: $F_{Composite}$ the overhead added by one level of indirection introduced by the Composite, $V_{Composite}$ the overhead of the virtual functions used in the Composite - Component relations, and $C_{Composite}$ the Composite object static instantiation. When using dynamic instantiation, the overhead will include $N_{Composite}$, which is not the case of the example. The time distribution is shown in Table 10.

TABLE 10. Composite pattern execution time distribution

T	F	V	C	D	N	O
2529	10	5	201	568	1744	122
100%	0.4%	0.2%	8%	21.4%	70%	4.5%

The performance may be degraded if there is a need to traverse or search the compositions very often. A solution in this case, is to cache traversal or search information about the Composite's children inside the Composite.

In the case study example only one traversal was executed. The intention was to show strictly the overhead added by the Composite, and not the effects of Composite-Iterator interactions.

FIGURE 33. Composite pattern example: call graph

5.4 Decorator

5.4.1 Background

Provides a flexible mechanism to attach new responsibilities to an object (not to an entire class). The OO way to add responsibilities or to extend functionality consists in using inheritance. But inheritance is inflexible, because the choice of the function is made statically. A better approach is to enclose the component in another object that adds the function. The enclosing object is called a **decorator**. The presence of the decorator is transparent to the component's clients. Transparency allows to nest decorators recursively, thereby a large number of added responsibilities is possible. It is a flexible approach to adding functionality to an object. Rather than put functionality in classes higher up in the hierarchy, add functionality as needed.

Structure

FIGURE 34. Decorator pattern: UML diagram

Main Classes

Component - an interface adding responsibilities to objects dynamically

ConcreteComponent - a subclass of the component for adding new responsibilities

Decorator - keeps a reference to its **ConcreteComponent** object and performs decoration operations on it. The **Decorator** may optionally perform additional operations before and after forwarding the request.

ConcreteDecorator - a subclass of the **Decorator**, it adds operations for specific functionality.

A major drawback of the Decorator is that it may lead to many little objects composed in complex ways, which can make the code hard to understand and debug.

Sequence Diagram

FIGURE 35. Decorator pattern: event sequence diagram

5.4.2 Case Study

The Decorator example presented here decorates the **ConcreteComponents**. The **Decorator** is like a shell around the object, it decorates the **ConcreteComponent** as needed. Two **ConcreteComponents** are instantiated: one, **ConcreteComponent1** has decoration2, 3, and 4; the second, **ConcreteComponent2** has decoration 1, and 4. Fig. 34 explains decorations (shells) on objects and the recursive calls to **Operation()** method.

5.4.3 Performance Evaluation

The Decorator gives options for dynamically choosing functionality in addition to the core functionality of ConcreteComponent. All Decorators add new layers of indirection, which mean loss of efficiency. Efficiency is also affected by the Decorator's mechanism of adding functionality dynamically. The non-pattern implementation of the Component example is shown in Fig. 36

FIGURE 36. Non-pattern implementation of the Component example

In the non pattern implementation, the instantiation of a Component requires eventually one dynamic creation of the Component, while with the Decorator pattern, each decoration is added dynamically at run time.

Without Decorator	With Decorator
<pre> Component *ConcreteComponent1 = new ConcreteComponent1 (); ConcreteComponent1 -> Operation(); </pre>	<pre> Component *ConcreteComponent1 = new ConcreteDecorator2 (new ConcreteDecorator3 (new ConcreteDecorator4 (new ConcreteComponent ())) , ConcreteComponent1 -> Operation(); </pre>

The total execution time and the overhead of the Decorator pattern is illustrated by the formulas:

Total execution time $T = F + N + C + D$

Overhead $O_{Decorator} = (C + N + D)ConcreteDecorator$

The call graph of the Decorator example is shown in Fig. 37: 79% of the execution time is spent in dynamic objects creation. For the Decorator, the total execution time is 2492 clock cycles, and the overhead is 1225 clock cycles. The execution time distribution is shown in the Table 11.

TABLE 11. Decorator example execution time distribution

T	F	N	C	D	O
2492	174	1972	195	151	1225
100%	7%	79%	8%	6%	50%

The high overhead is the result of “decorating” a Component. The Decorator’s mechanism consists in adding objects dynamically in a chain of decorations. Similar to Prototype, Singleton, and Proxy patterns, the Decorator pattern uses dynamic object creation with *new*. Dynamic object instantiation is required because the Decorator allows to select functionality or add decorations at run time. A solution to high overheads caused by dynamic instantiation of the objects, is to implement a customized *operator new* [Meyer98].

FIGURE 37. Decorator pattern example: call graph

5.5 Facade

5.5.1 Background

Facade is a unique interface to other interfaces in a system, which helps minimize the communication and dependencies between subsystems. It is a public interface that can be accessed by all clients.

Structure

FIGURE 38. Facade pattern functional diagram

Main Classes

Facade - has the knowledge to map the request with the originator and the destination.

Subsystem classes - implement part of the system functionality, handling the functions and the requests received from the Facade object. The subsystems have no knowledge of the Facade, they keep no reference to it.

The Facade forwards the requests received from the Client classes to the appropriate subsystem objects. Most of the time the subsystem's objects perform the actual work, however sometime, the

Facade may have to do work of its own to translate its interface to subsystem interfaces. Clients that use the Facade don't have to access its subsystem objects directly.

The pattern decouples Clients from subsystem components, making the subsystem easier to use, and allowing to vary the components of the subsystem without affecting its clients. The coupling between Clients and subsystem can be reduced even further by making Facade an abstract class with concrete subclasses for different implementations of a subsystem.

Sequence Diagram

FIGURE 39. Facade pattern: event sequence chart

5.5.2 Case Study

The case study example consists of using a Facade on a system made of six other subsystems that implement different functions. Each subsystem is represented by a class: System1, System2, System3, System4, and System5. Most of the Facade's clients do not need to access these classes directly, also they do not need to know details of each subsystem functional implementation. To provide a high-level interface that decouples the clients from these classes, the subsystem also includes a Facade class, a unified interface to all the subsystems.

The Facade class provides clients a simple interface to all the subsystems, without hiding the lower-level class completely. The Facade example is based on the pattern structure from [Gamma95]. The configuration implemented in the case study uses several patterns: Composite for System4, Visitor for System5, Builder for System3, and Facade.

FIGURE 40.Facade example UML diagram

5.5.3 Performance Evaluation

When the client and the subsystems are implemented independently, Facade can help save time, by minimizing recompilation when subsystems are changed. It is assumed subsystem's compilation dependencies with Facade are minimal. The main drawback of using Facade is the impact on the run-time performance. From the functions call graph presented in Fig. 41, it is obvious that some loss of efficiency is introduced with the layer of indirection added by the Facade class. The total execution time and the overhead formulas are:

$$\text{Total execution time} \quad T = F + C + N + D$$

$$\text{Overhead} \quad OFacade = F_{Facade} + (N + C + D)_{Facade}$$

The time distribution is shown in Table 12.

TABLE 12. Facade- execution time distribution

T	F	N	C	D	O
1169	209	974	50	7	860
(100%)	(15%)	(80%)	(5%)	(<<1%)	(75%)

The overhead is entirely caused by the Facade class, and it has the following components: Facade object creation with *new*, N = 808; Facade's Creator execution, C = 7; and Facade's member function invocation, F = 48. Static instantiation of the Facade object [Copl96] reduces the overhead. The formula of the overhead becomes:

$$\text{Overhead} \quad OFacade = F_{Facade} + C_{Facade}$$

Although, in the case study example, the overhead seems large it does not correspond to the real situation. The Facade class, in a real system, is the general interface of many subsystems; therefore, its overhead represents a small percentage of the total execution time of the entire system, mainly when the Facade's objects is statically instantiated.

FIGURE 41. Facade pattern example: call graph

5.6 Flyweight

5.6.1 Background

Flyweights are objects that can be used in multiple contexts at the same time. A Flyweight acts as an independent object in each context, being unable to make assumptions about the context in which it operates. The Flyweight can have **intrinsic** and **extrinsic** states. An intrinsic state is stored in the Flyweight and consists of information that is independent of the context, making it sharable. An extrinsic state depends on and changes with the context and therefore cannot be shared. Client objects are responsible for passing extrinsic state to the Flyweight when needed.

Structure

FIGURE 42. Flyweight pattern: UML diagram

Main Classes

Flyweight - is an interface that enables sharing and is used to receive and act on extrinsic state. It is a shared object that can be used in multiple contexts simultaneously.

SharedConcreteFlyweight - implements the Flyweight interface; it adds storage for intrinsic state only, intrinsic states being independent of the ConcreteFlyweight object's context.

UnsharedConcreteFlyweight - there are unshared Flyweight subclasses. UnsharedConcreteFlyweight objects have ConcreteFlyweight objects as children which are used to store the intrinsic state of the Flyweight.

FlyweightFactory - a factory of flyweight objects, which creates, manages, and ensures that Flyweights are shared properly. For a client request, the FlyweightFactory objects supplies an existing instance or creates one, if none exists.

Client - keeps a reference to Flyweight(s) and stores or computes the extrinsic state of Flyweight(s). Clients pass this state to the Flyweight when they invoke its operations.

Clients should not instantiate ConcreteFlyweights directly, they must obtain ConcreteFlyweight objects exclusively from the FlyweightFactory object to ensure they are shared properly.

Sequence Diagram

FIGURE 43. Flyweight pattern: event sequence chart

5.6.2 Case Study

Consider the implementation of a document formatter/editor. The OO implementation of a document editor uses objects to represent tables and figures. To increase flexibility they are promoting the instantiation of one object per character. The drawback of such implementation is the cost. For even a small size document, it would create a large number of character objects which will consume a lot of memory, and will require an unacceptable run-time overhead. A solution to this problem. Fig. 44, is the Flyweight pattern which allows objects to be shared among many contexts. Because in general, the number of different character objects is far less than the number of characters in the document, the total number of objects is substantially less than what a standard OO implementation would use. A document with characters using the same font and color does not need to allocate more than 100 character objects regardless of the document's length. In real applications this number will not increase as most documents use no more than 10 different font-color combinations. For this case, an object abstraction for individual character is recommended. The case study example is a direct implementation of the structure presented in Fig.42.

Pattern Application - Roles played by case study classes

The **Flyweight Factory** class, has two methods: CreateFlyweight to create new flyweights and CreateUnshared Flyweight to create unshared flyweights.

The **Flyweight**, an abstract class for flyweight objects. Operations that may depend on extrinsic state are passed to them as parameters.

The Flyweight Context is a part of the **Client**, which stores the attribute extrinsically, acting as a repository of extrinsic states.

The **ConcreteFlyweight** is a shared Flyweight.

The **UnsharedConcreteFlyweight** can be a Composite having a set of attributes.

