Enhanced modelling and design concepts for software systems development: a collection of generative design patterns

OLUWATOYIN aYODEJI OBIKOYA

A thesis submitted to the University of Huddersfield in partial fulfilment of the requirements for the degree of Doctor of Philosophy

The University of Huddersfield

March, 2018

Copyright statement

1. The author of this thesis (including any appendices and/or schedules to this thesis) owns any copyright in it (the “Copyright”) and s/he has given The University of Huddersfield the right to use such copyright for any administrative, promotional, educational and/or teaching purposes.
2. Copies of this thesis, either in full or in extracts, may be made only in accordance with the regulations of the University Library. Details of these regulations may be obtained from the Librarian. This page must form part of any such copies made.
3. The ownership of any patents, designs, trademarks and any and all other intellectual property rights except for the Copyright (the “Intellectual Property Rights”) and any reproductions of copyright works, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions

# Abstract

A generative development pattern language is still a largely unexplored area of software development even though it is arguably the logical next step in the research into design patterns. This is because while classical design patterns as defined in several pattern catalogues are mostly static components of reusable design frameworks or code that describe how to solve particular or specific development problems, they fail to fully describe the relationships these constructs inevitably maintain with other individual design patterns. Whereas, generative design patterns are a consequence of these relationships and as such inevitably detail how these relationships come about. This in effect makes generative patterns dynamic constructs that not only describe how to create itself but can be observed in the constructs they help to create.

The focus of this research has been to populate a generative development catalogue while detailing in explicit detail how these combinatory structures come about. In the process, this thesis details a combination mechanism that merges the primary methods of constituent patterns and also recognises that the choice of which patterns are combinatory is a lot more complex than previous research indicated and requires a consideration both the relationships the patterns maintain with each other but how their function affects those relationships.

Using Wilson’s generative pattern framework, these generative constructs are not only defined but evaluated to determine their efficacy and identify any advantageous behaviour that results from their development.

# Table of Contents

[Abstract 3](#_Toc510067103)

[Table of Contents 4](#_Toc510067104)

[List of Figures 7](#_Toc510067105)

[List of Tables 9](#_Toc510067106)

[Dedications and Acknowledgements 10](#_Toc510067107)

[List of abbreviations 11](#_Toc510067108)

[Chapter 1 12](#_Toc510067109)

[1.1 Introduction 12](#_Toc510067110)

[1.2 Generative Programming 16](#_Toc510067111)

[1.3 Research domain and problem 17](#_Toc510067112)

[1.4 Research vision and main objectives 19](#_Toc510067113)

[1.5 Research hypothesis and assumptions 21](#_Toc510067114)

[1.6 Contribution to Knowledge 23](#_Toc510067115)

[1.7 Approach to research 23](#_Toc510067116)

[1.8 Thesis structure 26](#_Toc510067117)

[Chapter 2 29](#_Toc510067118)

[2.1 Introduction 29](#_Toc510067119)

[2.2 Design patterns 31](#_Toc510067120)

[2.3 Pattern relationships 33](#_Toc510067121)

[2.3.1 Overview of Zimmer’s work 33](#_Toc510067122)

[2.3.2 Overview of Noble’s work 36](#_Toc510067123)

[2.4 Patterns co-operation (composition) vs pattern combination (compound) 40](#_Toc510067124)

[2.4.1 Model-View-Controller 45](#_Toc510067125)

[2.4.2 Strategy Pattern combined with Factory Method 47](#_Toc510067126)

[2.5 Generative Design Patterns 50](#_Toc510067127)

[2.5.1 Overview of Wilson’s work 51](#_Toc510067128)

[2.6 Developing a generative design pattern 54](#_Toc510067129)

[2.6.1 UML notation 55](#_Toc510067130)

[2.7 The state of the art as concerns metrics 57](#_Toc510067131)

[2.8 Conclusion 60](#_Toc510067132)

[Chapter 3 63](#_Toc510067133)

[3.1 Introduction 63](#_Toc510067134)

[3.2 Methodical Framework 64](#_Toc510067135)

[3.3 Design Science Research 65](#_Toc510067136)

[3.3.1 The Three-Cycle View of Design Science Research 66](#_Toc510067137)

[3.4 Design Science Research Methodology 67](#_Toc510067138)