In this example the Flyweight pattern is used to maintain the attributes associated with the Flyweight. To avoid memory allocation for each attribute of every Flyweight, the attribute is stored extrinsically by the Client (FlyweightContext attributes) object. The FlyweightContext starts to behave as a repository of extrinsic states (or attributes), in the same time maintains a mapping

between a Flyweight and its attributes in different contexts. For any operation which needs to know the Flyweight's attribute in a context, a FlyweightContext will be passed to it as a parameter. The operation can now query the FlyweightContext for the attribute in that context. The context depends on the Flyweight's location in the Flyweight structure. During the traversal of the structure, the FlyweightContext is updated with the current position in the structure. A BTree can be used to store the Flyweight to attribute mapping. BTree is updated in response to attribute changes and whenever Flyweights are added to or removed from the flyweight structure.

When the FlyweightContext is queried for the attribute of the current Flyweight, it descends the BTree, until it finds the attribute for the current index. Because the frequency of attribute changes is relatively low, the tree stays small relative to the size of the Flyweight structure. This keeps storage costs down without an inordinate increase in look-up time. Look-up time in this scheme is proportional to the attribute change frequency. Worst-case performance occurs when an attribute change occurs on every Flyweight, but that's unusual in practice.

5.6.3 Performance Evaluation

The Flyweight is a shared object that can be used by multiple Contexts simultaneously, by handling two states: the intrinsic state and the extrinsic state. The intrinsic state is stored in the Flyweight object, while the extrinsic state is stored by the Client and passed to the Flyweight object when the Client invokes it. To ensure the Flyweights are properly shared, Clients should not instantiate concrete Flyweights directly but obtain them from the FlyweightFactory. The “Shared Flyweight” model has an important benefit: it saves memory. Memory storage savings are facilitated by the reduction in the total number of instances that come from sharing, and from the smallest amount of intrinsic state per object. However, the “Shared Flyweight” model may increase the run time cost. The case study is an example of the Composite objects implemented as Flyweights. The total cost and the overhead can be calculated with the formulas:

Total execution time **$T = F + C + N + D$**

$$\text{Overhead} \quad \mathbf{O}_{Flyweight} = \mathbf{F}_{ClientOperation} + (\mathbf{N} + \mathbf{C} + \mathbf{D})\mathbf{ConcreteFlyweight} + (\mathbf{F} + \mathbf{V})\mathbf{ConcreteFlyweight}$$

where *FClientOperation* is the time spent by the Client's function to find an attribute for the current Flyweight object, and *CConcreteFlyweight* represent the execution time of the ConcreteFlyweight Constructors. The real-time performance decreases when the frequency of changing the state increases. It is important to mention, that the initialization of the Flyweights may be expensive. In this example, the cost of the initialization of an array of flyweights was 1554 clock cycles. The Performance Evaluation and the critical path are shown in Fig. 44, the execution time distribution in Table 13.

TABLE 13. Flyweight - execution time distribution

T	F	C	D	N	O
2740	46	1591	15	1088	180
100%	1.5%	58%	0.5%	40%	6.5%

ConcreteFlyweight object creation always happens at run time, it cannot be replaced with static object creation. However, an improvement can be obtained by using a customized *operator new* as described in [Meyer98].

FIGURE 44. Flyweight pattern example: call graph

5.7 Proxy

5.7.1 Background

The Proxy pattern provides a representative for another object to have controlled access to it.

Controlled access to an object is sometimes required in a design, when the design that needs to defer the cost of its creation and initialization until there is a need to use it. The Proxy design pattern facilitates the communication of the Client with a representative rather than to the component itself. It can serve many purposes: enhanced efficiency, easier access and protection from unauthorized access.

The representative --called a *Proxy*-- offers the interface of the component but performs additional pre and postprocessing such as access-control checking, or making read-only copies of the original. The Client invokes the functionality of its original in an indirect way by accessing the Proxy. The Client does not have to change its calling behaviour and syntax from what it uses to call local components.

Structure

FIGURE 45.Proxy pattern: UML diagram

Main Classes

Proxy offers the same interface as the original, and ensures correct access to the original. To achieve this the proxy maintains a reference to its original. It controls access to the original and may be responsible for creating or deleting it. Usually there is a one-to-one relationship between the proxy and the original, though there are exceptions to this rule for *Remote* and *Firewall* proxies, two other variants of the pattern.

Abstract Original provides the interface implemented by the proxy and the original. In a language like C++, with no table difference between subtyping and inheritance, both the proxy and the original inherit from the abstract original. Clients use this interface to access the original.

Original - is the real object.

Sequence Diagram

FIGURE 46.Proxy example: event sequence chart

Preprocessing involves actions like looking up the address of the original or checking in a local cache if the requested information is locally available. Postprocessing involves actions like caching for results, calling the destructor of the Original, or reassigning a look.

Variants

There are eight variants of the generic Proxy pattern as stated in [Gamma95].

- A *Remote Proxy* encapsulates and maintains the physical location of the original. Remote Proxy Decouples clients from the location of server components. By putting all location information and addressing functionality into a Remote Proxy, clients are not affected by migration of servers or changes in the networking infrastructure. This allows client code to become more stable and reusable.

- A *Protection Proxy* protects the original from unauthorized access.

- To implement a *Cache Proxy*, extend the proxy with a data area to temporarily hold results.

- A *Synchronization Proxy* controls multiple simultaneous client access.

- A *Counting Proxy* can be used for collecting usage statistics, or to implement a well-known technique for automatically deleting obsolete objects - reference counter.

- A *Virtual Proxy*, also known as *lazy construction*, assumes that an application references secondary storage.

- A *Firewall Proxy* subsumes the networking and protection code necessary to communicate with a potentially hostile environment. Usually the firewall proxy is implemented as a daemon process on a firewall machine, which can also be referred to as a 'proxy server'.

- A *Smart Reference* is a replacement for a bare pointer that performs additional actions when an object is accessed. The Smart Reference is a combination of the aspects of Counting, Virtual, and Synchronization Proxies.

One of the benefits of Proxy is to provide enhanced efficiency and lower cost. The Virtual Proxy and Cache Proxy variants help to implement a 'load-on-demand' strategy. This allows you to avoid unnecessary loads from disk and usually speeds up your application. All Proxy variants provide separation of housekeeping code from functionality.

5.7.2 Case Study

The case study presents the Virtual Proxy variant. The Virtual Proxy pattern makes the clients of a component communicate with a representative rather than with the component itself. Introducing such a placeholder can serve many purposes including enhanced efficiency, easier access and protection from unauthorized access.

The following class diagram illustrates the structure of the Proxy example.

FIGURE 47.Proxy example: UML diagram

Pattern Application - Roles played by case-study classes:

The **Client** has access to the embedded original through the interface defined by the **AbstractOriginal** class.

The **Proxy** class maintains a reference to the **Original**, and stores a reference to the real original instance. The reference is passed as an argument to the **OriginalProxy** constructor; it is not valid until the Proxy instantiates the **Original**.

Methods:

GetExtent - forwards the request to the image only if it's instantiated; otherwise imageProxy returns the extent it stores.

GetOriginal - checks if the Original is instantiated; if the Original does not exist it creates a new Original.

5.7.3 Performance Evaluation

There is a high degree of gain in efficiency for the Virtual and the Cache variants of the pattern. This gain in efficiency comes as a result of a feature which allows to copy parts of the original on demand. Copying a large object can be expensive. The Proxy allows to copy an object only when the object is modified, thus reducing the cost.

All Proxy variants have a drawback: an additional layer of indirection that translates in loss of efficiency.

A call graph of the Proxy example presented in Fig. 48 and the formulas for the total execution time and the overhead are:

Total execution time $T = F + C + N + D + F_{System}$

Overhead $O_{Proxy} = (C + N + D)Proxy + F_{ProxyService} + (N + D)Original$

The overhead is caused by the level of indirection added by the Proxy. This level of indirection translates in Proxy's dynamic object creation and its functions utilization.

The real time distribution of the case study example is presented in table 14; it shows that the Proxy's overhead represents 35% of the total execution time, most of it being caused by Original Proxy object and Original object instantiations with *new*.

TABLE 14. Proxy - execution time distribution

T	F	C	N	F _{System}	D	O
2243	37	69	1516	621	0	794
100%	2%	3%	88%	7%	0%	35%

Similar to Facade and Composite patterns, the Proxy object was created in the function *main* using dynamic instantiation with *new*. Dynamic instantiation of the Proxy object is very expensive.

A solution to this problem consists in replacing dynamic instantiation with static instantiation [Cop196]. The Proxy object does not require dynamic object creation at run time. This is why static Proxy object creation is preferable.

However, OriginalProxy creates the Original object at run time, dynamically. This cannot be replaced with static instantiation. An optimized solution is suggested in [Marti98]; it consists in implementing a customized operator *new*.

FIGURE 48.Proxy example: call graph

5.8 Conclusion on Structural Patterns

Structural patterns may look very similar, but what distinguishes them are their intent.

- Adapter and Bridge both use indirection but for different reasons

Adapter: to match an interface the client expects to the one Adapter provides, it is used late

Bridge: to provide a client access to different implementations in a transparent way, to provides stability to clients in presence of implementation evolution. it is used early

- Composite, Decorator both use recursive composition also for different reasons

Composite: for bringing apparent uniformity to a family of arbitrary complex structures

Decorator: for adding responsibilities to an object in an open-ended way; Decorator uses object composition to avoid explosion in number of classes resulting from using subclassing, and allow for dynamically adding responsibilities

- Proxy and Decorator shield an object through an identical interface but Proxy is intended to stand-in for the object

- Adapter and Proxy are related patterns

Adapter: gives a different interface to its Adaptees

Proxy: presents exactly the same interface as the real object

- Decorator and Proxy have similar implementation but different purpose

Decorator: adds functionality

Proxy: control access

A summary of Structural Pattern characteristics and their real time performance, is presented in Table 15 below:

TABLE 15. Performance of Structural Patterns

Pattern Name	Description	Performance	
		Memory	Execution Time
Adapter	Converts the interface of one class into that of another.	Small increase caused by Adaptor's object and virtual functions	Small Overhead: $O = F_{Request} + C_{Adapter}$
External Polymorphism	Has the creation of the Adapters automated.	Small increase caused by EP object and virtual functions	Small Overhead: $O = F_{Request} + C_{ExtPoly}$
Bridge	An interface that decouples Abstraction from its implementation.	Small increase caused by virtual function and Abstraction's object	Medium Overhead $OB = (F + V)Abstraction + (C+N+D)Abstraction$
Composite	Creates object composites	Small increase caused by virtual function and Composite's object	Medium Overhead: $O = (F + V)Composite + (C+N+D)Composite$ OR Small Overhead $O = (F + V)Composite + C_{Composite}$
Facade	A unique interface for other subsystems.	Small increase caused by Facade's object	Small Overhead: $O = F_{Facade} + C_{Facade}$
Decorator	Attaches additional responsibilities to an object dynamically.	Large increase caused by many Decorator objects	Large Overhead $O = (C+N+D)Decorator$
Flyweight	Uses sharing to support large numbers of fine-grained objects efficiently.	Saves memory (see Description).	Small Overhead: $O = F_{ClientOperation} + (F+V+C+N+D)ConcreteFlyweight$
Proxy	Provides a surrogate or placeholder for another object to control access to it.	Small increase caused by Proxy's object	Small Overhead when using static Proxy object creation $O = C_{Proxy} + F_{ProxyService} + (N+D)Original$

Chapter 6

6.0 Behavioural Patterns

Behavioural patterns handle both the patterns of objects and the communication between the objects. These patterns are used in the designs that deal with partitioning of behavioural responsibilities. There are two types of behavioural patterns:

- behavioural class patterns which use inheritance between classes
- behavioural object patterns which use object composition instead of inheritance to partition the behaviour, or others which encapsulate behaviour in an object and delegate requests to it.

Behavioural class patterns:

Template Method - an abstract definition of an algorithm. It factors out the skeleton of an algorithm and saves it in a superclass.

Interpreter - represents a grammar as a class hierarchy.

Behavioural object patterns:

Chain of Responsibility - delegates a request to one of the component objects in a chain of objects

Command - encapsulates a request as an object

Iterator - used to access elements of an aggregate

Mediator - encapsulates the interactions of a collaborating team

Memento - restores an object's internal state

Observer - updates the dependents of an object automatically upon changes to this object

State - allows an object to change behaviour when its state changes

Strategy - separates the strategy from the tactics in a family of algorithms to allow for variations in tactics

Visitor - separates operations on elements of a heterogeneous structure from the structure itself

6.1 Chain of Responsibility

6.1.1 Background

Decouples senders from receivers, by giving multiple objects a chance to handle the request. Chains the receiving objects and pass the request along the chain until one of them handles it.

The first object in the chain receives the request and either handles it or forwards it to the next candidate on the chain. The object that made the request has no knowledge of who will handle it. The request has an implicit receiver. To forward the request along the chain, and to ensure receivers remain implicit, each object on the chain shares a common interface for handling requests and for accessing its successor on the chain.

Structure

FIGURE 49.Chain of Responsibility pattern: UML diagram

Main Classes

Handler - an interface that handles requests received from the client, and forwards the request to the successor.

ConcreteHandler - handles requests it is responsible for, else forwards the requests to its successor.

The Chain of Responsibility can simplify object interconnections. Instead of having objects maintaining references to all the receivers, they keep a single reference to their successor. It also

has reduced coupling. Receiver and sender have no explicit knowledge of each other, also an object in the chain does not know about the structure of the chain.

However, the patterns offers no guarantee that the request will be handled since the request has no explicit receiver. The request can fall off the chain without being handled, or can go unhandled when the chain is not properly configured.

Sequence Diagram

FIGURE 50.Chain pattern: event sequence chart

6.1.2 Case Study

The case study example illustrates how a Chain of Responsibility handles requests for an on-line system, which is part of a user interface.

The user obtains information on any part just by clicking on it. The answer that is provided depends on the part of the interface and its Context. If no specific information exists for that part of the interface, then the system displays a general message about the immediate context. The information is organized according to its generality - from the most specific to the most general. The request is handled by one of several user interface objects, it is passed along a chain of objects until one handles it.

The problem is that the object that provides the answer is not known to the object that initiates the request. The solution provided with this example is the Chain of Responsibilities, decouples the originator of the request from the objects that provide information.

Structure

FIGURE 51.Chain of Responsibility example: UML diagram

Pattern Application - Roles played by case-study classes:

Handler defines an interface for handling requests. It maintains and keeps a reference to its successor on the chain of handlers.

ConcreteHandler2 - a ConcreteHandler abstract Class

ConcreteHandler22 - a subclass of the ConcreteHandler, its successor is also a ConcreteHandler2

ConcreteHandler21 - a subclass of the ConcreteHandler, its successor is also a ConcreteHandler2

The **ConcreteHandler1** is the successor of the ConcreteHandler2. The end of the chain is an instance of ConcreteHandler1.

6.1.3 Performance Evaluation

The Chain of Responsibility avoids coupling the sender of a request to its receiver by giving more than one object the chance to handle the request. The receiving objects are chained and the request is passed along the chain until an object can handle it.

The chain has a common interface for handling the requests and for forwarding the request to the successor on the chain. New types of request can be defined by subclassing, subclasses having the freedom to define new parameters with this pattern. The main impact on the performance is introduced by the forwarding scheme along the chain and by subclassing. The forwarding mechanism has two components affecting the overall performance: chain objects dynamic instantiation, which consumes most of the execution time because of the *new/delete* function calls; the forwarding scheme itself, consisting of calls to the next in chain operation function. As long as the chain is not very long, the performance may not be very much affected, and may be comparable with the performance of the typical functional C implementation, using case statements. Performance evaluation of the Chain pattern consists of defining the total execution time and the overhead:

$$\begin{array}{ll} \text{Total execution time} & T = F + C + N + D \\ \text{Overhead} & O_{Chain} = (N+C+D)ChainElements + F_{ForwardingScheme} \end{array}$$

For the forwarding scheme, where the request is handled by the first element of the Chain, Fig. 52. $O = 340$ clock cycles, the largest cost being paid to dynamic creation of chain elements, with *new* $N = 300$ clock cycles. In this example the chain element is the ConcreteHandler22.

FIGURE 52. Event sequence chart for request handled by the first element of the Chain.

The execution time distribution, of this application, is summarized in Table 16:

TABLE 16. Chain pattern execution time distribution

T	F	C	N	O
1329	27	110	1202	340
100%	2%	5.5%	92.5%	30%

The forwarding schema closed on the second element of the chain, has a larger overhead, the increase in overhead is caused by the call to the next handler function in the chain, in our example `ConcreteHandler21:: Handle`.

FIGURE 53. Event sequence chart for the request handled by the second element of the Chain
To improve performance, a simple solution would be to use static instantiation of the element of the chain, when the elements of the chain are known at compile time.

FIGURE 54. Chain pattern example: call graph

6.2 Command

6.2.1 Background

The Command pattern turns a request into an object and decouples the initiator of the request from the object satisfying the request, the receiver object. This gives a lot of flexibility in designing user interfaces, Commands being easy to assemble in composite commands, or added without changing the existing classes.

Command pattern encapsulates service requests into command objects. The Client decides which command to use and creates a new command object for each user request, by configuring the Invoker with a command object that can be executed for several user requests. The suppliers of the Clients are the Receivers; each command may have a Receiver.

Structure

FIGURE 55.Command pattern: UML diagram

Main Classes

Command declares an interface for executing an operation

ConcreteCommand provides the concrete implementation of **Execute** method by invoking the **Action** on **Request**.

Client creates a **ConcreteCommand** object and sets its receiver.

Invoker triggers the **Command** to execute the request

Receiver performs the operations associated with a request.

Hysteresis can be a problem in ensuing a reliable undo/redo Command mechanism. Errors can accumulate as commands are executed, unexecuted and reexecuted repeatedly, such that an application's state diverges from the original. Solution to this problem is to store more information in the Command to ensure that objects are restored at their original state.

Sequence Diagram

The following diagram shows the interactions between the Command's pattern objects:

FIGURE 56. Command pattern: event sequence chart

6.2.2 Case Study

The case study example consists in the implementation of two commands: an **ConcreteCommand2** and a **ConcreteCommand1**. The **ConcreteCommand2**'s **Execute** operation prompts the user for a document name, creates a corresponding **Document** object, adds the document to the receiving application, and opens the document.

The case study example deals with a **ConcreteCommand1** implementation. **ConcreteCommand1** is simple; it is not undoable and does not require arguments, allowing to be implemented as a

template class parametrized by the Receiver1 type. In this implementation ConcreteCommand1 maintains a binding between a Receiver objects and an Action stored as a pointer to a member function. To invoke a command that calls Action on an instance of class Reciever1, the Client has to create a “receiver”, and then a ConcreteCommand1.

```
Receiver1* receiver = new Receiver1;

Command* aCommand = new ConcreteCommand1 <Receiver1>
(receiver, & Receiver1::Action);
```

Then, the Client can invoke the operation Execute(). This implementation is specific to a command that does not require arguments, as is the case of the ConcreteCommand1.

FIGURE 57.Command example: UML diagram

Pattern Application - Roles played by case-study classes:

Command - an abstract interface class

ConcreteCommand1, ConcreteCommand2 - ConcreteCommands, subclasses of the Command class

Reciever2 - a Receiver of the ConcreteCommand2

Reciever1 - a Receiver of the ConcreteCommand1

Main - a Client and an Invoker

6.2.3 Performance evaluation

To increased flexibility the Command uses the concept of encapsulating service requests into command objects, such that changing the implementation of a command or introducing a new command does not affect other parts of the application.

The price of the increased flexibility is paid in loss of efficiency. As with all patterns that decouple components, the additional indirection, costs storage and time. An application that performs a service requests (a command) directly does not impose an efficiency penalty. The total execution time and the overhead were evaluated based on the call graph shown in Fig. 58.

$$\text{Total execution time} \quad T = F + V + C + N + D$$

where N presents the dynamic instantiation time of the Receivers, and ConcreteCommands objects. C represents the execution time of the ConcreteCommands, and Receivers constructors. and F represents the execution time of the Concrete Commands operations.

$$\text{Overhead} \quad O_{\text{Command}} = (V + C + N + D)_{\text{Command}}$$

The measured values are: $T = 1757$ clock cycles with an overhead 565 clock cycles. The execution time distribution for the ConcreteCommand1 example is given in the Table 17:

TABLE 17. Command pattern execution time distribution

T	F	V	N	C	O
1757	58	25	1630	44	565
100%	3.5%	1%	93%	2.5%	~35%

ConcreteCommand2 which is used when a request is made to open a document, creates the Document object at run time [Gamma95]. In this case the overhead will include the time to instantiate the Document object with *new*:

$$\text{Overhead} \quad O_{\text{Command}} = (V + C + N + D)_{\text{Command}} + N_{\text{Document}}$$

The pattern allows static instantiation of the Command objects, as a simple solution for overhead minimization. The overhead formula, in this case, becomes:

$$\text{Overhead} \quad O_{\text{Command}} = (V + C)_{\text{Command}}$$

FIGURE 58. Command pattern example: call graph

6.3 Interpreter

6.3.1 Background

The Interpreter shows how to define a grammar for a given language, how to represent sentences in the language, and how to interpret these sentences. For each grammar rule, the Interpreter pattern uses a class. The Interpreter is used with simple languages.

For complex grammars, tools such as parsers or generators are preferable.

Structure

FIGURE 59. Interpreter pattern: UML diagram

Main Classes

AbstractExpression declares an abstract operation, common to all nodes in the tree. The abstract operation is defined by the *Interpret* method.

TerminalExpression implements an instance of the Interpreter operation associated with each terminal symbols in the grammar.

NonterminalExpression a class to represent a grammar rule $R = R_1 R_2 \dots R_n$. R_1 through R_n are instance variables of their class, of type **AbstractExpression**. The *Interpret* methods

implement the concrete operation for nonterminal symbols in the grammar, R_1 through R_n . It calls itself recursively on the variables representing R_1 through R_n .

Context contains information global to the Interpreter.

Client builds an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes, that invoke the Interpret operation.