[3.4.1 Research Phase one (RP1) 69](#_Toc510067139)

[3.4.2 Research Phase two (RP2) 72](#_Toc510067140)

[3.4.3 Research Phase three (RP3) 79](#_Toc510067141)

[3.5 Conclusion 82](#_Toc510067142)

[Chapter 4 83](#_Toc510067143)

[4.1 Introduction 83](#_Toc510067144)

[4.2 Generative patterns 83](#_Toc510067145)

[4.2.1 Pattern Combinations 84](#_Toc510067146)

[4.3 Non-Combinatory Patterns 122](#_Toc510067147)

[4.3.1 Visitor as a non-combinatory pattern 122](#_Toc510067148)

[4.4 Conclusion 132](#_Toc510067149)

[Chapter 5 133](#_Toc510067150)

[5.1 Introduction 133](#_Toc510067151)

[5.2 Evaluation strategy 135](#_Toc510067152)

[5.3 Framework for Evaluation in Design Science (FEDS) 136](#_Toc510067153)

[5.3.1 Application of FEDS 139](#_Toc510067154)

[5.4 Conclusion 166](#_Toc510067155)

[Chapter 6 168](#_Toc510067156)

[6.1 Introduction 168](#_Toc510067157)

[6.2 Thesis Summary 168](#_Toc510067158)

[6.3 The Contributions of the Study 172](#_Toc510067159)

[6.4 Critiquing the Study 172](#_Toc510067160)

[6.5 Implications of the findings 174](#_Toc510067161)

[6.6 Conclusion 175](#_Toc510067162)

[Chapter 7 176](#_Toc510067163)

[7.1 Introduction 176](#_Toc510067164)

[7.2 Investigating non-combinable design patterns 176](#_Toc510067165)

[7.3 Stress Testing the identified generative patterns 177](#_Toc510067166)

[7.4 Generative pattern CASE tool development 178](#_Toc510067167)

[7.5 Conclusion 179](#_Toc510067168)

[References 180](#_Toc510067169)

[Bibliography 203](#_Toc510067170)

[Appendix A 204](#_Toc510067171)

[Appendix B: Short-listed pattern combinations 206](#_Toc510067172)

[Appendix C: Templated Factory Scenarios (A \* B) 212](#_Toc510067173)

[C.01 Scenario 1 212](#_Toc510067174)

[Appendix D: Template and Factory Scenarios (A + B) 215](#_Toc510067175)

[D.01 Scenario 1 215](#_Toc510067176)

[D.02 Scenario 2 218](#_Toc510067177)

[Appendix E: Prototyped Template Scenarios (A \* B) 220](#_Toc510067178)

[E.01 Scenario 1 220](#_Toc510067179)

[E.02 Scenario 2 223](#_Toc510067180)

[Appendix F: Prototype and Template Scenarios (A + B) 226](#_Toc510067181)

[F.01 Scenario 1 226](#_Toc510067182)

[F.02 Scenario 2 229](#_Toc510067183)

[Appendix G: Templated Builder Scenarios (A \* B) 231](#_Toc510067184)

[G.01 Scenario 1 231](#_Toc510067185)

[G.02 Scenario 2 235](#_Toc510067186)

[Appendix H: Template and Builder Scenarios (A + B) 239](#_Toc510067187)

[H.01 Scenario 1 239](#_Toc510067188)

[H.02 Scenario 2 242](#_Toc510067189)

# List of Figures

[Figure 1.1: Top-down view of the research process 24](#_Toc509806242)

[Figure 2.1: Relationships between the GoF patterns (Zimmer, 1995) 35](#_Toc509806243)

[Figure 2.2: Zimmer's operational layers showing the identified pattern relationships (Zimmer, 1995) 36](#_Toc509806244)

[Figure 2.3: Abstract Factory Design Pattern (Bevis, 2012) 42](#_Toc509806245)

[Figure 2.4: Prototype Pattern (Bevis, 2012) 42](#_Toc509806246)

[Figure 2.5: Abstract Factory pattern working with prototype Pattern 43](#_Toc509806247)

[Figure 2.6: Pluggable Factory Pattern-Abstract Factory and Prototype Patterns Combined (Vlissides, 1999) 44](#_Toc509806248)