The grammar can easily be changed or extended by using inheritance: most of the classes are used to represent grammar rules. This pattern requires at least one class to be defined for every rule in the grammar. When the grammar has many rules it is hard to manage and maintain the code

Sequence Diagram

FIGURE 60. Interpreter pattern: event sequence chart

6.3.2 Case Study

The case study is based on the pattern structure from [Gamma95]. It describes a system which does the evaluation of an **Abstract expression**.

The grammar of the system defines the rules for different expressions. Two operations are defined on Abstract expressions.

The first operation is *Evaluate*; it evaluates an Abstract expression in a context that assigns a value to each variable.

The second operation, *Replace* generates a new Abstract expression by replacing a variable with an expression. *Replace* shows how the Interpreter pattern can be used for evaluating, and manipulating expressions.

The contents of the *AbstractExp*, *TerminalExp*, and *NonTerminalExp* classes are shown in Fig. 61. This example illustrates how many kinds of operations can “interpret” a sentence.

Of the three operations defined for *AbstractExp*, *Evaluate* shows what an interpreter should do most closely - to interpret a program or expression and to return a simple result.

Replace can be viewed as an interpreter whose context is the name of the variable being replaced along with the expression that replaces it, and whose result is a new expression. At the limit *Copy* can also be seen as an interpreter with an empty context.

This example demonstrates how the Interpreter pattern uses different operations to interpret a sentence. It also shows how the pattern works - it interprets an expression and returns a result.

Structure

FIGURE 61. Case study: UML diagram

Pattern Application - Roles played by case-study classes:

AbstractExpression class - an interface for all classes that define an Abstract expression. Its Interpreter method is implemented by the Evaluate method.

Context - a Context class defining the mapping of TerminalExp to values.

TerminalExp class - representing a variable, or a constant

NonTerminalExp class - representing an AND, OR, or NOT expression.

6.3.3 Performance Evaluation

The Interpreter pattern can be viewed as an operation distributed over a class hierarchy; in general, it is associated with a Composite pattern which has an Interpreter operation. The case study example has two Interpreter operations Evaluate and Replace. Evaluate operation is used to interpret an expression and returns a simple result. Replace operation has a context with the name of the variable to be replaced and the expression replacing it, the result being a new expression.

In this evaluation, the real time performance of the Interpreter is not associated with the performance of the Composite pattern. The overhead associated with the mechanism of the pattern refers only to the Interpret operations. The real time performance of the Composite pattern can be found in paragraph 4.3. There are differences in the overheads associated with the Interpreter operations which are often difficult to evaluate; the overheads may change with the implementation.

The total execution time and the overhead formulas are:

Total execution time $T = F + V + N + C + D$

Overhead $O_{Interpreter} = (F + V)_{Interpreter} + F_{Context} + (C + N + D)_{Context}$

For this example the measured total execution time $T = 2837$ clock cycles, and the overhead $O = 23$ clock cycles.

The overhead optimization can be easily achieved with static instantiation of the Context. The overhead formula, in this case, becomes:

Overhead $O_{Interpreter} = (F + V)_{Interpreter} + F_{Context} + C_{Context}$

The execution time distribution is summarized in Table 18:

TABLE 18. Interpreter execution time distribution

T	F	V	N	C	O
2837	16	7	2200	608	383
100%	<1%	<1%	77%	22%	13%

FIGURE 62.Interpreter pattern: call graph

6.4 Iterator

6.4.1 Background

The Iterator pattern provides a mechanism to access and traverse the elements of an aggregate (a list or a table) without knowing its internal structure. It is responsible of keeping track of the current element.

Structure

FIGURE 63.Iterator pattern: UML diagram

Main Classes

Iterator - an interface for accessing and traversing elements. It is tightly coupled to the **Aggregate**

ConcreteIterator - the concrete Iterator. It knows the current position in the traversal of the aggregate. New **ConcreteIterators** can be defined for new traversals.

Aggregate - an interface that has the mandate to create the Iterator object. To change the **Aggregate Class** without changing the client code, the design should use polymorphic iteration.

ConcreteAggregate - implements the `CreateIterator` method, which returns an instance of the **ConcreteIterator**.

Variants of the Iterator pattern

ExternalIterator - when the client controls the iteration. External Iterators can be difficult to implement over recursive aggregate structures like those in the Composite pattern, because a position in the structure may span several layers of nested aggregates. Better use Internal Iterators. External Iterators are more flexible than Internal Iterators.

InternalIterator - when the Iterator controls the iteration.

RobustIterator - when the insertions and removals do not interfere with traversal, and it does it without copying the aggregate.

PolymorphicIterator - when one can change the Aggregate Class without changing client's code. Should be used only when there is a need for polymorphism.

NullIterator - it's a degenerate Iterator helpful for handling boundary conditions.

Sequence Diagram

FIGURE 64.Iterator pattern: event sequence chart.

More than one traversal can be active at a time, as the Iterator keeps track of its own traversal state.

6.4.2 Case Study

The case study presents an External Iterator example, which is used to traverse a Queue in *forward order*. The Queue class supports iteration through its public interface. A transparent usage of different kinds of traversals is implemented with an abstract Iterator, a container for the Iterator interfaces (First(), Next(), IsDone(), CurrentItem()). The QueueIterator, which is a concrete

Iterator, is a subclass of the Abstract Iterator. It stores the Queue and an index (`_current`) in the queue. Assuming that we want to print 10 employees from a queue of employees, the external Iterator mechanism defines a method that takes Employee as an argument `PrintEmployee (Employee)`, and uses the methods defined by the Iterator, to print the queue.

Pattern Application - Roles played by case-study classes:

Queue - an Aggregate that defines the `CreateIterator` method

QueueOfEmployees - a `ConcreteAggregate` that overrides the `CreateIterator` method of the Queue

Iterator - an abstract Iterator that defines the following methods `First()`, `Next()`, `IsDone()`, and `CurrentItem()`

QueueIterator - a `ConcreteIterator` which overrides the `First`, the `Next`, the `IsDone`, and the `CurrentItem` methods of the abstract Iterator

6.4.3 Performance Evaluation

Iterator is one of the most used pattern. All class libraries use different Iterator variants. The Queue Iterator example is an external polymorphic Iterator, to support polymorphic iterations over different queue implementations. The polymorphism is accommodated through the abstract Queue class interface. For this implementation variant, it was required to have a powerful abstract Queue class interface. In our example the interface consists of the following methods: `First()`, `Next()`, `IsDone`, and `CurrentItem`. The run time performance of the pattern was measured with Quantify, and the results are shown in Fig. 65.

The total execution time and the overhead are:

$$\begin{array}{ll} \text{Total execution time} & T = F + C + N + D \\ \text{Overhead} & O_{\text{Iterator}} = F_{\text{Iterator}} + (C + N + D)_{\text{Iterator}} \end{array}$$

For the case study example the execution time distribution is presented in Table 19. This table is a summary of Iterator example execution shown in Fig. 65.

6.5 Memento

6.5.1 Background

Sometimes it is required to capture the internal state of an object, without violating encapsulation, such that the object can be restored to this state later. This can be implemented with Memento pattern.

Structure

FIGURE 66.Memento pattern: UML diagram

Main Classes

Memento - stores a snapshot of the internal states of the Originator, as much as his Originator wants, and protects against being accessed by other objects. Memento is passive. Only its creator, the Originator, will assign or retrieve its state.

Originator - creates a Memento containing a snapshot of its current internal state, using Memento to restore its internal state. It has access to all data required to restore itself to its previous state. It has a wide interface with Memento.

Caretaker- takes care of the Mementos. It does not access or operate on the contents of a Memento, it only passes the Memento to other objects.

The pattern has benefits: it preserves encapsulation by not exposing the information stored outside the Originator but managed by the Originator, it simplifies Originator by having clients manage the state they ask for, and stores the incremental changes to the Originator's internal state.

Sequence Diagram

FIGURE 67. Memento pattern: event sequence chart

When the Originator does not need to go back to a previous stage, the Caretaker will not pass the Memento back to the Originator.

6.5.2 Case Study

The case study example consists of a Target class that supports connectivity between objects. The Originator keeps track of the connections and describes these connections, maintaining connectivity relationships between objects. Specifically the example consists in executing an operation, and then unexecuting it.

In many applications supporting “undo” is not an easy task. A simple way to undo an operation is to store the original data and restore the object back to its equivalent state. The Originator in this example, has a public interface which may not be sufficient to allow precision on the reversal of its actions on other objects. The undo mechanism and the Originator must be used together in the undo process, without affecting the Originator internal methods. A solution to this problem is the Memento pattern.

Structure

FIGURE 68. Memento case study: UML diagram

Pattern Application - Roles played by case-study classes:

Memento - is opaque to other objects.

Originator is implemented as a Singleton. It will initialize the **Memento** with information that characterizes its current state. Only the **Originator** can store and retrieve information from **Memento**. `Instance()` returns an **Originator** object.

CareTaker - keeps the undo mechanism, for the translation of an object from one position to another. For example, as a result of a move operation, a **Memento** can be requested. The **Originator** creates and returns an instance of a **Memento**, which contains data structures that describe the current state of the **Originator**'s internal equations and variables. Later when the user does an undo of the operation, the **Memento** is returned back to the **Originator**. Based on the information stored in the **Memento**, the **Originator** changes its internal structures to return variables to their exact previous state.

The `Execute` and `Unexecute` methods of the **CareTaker** handle the *do-undo* operation. The operation needs to know the new coordinates. The new coordinates are defined by a public variable of the **CareTaker** class.

6.5.3 Performance Evaluation

Because the pattern has some performance limitations, using Memento pattern is not always appropriate. In some cases the Originator needs to copy large amounts of information and store them in the Memento. This situation may incur a large overhead, proportional with the amount of information copied in the Memento. In this cases is preferable to avoid using the Memento pattern.

The pattern should be used only when restoring the Originator is not time or memory expensive. The case study Memento example follows this rule. It uses a memento pattern to execute a move operation and then undo it; in this example these two operations does not require to copy and store large amounts of data in the Memento object.

The run time execution graph of the Memento example is presented in Fig. 69. The performance of the Memento pattern can be estimated with the following formulas:

$$\text{Total execution time} \quad T = F + C + N + D$$

$$\text{Overhead} \quad OMemento = (C + N + D)Memento + FMemento + FCreateMemento$$

Where $N_{Memento}$ represents the cost of creative Memento object dynamically, $FMemento$ refers to the cost of memento's functions and $CMemento$ is Memento's constructor execution time. The main contributors to the total execution time and the overhead of the Memento example are summarized in Table 20.

TABLE 20.Memento execution time distribution

T	F	C	N	D	O
1437	89	32	1316	0	146
100%	6%	2%	92%	0%	10%

The overhead is small because little information is copied in the Memento. To improve performance, the design can use a customized *operator new* instead of the default *operator new*. A detailed description of this solution is presented in [Meyer98].