[Figure 2.7: The Model-View-Controller Framework (Freeman & Sanderson, 2011) 46](#_Toc509806249)

[Figure 2.8: Strategy pattern and Factory method combined 48](#_Toc509806250)

[Figure 2.9: Composite and decorator (Wilson, 2008) 50](#_Toc509806251)

[Figure 2.10: Structure of a generative design pattern (Wilson, 2008) 53](#_Toc509806252)

[Figure 2.11: UML Notation for Association, Aggregation, and Composition (Gomaa, 2011;2010) 57](#_Toc509806253)

[Figure 3.1: Relationship between research phases and the three-cycle view of Design Science Research 65](#_Toc509806254)

[Figure 3.2: The Three-Cycle view of DSR (Hevner, 2007) 67](#_Toc509806255)

[Figure 3.3: Design Science Research Methodology (DSRM) (Kuechler & Vaishnavi, 2008) 68](#_Toc509806256)

[Figure 3.4: Prototype pattern showing the pattern method 77](#_Toc509806257)

[Figure 3.5: Prototype Pattern Showing the Pattern Method 77](#_Toc509806258)

[Figure 3.6: Prototype pattern combines Abstract Factory 78](#_Toc509806259)

[Figure 3.7: Pluggable factory (combination of prototype and Abstract Factory Patterns 78](#_Toc509806260)

[Figure 4.1: The Factory Method (dofactory, 2017) 88](#_Toc509806261)

[Figure 4.2: The Template Method (dofactory, 2017) 93](#_Toc509806262)

[Figure 4.3: Scenario implemented via Templated Factory 98](#_Toc509806263)

[Figure 4.4: The Prototype Pattern (dofactory, 2017) 104](#_Toc509806264)

[Figure 4.5: Scenario implemented via the Prototyped template 107](#_Toc509806265)

[Figure 4.6: The Builder Pattern (dofactory, 2017) 113](#_Toc509806266)

[Figure 4.7: template method combined with builder pattern 118](#_Toc509806267)

[Figure 4.8: The visitor pattern (Guizzo & Vergilio, 2016) 126](#_Toc509806268)

[Figure 5.1: Static representation of the template and the factory method working together. 142](#_Toc509806269)

[Figure 5.2: Class diagram for the factory component 143](#_Toc509806270)

[Figure 5.3: Template method showing the logger component 143](#_Toc509806271)

[Figure 5.4: templated factory 147](#_Toc509806272)

[Figure 5.5: Static template method working with prototype pattern 150](#_Toc509806273)

[Figure 5.6: Generative artefact (Prototyped Template) 151](#_Toc509806274)

[Figure 5.7: code map for the prototype and Template 154](#_Toc509806275)

[Figure 5.8: Code map for the Prototyped Template generative pattern 155](#_Toc509806276)

[Figure 5.9: Code map for static representation 159](#_Toc509806277)

[Figure 5.10: Code map of generative construct 161](#_Toc509806278)

# List of Tables

[Table 2.1: Some Popular Measurable Quantities in Software Development 59](#_Toc509806432)

[Table 3.1: Comparing the Abstract Factory Pattern against the Prototype pattern 74](#_Toc509806433)

[Table 4.1: Factory Method 89](#_Toc509806434)

[Table 4.2: Template Method 95](#_Toc509806435)

[Table 4.3: Templated Factory 101](#_Toc509806436)

[Table 4.4: Prototype Pattern 105](#_Toc509806437)

[Table 4.5: Prototyped Template 110](#_Toc509806438)

[Table 4.6: Builder pattern 116](#_Toc509806439)

[Table 5.1: Templated Method functional evaluation 141](#_Toc509806440)

[Table 5.2: The logger component implemented with a template method pattern 144](#_Toc509806441)

[Table 5.3: The character creation component implemented with the factory method pattern 145](#_Toc509806442)

[Table 5.4: The client class for the static representation of template and factory methods working together. 146](#_Toc509806443)

[Table 5.5: Client class for the templated factory 148](#_Toc509806444)

[Table 5.6: Prototyped Template Functional evaluation 149](#_Toc509806445)

[Table 5.7: Client code for both static and generative implementations of the template and prototype 153](#_Toc509806446)

[Table 5.8: Table showing the purpose of the templated builder 157](#_Toc509806447)

[Table 5.9: Static AbstractBuilder and Director classes 158](#_Toc509806448)

[Table 5.10: The Generative AbstractBuilder with its protected methods and Director classes 160](#_Toc509806449)

Table 5.11: performance metrics for templated factory…………………………….164

Table 5.12: performance metrics for prototyped template………………………….164

Table 5.13: performance metrics for templated builder…………………………….164

# Dedications and Acknowledgements

First and foremost, I would like to thank God for seeing me through till the end of this research.