FIGURE 69.Memento pattern: call graph

6.6 Observer

6.6.1 Background

The pattern defines a one subject and many observers objects dependency such that when the subject changes state, all the observers are notified and updated automatically. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state. The Subject is the publisher of notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications. The Observer pattern decouples concrete subjects from concrete observers: subjects can be reused without reusing their observers, and vice versa. All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class.

Structure

FIGURE 70. Observer pattern: UML diagram

Main Classes

Subject - an interface for registration/deregistration of observers. Has knowledge of its independent observers.

Observer - an interface for objects that are notified of changes in a subject, and consequently updated. The Update function can be made more efficient by extending the subject's registration interface to allow registering observers only for specific events of interest. When such an event occurs, the subject informs only those observers that have registered interest in that event. Observers have no knowledge of each other's presence.

ConcreteSubject - stores state of interest to ConcreteObserver objects, and sends a notification to its observers when its state changes.

ConcreteObserver - the implementation of the Observer's interface that is registered with the ConcreteSubject object. It stores states that are consistent with the subject's states.

Sequence Diagram

The interaction diagram illustrates the collaboration between a subject and an observer

FIGURE 71. Observer pattern: event sequence chart

6.6.2 Case Study

The case study example is the implementation of the pattern structure described in Fig. 72. It consists in building a Subject which supports multiple Observers. A Concrete Subject maintains the state. The Concrete Observers display the state. They both display the same state.

Structure

FIGURE 72. Memento case study: UML diagram

Pattern Application - Roles played by case-study classes:

ConcreteSubject has two methods `Notify` and `UpdateState`. `UpdateState` operation provides the accurate state; this operation updates **ConcreteSubject**'s internal state, and calls `Notify` to inform **ConcreteObservers** of the state change.

6.6.3 Performance Evaluation

The Observer pattern brings solutions for different types of problems as for example allowing to add Observers without modifying the Subject, or allowing to reuse Subjects without reusing the Observer, or allowing to reuse Observers without reusing Subjects. However, the pattern can introduce problem depending on the specific implementation. One of these problems is spurious updates which have impact on the performance. This problem has several causes: undefined dependencies criteria, simple update protocol which does not provide details on what was changed

in the Subject, forcing the Observer to work hard to understand the changes. Without additional protocol information to help observers discover what was changed, the Updates may overload the observers.

Aside from the performance issue mentioned above, an overload may be added by the function Notify. This may be the case when an Observer depends upon the state of many subjects. In this case, the design have to ensure that the observer is notified only after all the subjects have been modified to avoid notifying Observers more than ones. The Performance Evaluation is based on the example built from [Gamma95]. The total execution time and the overhead of the Observer pattern using dynamic objects creation are:

$$\begin{array}{ll} \text{Total execution time} & T = F + C + N + D \\ \text{Overhead} & O_{Observer} = (F + V)_{Mapping} + (N + C + D)_{Observer} \end{array}$$

$F_{Mapping}$ and $V_{Mapping}$ refer to the functions which implement the mapping of the Subjects to their Observers. These functions are: Attach, Detach, Notify, Update. The easy way of doing this mapping would be to store references to the Observers explicitly in the Subject. This may be space inefficient when there are many Subjects and few Observers. A solution is to trade “space for time” by using hash-tables to implement the Subject to Observer mapping. Hash-tables approach, however, increases the cost of accessing the Observers.

Another part of the overhead refers to the way Notify is called to trigger the updates of the Observers: having Subject’s state-setting functions, call Notify once the Subject’s state has changed, has the disadvantage of causing several consecutive updates, which is inefficient. A solution to this problem would be to make Clients responsible for calling Notify at the right time. This has, however, the disadvantage of asking Clients to accept more responsibilities.

The last part of the overhead refers to the ConcreteSubject instantiation. The results of the real-time measurements are $T = 1186$ clock cycles and $O = 420$ clock cycles. The time distribution is shown in the Table 21 below:

TABLE 21.Observer example execution time distribution

T	F	V	N	C	O
1186	18	9	1088	71	420
100%	2.5%	2.5%	89%	6%	35%

The total execution time and the overhead can be decreased by using static object creation for the Subjects and Observers. The total execution time and overhead formulas in this case are

$$\text{Total execution time} \quad T = F + C$$

$$\text{Overhead} \quad O = (F + V)\text{MappingFunction} + C_{\text{Observer}}$$

FIGURE 73.Observer pattern example: call graph

6.7 Mediator

6.7.1 Background

To promote reusability Object-Oriented design enforces the distribution of behaviour among interconnected objects. However, this architecture makes difficult to change system's behaviour, because to change functionality of an object requires to investigate, and eventually change, the interconnected objects. This problem can be avoided by encapsulating common collective behaviour in an object. This object is called **Mediator**. A **Mediator** object has several responsibilities: it serves as intermediary for all objects in a set and coordinates the interactions of the objects in the set. The **Mediator** pattern has many benefits: the behaviour of the system can be changed by subclassing **Mediator** only; **Colleague** classes can be reused as is; decouples colleagues, simplifies object protocols replacing many-to-many interactions with one-to-many interactions and finally centralizes control, trading complexity of interaction for complexity in the mediator. Encapsulating interactions in an object, separates objects interactions from their individual behaviour. That helps to maintain and understand the system.

Structure

FIGURE 74. Mediator pattern: UML diagram

Main Classes

Mediator an interface for communication with other objects in the set. There is no need to define abstract **Mediator** class when colleagues work with only one **Mediator**.

ConcreteMediator implements **Mediator** behaviour that consists of coordinating the interactions among **Colleague** objects, knows and maintains its colleagues.

Colleague classes each Colleague Class knows its Mediator object communicating with its mediator when requested, instead of communicating with another colleague.

Sequence Diagram

Colleagues communicate throughout a Mediator object. The mediator implements the cooperative behaviour by routing requests between the appropriate colleague(s).

FIGURE 75.Mediator pattern: event sequence chart

6.7.2 Case Study

The ConcreteMediator mediating between the colleagues, knows the colleagues in a dialog and coordinates their interaction. It acts as a hub of communication for colleagues. In this example the ConcreteMediator mediates between two Colleagues, ConcreteColleague1 and ConcreteColleague2. The two colleagues communicate to each other indirectly, through the ConcreteMediator, none of them having knowledge of each other implementation.

Most of the implementation is encapsulated in the ConcreteMediator, its responsibility being to control and coordinate the behaviour of the two Colleagues. Furthermore, because the behaviour is localized in one class, it can be changed or replaced by extending, or replacing that class. This implementation creates the Mediator in function *main* and never destroys it. When the creation of the Mediator is done at run time, destruction of the Mediator has to be also done at run time. In this case the Overhead has to include the time consumed in *delete* function.

Structure

FIGURE 76. Mediator case study: UML diagram

6.7.3 Performance evaluation

The main consequences of the Mediator pattern can be summarized by the following: It limits subclassing, decouples colleagues, simplifies object protocols, abstracts how objects collaborate, and centralizes control. The Mediator pattern is particularly good at improving resource management in the presence of concurrency. This pattern adds one layer of indirection, the Mediator itself, which translates in loss of efficiency. When introducing one layer of indirection, the overhead is caused by the class handling the indirection, the Mediator. The total execution time and the overhead, in this case are defined by the following formulas:

Total execution time $T = F + V + C + N$

Overhead $O_{Mediator} = (C + N + D)_{Mediator} + (F + V)_{Mediator} + F_{CreateColleagues}$

The measured values are $T = 2654$ clock cycles and $O = 687$ clock cycles, the overhead representing 25% of the total execution time T . Most of the overtime is caused by the Mediator object dynamic instantiation with *new*. The real-time distribution is shown in the table below:

TABLE 22. Mediator example execution time distribution

T	F	V	C	N	O
2654	27	27	174	2428	687
100%	1%	1%	6.5%	91.5%	26%

Mediator example uses dynamic instantiation of the Mediator objects at compile time. By using static instantiation of the objects, the total execution time and the overhead can be reduced. The total execution time and the overhead formulas become:

Total execution time **$T = F + V + C$**

$$\text{Overhead} \quad O_{\text{Mediator}} = C_{\text{Mediator}} + F_{\text{Mediator}} + F_{\text{CreateColleagues}}$$

FIGURE 77. Mediator example: call graph

6.8 State

6.8.1 Background

The State pattern describes how an object can exhibit different behaviour when its state changes. The main idea with this pattern is to introduce an abstract class called State to represent the states of the object. State pattern encapsulates state-specific behaviour and partitions behaviour for different state. All behaviour associated with a particular state is implemented into a ConcreteState subclass. State objects can be created only when they are needed. Destroying them explicitly is preferable as state changes are known at run-time.

Structure

FIGURE 78.State pattern: UML diagram

Main Classes

Context - an interface for clients, which maintains an instance of a ConcreteState subclass that defines the current state. The Context delegates state-specific requests to the current ConcreteState object.

State - an interface defining the behaviour associated with each state of the Context.

ConcreteState subclass - implements the behaviour associated with a state of the Context.

Sequence Diagram

FIGURE 79.State pattern: event sequence chart

6.8.2 Case Study

The case study is a simplified version of a communication protocol. The Context Class represents a network connection. A Context object can be in one of the following states: Established, Listening, Closed. When a Context object receives requests from other objects, it responds differently depending on its current state. The State pattern allows Context class to exhibit different behaviour in each state. The key idea in this pattern is to introduce an abstract Class called State to represent the states of the network connection. The State class declares an interface common to all classes that represent different operational states. Subclasses of the State class implement state-specific behaviour.

State transitions are implemented by the ChangeState function of the Context class. State subclasses maintain no local state. Only one instance of each is needed at any time. Every new instance of a State subclass is created with the Instance () operation when the State is changed. An operation implemented by the subclasses, after performing state-specific work, will call the ChangeState operation. to change the state of the Context. State subclasses are the only ones defining each state transition. This implementation creates states ahead of time and never destroys them.

Structure

FIGURE 80.State pattern example: UML diagram

Pattern Application - Roles played by case-study classes:

Context provides an interface for transmitting data and handles requests to change states.

State duplicates the state-changing interface of Context. Each State operation takes Context as a parameter, letting State access data from Context and change the Context's state.

Established, Listen, Closed - ConcreteStates which maintain no local state, to allow to be shared. Only one instance of each is required. The unique instance of each State subclass is obtained by the static Instance operation. Each State subclass implements state-specific behaviour for valid requests in the state.

There are several implementation alternatives for the State pattern. State transitions, creation, destruction of objects, or inheritance can be implemented in different ways.

State transitions can be implemented in the Context or in the State subclasses, *creation/destruction* of State objects can be done as needed or creation of all States can be done initially, *changing the behaviour for a request* can be accomplished by changing objects class at run time through dynamic inheritance. In the case study State example, the state transitions are implemented in the context, and the state objects construction is done initially.

6.8.3 Performance Evaluation

An alternative to State pattern is to have one monolithic implementation with the internal states defined by data values, and checked explicitly by operations. Such implementation requires case statements, making the design less flexible and potentially less performant. Introducing State subclassing may increase the number of subclasses when many states are required, but it provides better performance than large switch or if/then/else statements.