I would also like to thank my supervisory team and in particular, Dr David Wilson whose work on generative design patterns I am contributing to and Dr. Gary Allen for all their help, support and feedback during this process. I couldn’t have done this without the input from them and I am indeed grateful.

I am grateful to the university and the academic staff in the department of computing and Engineering for the hard work that made it possible to access all the resources that went into this thesis.

Finally, I would like to thank my father who was always there to encourage me in those times when things got tough. The tough love was appreciated dad.

# List of abbreviations

GOF – GANG OF FOUR

FEDS- FRAMEWORK FOR THE EVALUATION IN DESIGN SCIENCE

DSR- DESIGN SCIENCE RESEARCH

DSRM- DESIGN SCIENCE RESEARCH FRAMEWORK

# 

INTRODUCTION

## Introduction

In the past decade, it has become more and more difficult to distinguish software design from software engineering as a subject. Downey (2009) observes that while software design by itself refers to the process via which base components and their dependencies are brought to bear to develop a blueprint that guides the process of constructing a software system, there has been observed a shift towards a more engineered approach. This approach, generally thought of as software engineering is at its core the application of engineering principles to the process of software creation. Proponents, for and against this shift have argued the benefits and disadvantages of this trend, leading to a realisation of the ever-increasing degrees of overlap between these two fields.

This overlap informs the IEEE (1990) definition of software design as “both a process of defining the architecture, components, interfaces, and other characteristics of a system or component and the result of that process”. This is because, software design in practice, incorporates many aspects of software engineering and as such could be difficult to identify as a stand-alone endeavour. As (Fleischmann et al, 1994) observes, there exists an overlap between the design of a piece of code and the implementation of that piece of code. The realisation of this led to Dijkstra’s (1988) supposition that software design is in fact, the design of a design. This definition agrees with Knuth (1989) who describes as futile, any attempts to design software without prior or concurrent implementation. With this in mind, a limited definition could be given. This definition describes software design as the process via which an agent develops the specification for a software artefact with the aid of primitive components which are bound by specified constraints. These constraints could be environmental, meaning they are as a result of the environment within which the design occurs or they could be procedural, meaning they are imposed on the system by virtue of the process involved in realising the design. Consequently, it could be argued that the term software design is a catch-all term for that activity which conceptualises, frames, implements and commissions a software system. It should be noted that these activities within the context of software development often follow the extraction of a “requirements specification” and precede actual coding or implementation of the project. Due to its placement in the development cycle, software design usually encapsulates the process of planning both low-level algorithm design and the development of a high-level, architecture design (Zhu, 2005).

McDermid (2013) observes that software design is arguably the most difficult part of the development process. This rings true because, for a particular software product, the design of effective technical solutions that meet its user requirements while encapsulating the business logic and satisfying the use cases in an efficient manner is difficult. Henninsson (2001) argues that the process of software development is one of trade-offs and compromises between often conflicting intentions. This difficulty becomes more complex as the software project increases in size, scope, complexity and the number of developers working on it at any given time, each with his or her own subjective assumptions as to how best to address the design and development problems that are bound to come up during the development process.

In an effort to address the subjectivity of the code design process, Gamma, Helm, Johnson, & Vlissides (1995) adopted the concept of design patterns from the works of Alexander in the field of architecture and used this to posit the idea of standardised software design artefacts which not only offered solutions to common development problems but were also generally understood and could be used to convey design and development intent.

These artefacts, called the “Gang of Four” (GoF) patterns, each consolidates the best practices in industry on how to solve common-place software design and development problems while complying with existing object-oriented principles and became the de facto rubric via which effective code design is taught and implemented.