Creating State objects when needed and destroys them thereafter, brings another improvement of the performance; this refers mainly to memory consumption at run time. There are other solutions to implement state driven code as mentioned in [Carg92]. One of the solutions presented by [Carg92] is to map inputs to state transitions and store this mapping in a look-up table. A look-up table however is less efficient than a virtual function call.

For the State pattern implementation example discussed in the case study, the total execution time and the overhead can be expressed with the formulas:

$$\text{Total execution time} \quad T = F + V + C + N + D$$

$$\text{Overhead} \quad O = (C + N + D) \text{ConcreteStates} + F \text{Instance} + V \text{ConcreteStates}$$

The Table 23 presents the execution time distribution of the State example, the total execution time to create 3 states is 1459 clock cycles with an overhead of 684 clock cycles. In both cases N has a large contribution, representing 90%(1316 clock cycles) of total execution time and 81% (480 clock cycles) of the overhead.

State pattern does not introduce a very high overhead. It may be even smaller than the overhead used in a typical C implementation, with case statement or table look-up.

TABLE 23.State example execution time distribution

T	F	V	C	N	O
1464	40	19	89	1316	684
100%	2%	2%	6%	90%	47%

State example uses dynamic object creation with `new`, which is expensive. The mechanism of the pattern allows to create some of the objects ahead of time (Context and Listen) and never destroy them. These objects, which can be instantiated at compile time, can use static instantiation, the other being created as needed at run time, need to use dynamic instantiation. This brings on one side improvement of the execution time, on the other side improvement in memory consumption. But because most of the ConcreteStates are instantiated at run time, the Overhead is affected by their object instantiation with `new`. A solution to this problem is presented in [Meyer98].

FIGURE 81.State pattern example: call graph

6.9 Strategy

6.9.1 Background

The Strategy pattern shows how an algorithm can be implemented such that this implementation allows to vary the algorithm independently of its context. Variants of the algorithms or context behaviour can be structured in a hierarchy of Strategy classes. The hierarchies of Strategy classes define a family of algorithms or behaviours for contexts to reuse. Inheritance can help to separate common functionality of the algorithms, by encapsulating the algorithm in separate Strategy classes. This implementation has benefits: allows to vary the algorithm independently of its context, avoids conditional statements for selecting desired behaviour, permits selection of implementations of same kind of behaviour, applies to families of related algorithms.

The Strategy has several drawbacks however: Clients need to know about the strategies, it introduces a communication overhead between the Strategy and the Context, and it definitely requires more objects than a standard OO implementation.

Structure

FIGURE 82.Strategy pattern: UML diagram

Main Classes

Strategy - a common interface to a set of algorithms. The context uses this interface to call the algorithm defined by a ConcreteStrategy.

ConcreteStrategy - a concrete implementation of a variant of the algorithm using the Strategy interface.

Context - configured with a ConcreteStrategy object, maintaining a reference to a Strategy object. To allow Strategy to access its data, sometime the Context provides a specific interface. Clients interact with Context only.

Sequence Diagram

FIGURE 83.Strategy pattern: event sequence chart

6.9.2 Case Study

The case study describes a method of selecting an algorithm from a family of algorithms using the Strategy pattern. Hard-wiring all such algorithms into the classes that require them is not a good solution. To avoid hard-wired algorithms, a design can use the Strategy pattern mechanism. The Strategy pattern helps to encapsulate different algorithms in ConcreteStrategy classes. The case

study uses the Strategy to encapsulate variants of the algorithms in the ConcreteStrategy subclasses: ConcreteStrategy1, ConcreteStrategy2, ConcreteStrategy3.

Structure

FIGURE 84. Case study: UML diagram

Pattern Application - Roles played by case-study classes:

Context is responsible for maintaining and updating the Client's interface.

Strategy - an abstract class. Its subclasses implement different algorithms. The main Strategy interface is designed to support as many algorithms as subclassed Strategies exist in the design. To avoid changes with every new subclass, the Strategy interface needs to be very generic.

ConcreteStrategy subclasses of the Strategy which implement different algorithms.

6.9.3 Performance Evaluation

When a set of different behaviours are encapsulated into one class, conditional statement can be used to select the desired behaviour.

The Strategy provides an alternative to conditional statement inside a class, by encapsulating different behaviours in different classes. The drawback, in this case, is the increase in the number of objects in an application.

The Strategy pattern defines the Strategy class as an abstract interface to the ConcreteStrategy classes. The Context can use this interface to pass all needed information to the selected ConcreteStrategy subclass.

The real-time measurements are presented in Fig. 85, the total execution time and the overhead being calculated with the following formulas:

$$\begin{aligned} \text{Total execution time} & \quad T = F + V + C + N \\ \text{Overhead} & \quad O_{\text{Strategy}} = (C + N + D)_{\text{Strategy}} + V_{\text{Strategy}} \end{aligned}$$

The object instantiation mechanism consists of a nested object instantiation sequence. To instantiate a Composition it is required to pass a ConcreteStrategy to it.

*Context * context1 = new Context (new ConcreteStrategy1);*

The overhead in this example of the Strategy patterns is high. This is because it includes the instantiation of 3 ConcreteStrategy objects: ConcreteStrategy1, ConcreteStrategy2, and ConcreteStrategy3.

The execution time distribution and the overhead are presented in Table 24 below:

TABLE 24.Strategy example execution time distribution

T	F	V	C	N	O
1528	14	6	78	1430	1054
100%	0.7%	0.3%	5%	94%	69%

Same as with other patterns (Memento, Mediator, State), the Strategy can perform better if instead of dynamic object instantiation, the design uses static object instantiation. In this case, the total

execution time and the overhead will not include the components related to dynamic object instantiation, and become:

$$\text{Total execution time} \quad T = F + V + C$$

$$\text{Overhead} \quad O = C_{Strategy} + V_{Strategy}$$

Or use a customized operator *new* instead of the default operator *new*. A detailed description of this solution is presented in [Meyer98].

FIGURE 85.Strategy pattern example: call graph

6.10 Template Method

6.10.1 Background

A Template Method defines an algorithm in terms of abstract operations that subclasses override to provide concrete behaviour. It is a fundamental technique for code reuse, which can be found in any abstract class.

Structure

FIGURE 86. Template pattern: UML diagram

Main Classes

AbstractClass - defines abstract primitive operations overridden by the concrete operations in the concrete subclasses. The Abstract class includes also the Template Method defining the algorithm. The Template Method calls primitive operations as well as operations defined in AbstractClass or those of other objects. In C++, the primitive operations that a Template Method calls can be declared *protected* members, to ensure that they are only called by the Template Method. Primitive operations that *must* be overridden are declared pure virtual. The template method itself should not be overridden; therefore you can make the template method a nonvirtual member function.

ConcreteClass - defines the concrete primitive operations to implement subclass-specific parts of the algorithm.

Template methods invoke one of several types of operations:

- concrete operations (either on the ConcreteClass or on client class);
- concrete AbstractClass operations (operations that are generally useful to subclasses);
- primitive operations (abstract operations);
- factory methods,
- hook operations, which provide default behaviour that subclasses can extend if necessary. The Template Methods have to specify the type of each operation. The hook operations which may be overridden and abstract operations which must be overridden. To reuse an abstract class effectively, subclass writers must understand which operations are designed for overriding.

6.10.2 Case Study

The case study provides an example based on the description from Fig. 86

6.10.3 Performance Evaluation

The Template Method is used in almost every abstract class, to define the template of an algorithm. Generally speaking, the algorithm can have an *invariant part*, and a *variant part*.

The *invariant part* of the algorithm is implemented by the Template Method of the Abstract class, while the *variant part* is implemented by the methods of the subclasses.

To allow subclasses extend the algorithms, the pattern uses virtual methods. In the case study example the Template Method calls two primitive operations: PrimitiveOperation1 and PrimitiveOperation2. The primitive operations are all virtual functions to allow subclasses to extend the algorithm with their specific operations.

The only overhead introduced by this pattern is the overhead introduced by the virtual primitive functions.

Total Execution Time $T = F + V + C + N$

Overhead $O_{TemplateMethod} = (F + V)TemplateMethod$

where N represents the instantiation time of the `ConcreteClass` object, C represents the `AbstractClass` and `ConcreteClass` constructors time, F represents the Template Method and primitive methods execution time, and V represents the overhead introduced by the virtual attribute of the primitive functions.

The overhead is very small. Based on the measurements shown in Fig. 88, the total execution time $T = 981$ clock cycles, and the overhead $O = 24$ clock cycles.

FIGURE 87. Template Method example: call graph

6.11 Visitor

6.11.1 Background

Allows the definition of a new operation without changing the classes of the elements on which it operates. There are a number of variations of Visitor patterns: Standard Visitor, Default Visitor, Extrinsic Visitor, Acyclic Visitor. This presentation refers only to the Standard Visitor. Details on other types of Visitor pattern can be found in [Gamma95].

6.11.2 Standard Visitor

Standard Visitor defines two class hierarchies: the node hierarchy for the Elements being operated on, and the node Visitor hierarchy for the visitors defining operations on the Elements. A new operation can be defined by adding a new subclass to the Visitor class hierarchy. New functionality can be added simply by defining new node Visitor subclasses; no need to add new subclasses in the Element's hierarchy.

The pattern has important benefits: makes easy to add new operations that depend on the components of complex objects, gathers related operations and separates unrelated ones; related behaviour is not spread over the classes; it is localized in a visitor, and allows to visit objects that do not have a common parent class. Any type of object can be added to a visitor interface, contrary to the Iterator.

The main drawback in applying the standard Visitor pattern is the need to modify every concrete visitor whenever a new concrete element is added. Each new ConcreteElement gives rise to a new abstract operation on Visitor and a corresponding implementation in every ConcreteVisitor class. By assuming that the ConcreteElement interface is powerful to allow visitors do their job, the pattern often imposes public operations to access an Element's internal state. This compromises the Element encapsulation.

6.11.3 Standard Visitor

Structure

FIGURE 88. Visitor pattern: UML diagram

Main Classes

Visitor - The Visitor Class defines a `ConcreteElement()` method for each concrete class in the inheritance hierarchy of elements. The signature and name of each `VisitConcreteElement()` method identify the concrete class of the element being visited. `VisitConcreteElement()` method is the Visitor access interface to a particular concrete element.

ConcreteVisitor - implements a meaningful version of those `VisitConcreteElement()` methods that are relevant to the application. It may implement a combination of `VisitConcreteElement()` methods for concrete and abstract element, taking advantage of the polymorphism in the element Class hierarchy. It provides the context for the algorithm and stores its local state, which may accumulate results during the traversal of the structure.

AbstractElement - defines an `Accept()` method that has a visitor as its argument. The `Accept()` method is the starting point for a client to cause a traversal of a structure of elements. Note that multiple abstract elements may exist in an inheritance hierarchy of elements.

ConcreteElement - implements a specific `Accept()` operation that calls the appropriate `VisitConcreteElement()` method of its visitor. For reduced complexity in this pattern each concrete element is preferably a leaf node of the inheritance hierarchy.