Moving forward in time, numerous studies have been conducted on the subject of design patterns, their advantages and the disadvantages they bring to software design and development. Some of these advantages and disadvantages are expressed as follows:

The advantages of design patterns:

* Design patterns, to a large extent, make use of universal object-oriented concepts like decomposition, inheritance, and polymorphism. This forces practitioners to implement or at the very least follow these principles in implementing their code (Laplante, 2007).
* It has been argued that design patterns improve the software development process due to their reusable nature. This argument insists that viewing design patterns as discrete elements that could be applied at will within the context of a software development problem makes the process relatable. This is because it is then possible to boil down any software application to a system of cooperating design patterns (Zlobin, 2013).
* Design patterns encourage rapid development and documentation of coded applications. Since they function as problem-solving tools, design patterns can be prescribed for individual development problems, significantly reducing production time.
* Almost all common design patterns make use of descriptive naming conventions. This allows each pattern to capture the basic idea of its inner workings and its use in relation to a specific scenario.
* Furthermore, this descriptive naming trend extends to the naming of the individual components that make up a pattern, allowing for easier understanding and identification of specific components.

The disadvantages of design patterns:

* The described design patterns do not always translate into implementation. This is because there is a noticeable disconnect in how a number of design patterns are described versus how they are implemented. This is due to the subjective nature of the patterns which allows for widespread and varying interpretation. Consequently, the patterns often do not directly lead to code reuse.
* Design patterns as described in the GoF handbook are deceptively simple in that the solutions they define for particular but discrete problems are often times similar. This leads to significant amounts of confusion as to which patterns to implement and when to apply them (Laplante, 2007).
* In an effort to deliver efficient code designs, there is a tendency to overprescribe and oversubscribe to the use of these pattern artefacts. This leads to a phenomenon called pattern overload wherein indiscriminate use of design patterns results in adverse or negative code performance (Laplante, 2007).
* Also, the process of implementing or integrating design patterns into a software development process is a human-intensive activity that requires considerable experience and know-how on the part of the developer. These relate to issues of cost of development to both the developer and the users the software is being developed for.
* Furthermore, and crucially important to this thesis, the described patterns do not provide a clear blueprint for situations that require the use of two or more patterns.

Consequently, though these patterns adequately encapsulate the knowledge of numerous programmers into reusable elements that developers could adopt in developing solutions to the problems they encountered, there remains the problem of showing how these patterns work together (Wilson, 2008). Wilson (2008) observes that although the GOF book hints at the possibility of relationships existing between patterns in its section on related patterns, it noticeably does not go into detail as to the nature of such relationships, how these relationships could be implemented and what these relationships mean for the design endeavour. This is a rather significant omission which meant that the pattern elements identified in the book, while good for understanding design and indeed development decisions of particular sections of code, could not help with understanding the big picture as concerns projects or more fluid attempts at development. As such, there is a renewed demand for artefacts that detail this information effectively especially in the wake of the realisation that in practice, patterns do not work independently of one another. Such information would go a long way in improving the manner by which developers conceptualise a solution based on the problems to be solved and the specification or resources available to them.

## Generative Programming

With trends progressing towards automation, there has been an increased frequency of studies in the areas of automatic code generation. This has spawned fields including that of generative programming which according to (Czarnecki, K., & Eisenecker, U. (2000), "aims at modelling and implementing system families in such a way that a given system can be automatically generated from a specification written in one or more textual or graphical domain-specific languages”. This has had a direct influence on the development paradigm and how reusable elements can be deployed in specific development tasks. Some of this attention has naturally been given to the subject of design patterns.

Since it is generally accepted that these patterns are solutions to common problems (Bevis, 2012; Jezequel, Train & Mingins, 2000; Freeman, Sierra & Bates, 2004), in order to automate the code generation process, there is a need for developers and automation mechanisms to understand not only localised design decisions but also how these decisions translate across multifaceted systems. To achieve this, important questions concerning how pattern elements could be automatically generated must be answered. This has led to discussions about the automatic or generative implementation of design patterns and if they could function as a specification via which system families that address common problems can be evolved (Bonfe, Fantuzzi, and Secchi, 2013). This draws attention to the parallels between generative programming and design patterns. The similarities lie in the fact that both refer to approaches that are intended to address development problems and both speak of frameworks that could be used to address multiple scenario-specific problems.

As a direct result of this, there was a period of observable increase in interest in the field of design patterns especially as it relates to the creation of these generative constructs. Numerous approaches to resolving this can be identified but the passage of time has seen a progressive decline in the amount of research conducted in the subject area with resources instead being dedicated to identifying or even creating patterns in other avenues of development and information technology (Borchers, 2001). As a result, the subject of generativity remains a largely unexplored field in the study and development of design patterns.

Some researchers have argued that this sharp decline can be attributed at least in part to the preoccupation researchers in the industry have with observable outputs (Zou and Peterson, 2016;2015). This approach is largely an "end justifies the means" approach to software design which is not without its faults. This approach takes little cognizance of the degree of analysis and design involved in developing scalable systems that accommodate the complex and demanding requirements of modern applications.

## Research domain and problem

Having realised limitations of classical design patterns as detailed in Section 1.1, this research goes on to focus on the problems associated with the lack of information on how multiple design patterns work together. This is a crucial step for design patterns especially since in almost all cases, design patterns are required to cooperate with one another to achieve a particular goal or task. This problem forms the major research problem of this thesis.

There is a need for clarity though when determining to what extent design patterns can and should work together. As such, this chapter goes on to provide clear and concise definitions of what is meant by “patterns working together” as this would further help define the particular problems this research has set out to address.

The clause “working together” connotes the phenomena of cooperation between two or more entities to achieve a prescribed goal. As concerns software patterns, it refers to two variants of pattern symbiosis.

On one hand, pattern A remains distinct from pattern B and they come together forming a relationship that allows them to achieve a prescribed objective. For purposes of simplicity, this would be defined as follows:

“Pattern A + Pattern B = Pattern A + Pattern B”

As is the case in mathematics, the resultant entity is one that is wholly composed of both A and B. This scenario refers to co-operation between patterns.

On the other hand, Pattern A and Pattern B become one in a manner that the resultant artefact or any emergent behaviour exhibited by this artefact achieves a prescribed objective. This is denoted as follows:

“Pattern A \* Pattern B = Pattern AB”

Here the resultant entity is indistinguishable or inextricably composed of both pattern A and pattern B. Vlissides (1998) refers to this as a compound pattern.

Compound design patterns are coarse-grained patterns made up of a set of finer-grained patterns (Kamoun et al, 2014). These patterns detail the effects of applying multiple but varying patterns together. Kamoun et al (2014) see them as an assembly of design patterns that are simultaneously applied to solve a specific problem in a particular program or are implemented together in order to establish a specific set of design characteristics. Alternatively, the design patterns that constitute a compound pattern could represent or define a set of related features that are provided by a particular program or environment meaning that the concurrent application of patterns establishes a "solution environment" that is only possible due to this combination of tools and technologies.

Having defined these entities, the term “working together” that was used to characterise the relationship between patterns described earlier in the chapter refers to and emphasises a manner or modus of combination whereby compatible traditional design patterns become compound generative patterns which can then be defined with the aid of a generative pattern framework developed by Wilson (2008). To achieve this degree of focus, there is a need to explore the advances made in identifying the contributions a generative artefact would afford the development process.

Having introduced the idea of design patterns, generative programming, and compound patterns, it is possible to identify the main contention of this research’s problem domain.

Czarnecki and Eisenecker (2000) draw parallels between the concepts of generative programming and those of a generative pattern. Their argument identifies the existence of a problem space, a solution space and finally the configuration knowledge required to implement specific but appropriate solutions to particular problem types. This issue of configuration knowledge is an aspect of the paradigm that has not been fully explored in previous studies on patterns both traditionally and in the scope of generative programming and forms the main research problem tackled by this research topic.

## Research vision and main objectives

This Ph.D. research strongly opines that although there exists a multitude of research material concerning the nature of design patterns, how to identify them and when and where to implement them, there is a definitive lack of discussion that details how these patterns would work together. This is surprising, especially when one considers that many of these patterns do not exist in isolation but within programs or applications that employ numerous patterns in solving the specific problems they are created to solve. A discussion of how patterns relate to each other and could work together would go a long way in revealing how these patterns can be abstracted from the systems they are used to build. Such a possibility hearkens to one of the principal requirements of generative design and indeed generative programming. This refers to the automatic generation of source code from supplied frameworks and design constraints. This is because an understanding of how patterns work together to create advantageous emergent behaviour or functionality could streamline the process of development and automation. An understanding of these relationships (composited patterns and compound patterns) would greatly improve developer productivity and application scalability.