ObjectStructure - represents a collection of elements, such as a list or a set that uses a mechanism to traverse the collection. The mechanism is provided by either the elements themselves or by the base Visitor class.

An alternative implementation of the ObjectStructure is the Composite pattern.

Sequence Diagram

A Client of the Visitor creates a `ConcreteVisitor` object and then traverse the `ObjectStructure`, visiting each `Element` with the Visitor. When an `Element` is visited, it calls the Visitor operation that corresponds to its class. The `Element` supplies itself as an argument to this operation to let the Visitor access its state, if necessary

FIGURE 89. Standard Visitor: event sequence chart

6.11.4 Case Study

The base for the Visitor is the Composite presented in chapter 5.3. In the Visitor example, the Element classes of the Composite pattern are enhanced with the Accept operation, to allow them to interact with the Visitor.

Element's subclasses redefine Accept() operation for each type of element ConcreteElement1, or ConcreteElement2. The ElementVisitor has a virtual function for each part of the Element, VisitConcreteElement1(), VisitConcreteElement2(), which does nothing by default; these virtual functions are redefined by the ElementVisitor's subclasses, ConcreteElement1 and ConcreteElement2.

Structure

FIGURE 90. Standard Visitor case study: UML diagram

Pattern Application - Roles played by case-study classes:

ElementVisitor - a Visitor. Defines the **Accept()** operation which is a “double dispatch”, i.e. it depends on two types: the Visitor and the Element. Accept operation helps that instead of

binding operation statically into the Element interface, allows to consolidate the operations into the Visitor and then uses it to do the binding at run time.

ConcreteVisitor1 defines a particular algorithm over the element structure.

ConcreteVisitor2 defines another particular algorithm over the element structure.

Element - is an Abstract Element implemented as a Composite pattern. Its methods return the attributes of each part of the Element. The collection of element parts is gathered in a List, which can be traversed with a ListIterator (see Composite pattern).

ConcreteElement1 - a subclass of the Element class. Its methods redefine the methods of the Element.

ConcreteElement2 - a subclass of the Element class. Its methods redefine the methods of the Element as well.

The ObjectStructure in this example is the ListIterator of a collection of Element parts. Each ConcreteElement implements the Accept operation, that calls the matching Visit operation for the ConcreteElement. This example shows how to use a ConcreteVisitor with an Element.

6.11.5 Performance Evaluation

The table below shows the performance and characteristics of the Standard Visitor versus other Visitor pattern variants

TABLE 25. Visitor pattern variants

Pattern	Coupling between Visitor & Element	Run time efficiency	Ease of maintenance
Standard Visitor	very high	high	low
Default Visitor	high	high	medium
Extrinsic Visitor	low	low	high

6.12 Conclusions on Behavioral Patterns

Behavior patterns provide solutions for different behaviors. They allow to:

- Localize changes to an object

Strategy: encapsulates an algorithm

State: provides state-dependent behaviour

Mediator: provides interaction between collaborators

- Use objects as arguments to provide the following behaviors:

Command: encapsulates a request; client is insulated from knowing who carries out the request, and what is actually requested of the receiver

Memento: encapsulates internal state; client knows nothing about coupling of the Memento's internal state

- Use localizing versus distributing communication

Mediator: localizes interactions/dependencies among collaborators; control flow is easy to follow.

Observer: distribution of communication dependencies is determined through the Observer registration interface; control flow is harder to follow, but because subjects are decoupled from Observers, both are made more reusable.

- Understand how to decouple collaborators and avoid direct references.

Command: binding is localized in an object; sender is decouples from receivers, making senders reusable.

Observer: decouples subjects from Observers through a notification interface. receivers can be added to this interface dynamically.

Mediator: decouples collaborators from each other, but couples them to the Mediator.

A concise view of the Behavioral pattern is presented in Table 26.

TABLE 26. Performance of Behavioral Patterns

Pattern Name	Description	Performance	
		Memory	Execution Time
Chain	Has a common interface to handle requests. Decouples sender from receiver.	Medium increase mainly when the Chain uses many elements	Medium overhead $O = (N+C+D)ChainElements + FForwScheme$
Command	Decouples the initiator of the request from the object satisfying the request.	Small increase caused by Command's object	Medium overhead $O = (F + V + C)Command$
Memento	Allows the access to the internal state of an object without violating encapsulation.	Increased caused by Memento's object	Small overhead $O=(C+N+F)Memento + FCreateMemento$
Interpreter	Used to define the grammar of a language.	Small increase caused by Context's object and Interpreter's virtual functions.	Small overhead $O = (F+V)Interpreter + (C + F)Context$
External Iterator	Provides the mechanism to access and traverse the elements of an aggregate.	Small increase caused by Iterator's object	Small overhead $O = (F + C)Iterator$
Template Method	Allows to factor out common behavior for an Algorithm.	No impact	Small overhead $O = (F+C)TemplateMethod$
Observer	Defines the interaction between a Subject and many observers.	Small caused by Observer's object	Medium to small overhead $O = (F+V)Mapping + CObserver$
Mediator	An intermediary and coordinator of the interactions of all the objects of a Set.	Small caused by Mediator's object	Medium to small overhead $O = (F+C)Mediator + FCreateColleagues$
State	Encapsulates States behavior in ConcreteState subclasses.	Saves memory when states are created as needed..	Medium overhead $O = FInstance + (C+N+V)ConcreteStates$
Strategy	Allows to vary an algorithm independently of its Context.	Medium increase the pattern may create many objects	Medium overhead $O = (C+V)Strategy$
Visitor	Allows to add operations to classes without changing them (double-dispatching)	Medium increase as the pattern creates more objects.	Small overhead $O = (F+C+V)Visitor + (F+V)Accept$

Chapter 7

7.0 Idioms

As mentioned in paragraph 2.6, Idioms represent the lowest level patterns, being specific to a programming language. idioms are also called Coding patterns, and they can be looked at as old optimization techniques. Such techniques are true tricks for writing efficient C++ code. Most of the C++ users already know these coding patterns or idioms, though it is difficult to find the places anyone has written them down. This list of Idioms is taken from [Copl96] and refers only to the C++ programming language.

1. *Lifetime follows scope*

This Idiom lets the language invoke the Constructors and Destructors automatically, for those cases where the object is declared as member of the scope and the object's lifetime follows the scope. The program using this patterns is likely to have the following performnace attributes: reduced memory fragmentation; runs faster because of reduced memory management overhead; less garbage, and more objects properly cleaned.

2. *Call-by-Reference*

Call-by-value is expensive because of the copy constructor, which is used for the initialization of the argument with the passed value. This does not apply to *Call-by-Reference*, which is more efficient, because no constructors or destructors are called since no new objects are being created.

3. *Customized Allocation*

A pattern that provides reduced overhead for dynamic memory allocation. Each class manages dynamic data for its instances. Memory is statically engineered, or allocated at run-time in large blocks, managed with simple free list and allocation/deallocation strategies than a general purpose list. The *Customized Allocation* pattern provides a much improved real time performance, because of improved memory management (no memory fragmentation).

4. *Initialize, not assign*

Initialize members directly instead of assigning a default value inside Constructor's body. The immediate performance gain is an improved run-time performance, as the pattern eliminates constructor/destructor calls for an object.

Chapter 8

8.0 Conclusions and Future Research

8.1 Conclusions

The conclusions of this thesis are summarized in three subsections. The first subsection presents a summary of performance issues related to Design Patterns. The second subsection (**Contributions**) describes the fields of object-oriented programming to which this thesis contributes. The third section (**Reflections**) shows what I have learned during this research and the ways in which I expect to apply the resulting experience in future software projects.

8.1.1 Performance Summary

Performance Issues	Pattern name	Solution
Overhead caused by Virtual Functions	Factory Method, Abstract Factory, Builder, Prototype, Singleton, Adapter, Bridge, Composite, Flyweight, Command, Interpreter, Observer, State, Strategy, Visitor	<ul style="list-style-type: none"> • Avoid virtual functions when possible
Overhead caused by object instantiation with new / delete	Singleton, Bridge, Decorator, Flyweight, Proxy, Chain, Memento	<ul style="list-style-type: none"> • Use customized operator new[Meyer98] • When possible replace dynamic instantiation of the object with static instantiation[Cop196] • Create objects only when needed
Overhead caused by another layer of indirection	All except: Factory Method, Template Method	<ul style="list-style-type: none"> • Proper utilization of constructors and destructors • Initialize member data through an Initialization list
Overhead caused by too many objects	Decorator, Chain, Interpreter	<ul style="list-style-type: none"> • Used only when the number of objects handled by the system is not very large.
Overhead caused by Cloning (deep-copy)	Prototype	<ul style="list-style-type: none"> • Use deep copy for complex objects only • Use custom operators new/delete

Performance Issues	Pattern name	Solution
Overhead caused by Multiple Inheritance	Adaptor	<ul style="list-style-type: none"> • Solve ambiguity • Avoid diamond-shape inheritance hierarchy • Avoid using virtual functions
Overhead caused by the need to traverse the associated aggregate very often	Iterator, Composite	<ul style="list-style-type: none"> • Cache traversal

The overhead caused by object instantiation with *new* is very large. Careful design and implementation is advised for the patterns using this type of instantiation. The solutions listed in the performance summary table are largely described in [Meyer98]. Also it is important to mention that most of these solutions are included in the list of Idioms. For example: the solution for the overhead introduced by dynamic object instantiation with *new*, is the *Customized Allocation* Idiom, solutions to indirection are the *Lifetime Follows Scope*, and *Initialize Not Assign* Idioms.

8.1.2 Contributions

The following subsection describes the contribution to object-oriented programming in terms of design patterns, performance, and efficiency in using design patterns in C++ programming language.

Contribution to Design Patterns.

This thesis contains a performance analysis of the set of twenty three patterns presented in [Gamma95]. The analysis is based on sample code written in C++.

The main goal of the thesis is to complement the information on design patterns with the analysis of their performance and efficiency. In other words, with the results presented in this thesis I would like to answer the question if the patterns presented by Gamma et al. are performant and if not what are the causes. In the end I also wanted to find the answer to another question: what can be done to improve their performance. The result of the analysis are presented in the Conclusions of chapters

3, 4, and 5, showing the causes of poor performance for each individual pattern from Gamma's catalogue.

The contribution in this case consists in bringing more information and helping to better understand the mechanism of the patterns.

Contribution to Performance in using Design Patterns.

Programmers cannot ignore performance and efficiency, these are real concerns. It is easy to have a clean well architected design if you ignore performance, building the design on abstractions and decoupled modules. Abstraction is good, but it is often the enemy of performance. Any valid design must balance this two issues: abstraction and performance. The thesis has the ambition to answer the following question: How can I have a reusable, well architected software with no compromises in the efficiency? The answer to this question is based on the analysis presented in the chapters 2, 3, 4, and 5. These are the main advices:

- Replace dynamic instantiation of the objects with static instantiation when this is possible, or use optimized new/delete methods.
- Use combinations of design patterns which together can improve performance
- Use parameterized types (templates) when possible to improve real time performance
- Use design patterns and coding patterns together to get the benefits of both - abstraction from design patterns, and improved performance from coding patterns.

8.1.3 Reflections

This section summarizes some of my thought about this research on “design Pattern Performance”.

I began this work without knowing what a “Design Pattern” was. Even though I used Object Oriented programming languages, I missed the benefits of using design patterns in conjunction with object-oriented programming techniques. Conducting a research on design patterns opened new ideas and ways for future software projects. I learned to feel how the design of an application program can be improved in describing reusable parts of the application as design patterns.

Design Patterns and Performance are topics that will be increasingly important in future research and software development. This thesis represents just a start in this direction. Possible future research could explore the following:

Java/Smalltalk Alternatives to C++.

Java programming language is in its syntax closely related to C++, while it is an interpreter language as Smalltalk. These similarities of Java to already existing languages simplify a switch to Design pattern programming in Java for suitable applications. Many of the Design Patterns have their correspondents in Java or Smalltalk, but no performance analysis was done. It would be interesting to compare the performance of the existing C++ versions with the Java or Smalltalk versions of the patterns.

Performance Analysis of the Variants.

It was difficult to complete this analysis for the set of patterns presented in [Gamma95] and for all the variants of each individual pattern. For example, the Iterator pattern has nine variants, the Visitor four. There are many other variants, not formally published, making the task even more difficult. To complete the work, the analysis should be extended to study each variant individually, and then compare the performance of the variants against each other.

Cluster of Patterns, also named Pattern Languages.

Until now, activity related to design patterns was oriented primarily for studying individual patterns or defining new patterns. Little work was done on studying the performance of Clusters. Some work was done by Doug Smith in building Clusters of design patterns, to be used in network communications [Schm95b].

Multiple patterns can lead to large clusters of dependent patterns, which may become difficult to manage and maintain. Jiri Soukup in [Souku94] proposed a method that solves the problem, by representing each pattern with a special class named Pattern class. This class encapsulates all the behavior and logic of the pattern. Performance information can be added to each such class for a more complete definition of the pattern.

Another aspect of the study can refer to analyzing performance optimization for designs using clusters of different patterns.

Libraries of Generic Patterns.

Until now it was assumed that the programmer will implement patterns in code by designing the classes as described by the pattern. Jiri Soukup started to build a pattern library named PTL (Pattern Template Library). To date, as far as know, the library contains only five patterns [Souku97]. The analysis of the performance of the patterns included in the library would be very useful for designers using the library.

Appendix: UML Conventions

Note: The names of the Abstract type classes and of the Virtual Functions are indicated with *italic fonts*.

Bibliography

[Alex79] C. Alexander (1979) - *The Timeless Way of Building*, Oxford University Press, 1979

[Auer96] K.Auer, K.Beck - *Lazy Optimization: Patterns for efficient Smalltalk Programming* - in Pattern Languages of Program design, Vol.2, (J.Vlissides, JCoplien and N.Kerth eds.) pp. 19-42, Addison-Wesley, 1996

[Batt97] Jean Batten (1997) - *A Reusable Apply Function via the Command pattern*, C/C++ Users journal, <http://www.cuj.com>, June 1997

[Berg95] W.Berg, M.Cline, M.Girou - *Lessons Learned from the OS/400 OO Project* - Commun. ACM 38, 10, pp 54-64, Oct. 1995

[Booch94] Booch G. (1994) - *Object-Oriented Analysis and design with Applications*, Benjamin/Cummings Publishing Company Inc., Redwood City, CA

[Busch96] F.Buschmann, R.Meunier, H.Rohnert, P.Sommerland, M.Sthal (1996) - *Pattern Oriented Software Architecture - A System of Patterns*

[Carg92] Tom Cargill (1992) - *C++ Programming Style*, Addison-Wesley Readings, MA , 1992

[Carg96] Tom Cargill (1996) - *Managing Dynamic Object in C++*, Dr. Dobb's Journal, June 1996

[Copl96] James Coplien - *After all, we cannot ignore efficiency*, C++ Report (6), June 1996

[Ellis90] Ellis M.A. and Stroustrup B. (1990) - *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company, Reading MA

[Frank95] Franks G., Hubard A., Majumdar S., Neilson J., Petriu D.C., Rolia J, and Woodside C. M (1995) - *A Toolset for Performance Engineering and Software Design of Client-Server Systems*, Performance Evaluation (special issue on Performance Tools), vol. 24, 117-136

[Gamma93] E.Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Abstraction and Reuse of Object-Oriented Design*, Proceedings of ECOOP '93, p. 406-431.

- [Gamma95] Gamma E., Helm R., Johnson R. and Vlissides J. (1995) - *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company, Reading, MA
- [Islam95] N.Islam and M.Devarakonds - *An Essential Design Pattern for Fault-Tolerant Distributed State Sharing* - Commun. ACM 39, 10, pp. 65-74, Oct. 1996
- [Johns94] R.Johnson - *An Introduction to Patterns*, Report on Object Analysis & design, May-June 1994
- [Johns95] R.Johnson - *Software Requirements & Specifications - a lexicon of practice, principles and prejudices*, Addison-Wesley, 1995.
- [Klaus97] Klaus Kreft, Angelica Lang (1997) - *Building an Iterator for STL and Standard C++ library*, C++ Report (2), Feb. 1997
- [Linen96] Linenbach T. (1996) - *Reusable Binary associations in C++*, Dr. Dobb's Journal, vol. 21, issue 6, 34-41
- [Meyer96] Meyers S. (1996) - *Effective C++. 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley Publishing Company, Reading, MA
- [Meyer98] Meyers S. (1998) - *Effective C++. 50 Specific Ways to Improve Your Programs and Designs*, Addison-Wesley Publishing Company, Reading, MA
- [Monro97] R.Monroe, A.Kompanek, R.Melton, D.Garlan - *Architectural Styles, Design Patterns and Objects* - IEEE Software, vol. 14, 1, pp. 43-52, Jan/Feb. 1997
- [Marti98] R.Martin, D.Rhiele, F.Buschmann (editors) - *Pattern Languages of Program Design* - vol. 3 - Addison-Wesley, 1998
- [Neil95] Neilson J., Woodside C.M., Petriu D.C., and majudmar S. (Sep. 1995) - *Software Bottlenecking in Client-Server Systems and Rendezvous Networks*, IEEE Transactions on Software Engineering, vol. 21, no. 9, 776-782

[Niels96] Mark Nielsen, Nick Abdo(1996) - *Applying Design patterns to PowerBuilder*, Dr. Dobb's Journal, June 1996

[Ott96] Ott R. (1996) - *Simple Shared Object: An Architectural pattern for Simple Object Sharing in Object Currents*, Hathaway B. (ed), SIGS Publications, 1(3) Mar 1996

[Panck95] C.M.Pancake - *The Promise and the Cost of Object-Oriented Technology: a Five year Forecast* - pp. 33-49, Oct. 1995

[Petri95] Petriu D.C., Corriveau J-P., Metea L.S., Liu J. (1995) - *Analytic Performance models of Concurrent Object-Oriented Software*, SCE-95-26, Carleton University, Ottawa

[Petri98] D.C.Petriu, X.Wang - *Deriving Software Performance Models from Architectural Patterns by Grph Transformations* - Proc. of the 6-th International Workshop on Theory and Applications of Graph Transformations TAGT'98, pp. 340-347, Paderborn, Germany, Nov. 1998

[Petri99] D.C.Petriu, X.Wang - *From UML Description of High-Level Software Architecture to L:QN Performance Model* - to be presented at the International Workshop on Applications of Graph Transformation with Industrial relevance (AGTIVE'99), Kerkrade, The Netherlands, Sept 1-3, 1999

[Plaui97] P.J.Plauyer (1997) - *Container Adapters*, C/C++ Users Journal, <http://www.cuj.com>, June 1997

[Preew94] Pree W. (1994) - *Design Patterns for Object-Oriented Software Development*, Addison-Wesley Publishing Company, Reading, MA

[Pure95] Pure Software (1995) - *Quantify User's Guide* -Pure Software Inc., Sunnyvale, CA, 1995

[Rumba91] Rumbaugh J. Blaha M., Premierlani W., Eddi F. and Lorensen W. (1991) - *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, New Jersey

[Schm94] D.C.Schmit - *ACE: an Object-Oriented Framework for Developing Distributed Applications* - Proc. of the 6th USENIX C++ Technical Conference, Cambridge Mass., April 1994

- [Schm94a] Schmidt D.C. and Stephenson P. (1994) - *Achieving Reuse Through Design patterns: A Case Study of Evolving Object-Oriented System Software Across OS Platforms*, Proceedings of the 3rd SIGS C++ World Conference 1994
- [Schm95a] D.C.Schmit - *Experience Using Design pattern to deveop Reusable OO Communication Software* - Commun. ACM 38, 10, pp. 65-74, Oct. 1995
- [Schm95b] D.C.Schmit - *Reactor: an Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching* - in Pattern Languages of Program design (J.O.Coplien and D.C.Schmidt eds.), Addison-Wesley, 1995
- [Schm96] D.C.Schmidt - *Acceptor and Connector: Design patterns for Initializing Communication Services* - in Proc. of 1st European pattern Languages of Programming Conference, july 1996
- [Spitz98] B.Spitzbnagel, D. Garlan - *Architecture-Based Performance Analysis* - Proc. Of the Int, conference on Software Eng. and Knowledge Eng. (SEKE'98), pp.146-151, 1998.
- [Shaw96] M.Shaw - *Some patterns for Software Architecture* - in Pattern Languages of Program design, Vol. 2 (J.Vlissides, J.Coplien and N.Kerth eds.), pp. 255-269, Addison-Wesly, 1996
- [Smith90] C.U.Smith - *Performance Engineering of Software Systems* - Addison-Wesly, 1990
- [Souku94] Jiri Soukup (1994) - *Taming C++: pattern Classes and Persistence for large projects*, Addison-Wesley Publishing Company, Reading MA
- [Souku97] Jiri Soukup (1997) - *Implementing Patterns* - <http://www.Codefarms.com/papers/patterns.html>, Oct. 1997
- [Stanl93] Stanley B. Lipmann (1993) - *C++ Primer*, 2-nd edition, Addison-Wesley Publishing Company, Reading, MA
- [Stanl96] Stanley B.Lipmann (1996) - *Inside the C++ Object Model*, Addison-Wesley Publishing Company, Reading MA

[Vliss95] John M. Vlissides, James O. Coplien, Norman Kerth (1995) - *Pattern Languages of Program Design* (vol. 2)

[Vliss97] John Vlissides (1997) - *Type Laundering*, C++ Report (2), Feb. 1997

[Wild96] Fred Wild (1996) - *Instantiating Code patterns*, Dr. Dobb's Journal, June 1997

[Willi98] L.G. Williams, C.U. Smith - *Performance Evaluation of Software Architectures* - Proc. of the First Int. Workshop on Software and Performance WOSP'98, pp. 164-177, Santa-Fe, USA, 1998.