Furthermore, due to the fact that these artefacts are by themselves reliant on the complexity of inputs that define them, changing or varying these inputs would change or vary the type of code generated allowing similar templates to be used across wide-ranging application types with the only difference being the type of inputs supplied.

Consequently, the research vision is the propagation of the idea that software design can be boiled down to the permutation of a set of tools or design artefacts that detail how “combinable” pattern sets work together. These tools are termed generative design patterns.

As a research problem, the realisation of this vision forces one to address a number of questions. These are:

1. What pattern types are combinable?
2. How to identify these combinable pattern types?
3. How are these combinations realised?
4. What benefits, if any, do these combined patterns afford developers in the field?

Thus, the main objective of this research is to develop artefacts that result from the combination of traditional design patterns, but which detail how they come about in a manner that solves some of the important development problems encountered in developing applications.

To realise this, a structured view of how design patterns work together is needed, as this would influence how any artefact that is comprised of multiple patterns working in consonance would be built.

## Research hypothesis and assumptions

The over-arching research hypothesis in this study suggests that dynamic pattern artefacts referred to as generative patterns perform better than their static counterparts in similar contexts. To prove this, a number of assumptions had to be made. These assumptions are discussed later in this section. This hypothesis was further buttressed by a study of the literature and practical experiences which revealed the potential for derivation and allowed for an investigation of the different pattern relationships that could lend themselves to fluid pattern combination with at least minimal functional breakdown of the individual patterns involved or at most the development of advantageous behaviour. Functional breakdown, in this case, refers to the dissolution of the primary intent of the patterns as defined in the GOF definitions of them.

This research goes on to claim that although certain pattern configurations seem to lend themselves to combination by virtue of the defined relationships they exhibit in relation to certain other design patterns, the manner in which they are implemented defeats the realisation of a generative combination. These patterns, such as the “visitor pattern” which will be discussed in Chapter 4. It will be revealed that by virtue of the manner it operates, the visitor pattern does not allow for observable combination with other patterns and is thus, a stand-alone artefact that would resist the manner of combination proposed in this study.

In summary, the following assumptions are held true as concerns this study and greatly influence the identification of pattern candidates in each iterative research phases discussed in chapter 3 of this thesis.

* Main assumption 1

This study purports that design patterns that exhibit a “combines” or “uses” relationship with other patterns in the gang of four handbook lend themselves to more fluid combination mechanisms.

* Implication 1

This stems from an observation of compound patterns in literature. Previous work in the field has successfully combined the “Composite pattern” with a number of other patterns with which it maintains a “combines” relationship (Wilson, 2008) and the “Abstract Factory Pattern” with the “Prototype Pattern” (Vlissides, 1998). These go to show the potential for patterns that exhibit these relationships to function as generative patterns and as such this thesis would limit itself to pattern relationships that are either defined as “combines” or “uses”.

* Main assumption 2

Traditional design patterns that exhibit a similar relationship are effectively addressing the same problem and therefore any attempts to combine these pattern pairs would result in a duplication of function.

* Implication 1:

This assumption relates to the intent and manner of operation of individual design patterns and maintains that design pattern pairs that exhibit this type of relationship are exempt from generative combination attempts. It is understood that this is but an assumption and not a definitive statement as regards the combination of design patterns especially since compound patterns like the pluggable factory indicate that this might not always be the case. A definitive statement regarding this would require more experimentation which would increase the scope of this study.

* Main Assumption 3

Previous studies in the field suggest that the combination of more than two patterns result in generative constructs that are inherently unstable (Wilson, 2008). This instability manifests in the number of resources used up by the process of implementing the artefact and a violation of more than one of the SOLID principles that would be discussed in Chapter 2 of this thesis.

* Implication 1

The pattern combinations developed during the course of this study would limit constituent patterns to two except in scenarios where combining multiple patterns is necessary to achieve the required results.

## Contribution to Knowledge

This thesis aims to demonstrate the viability of the generative patterns framework as developed by Wilson (2008). To achieve this, it will:

* Expand on the generative pattern catalogue by detailing the process of generative compound pattern formulation or creation.
* Add three generative artefacts to the catalogue
* Identify a category of design pattern types that cannot successfully combine with other design patterns or whose combinations with other design patterns do not qualify as generative constructs within the context of this research.

## Approach to research

Owing to the numerous objectives and contexts this study touches on, a conscious effort is made to implement a uni-methodological framework that relies on a number of overlapping research phases. This framework allows the study to proceed in short bursts which end in the prototyping of any discovered generative artefacts. This approach is not unlike an agile methodology and allows for iterative improvements to be made to the body of work in succession which contributes a great deal to the eventual outputs produced. The entire research endeavour will be broken into three (3) co-related research phases (RPx), as shown in Figure 1.1. Each phase possesses a rigidly defined set of objectives and a context. The reason for this was to allow for each iterative burst to be tailored toward achieving a specific objective and in scenarios where this is discovered to be unfeasible, identify points in the investigative process that allow for the introduction of alternative ideology that would address the outcomes and explore any differences between the expected outcomes and the resultant outcomes. This approach underscores the research context in relation to the employed methodology that will be discussed in detail in Chapter three (3).

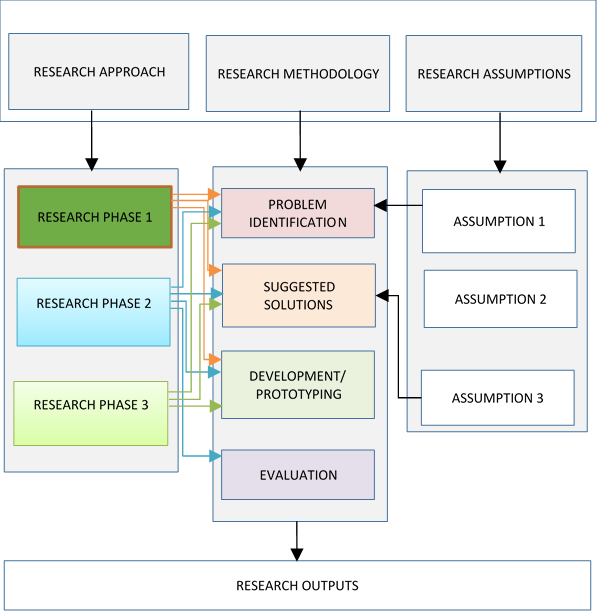


Figure 1.1: Top-down view of the research process

The first of the research stages focuses on exploring a range of contributions to the field of software design, design patterns, generative programming, compound patterns and in particular, the subject of generative design patterns. This was particularly important in order to identify avenues of research that had not been explored or investigated at length and how these absent studies could improve or advance the field. This phase played a significant role in the decision to extend the work of Wilson (2008) on a framework for defining generative design patterns. Wilson’s attempt at the development of generative patterns was a step in creating a common pattern repository of dynamic pattern combinations. This dynamism according to MacDonalds (2002) is one of the major limitations of traditional design patterns. This phase of the project helped identify and define the objectives and overarching goal of this and informs on the evaluation criteria that will be employed later on to determine the validity of the study's finding.

RP1 also investigates the issue of methodology development in the context of this research. It looked at a number of methodologies within the field and how these methodologies could fill any holes in the framework that was being implemented. It was in this phase that the decision was made to employ design science research as the research methodology based on the advantages it promised and its applicability to the type of research being conducted.

The second phase (RP2) focuses on possible solutions to the realised research problems identified in the first research stage (RP1). It was important to analyse how patterns could be combined and how this was influenced by the relationships these patterns already exhibited amongst each other. Any assumptions that stem from the execution of research phase 1 (RP1) would be applied to the development process in phase 2 and thus would significantly influence both the progression and the possible outputs of the phase.

A crucial part of this phase of the project focuses on the actual coding or implementation of the identified combinable pattern pairs and the realisation of functional prototypes of generative artefacts. Consequently, this phase is mostly iterative in nature as it investigates numerous combination mechanisms which are employed to realise effective combinations.

Research phase 3 (RP3) is primarily concerned with analysing and comparing the developed artefacts with more static counterparts. This phase of the project focuses on performance and benchmarking tests which will reveal the advantages and limitations of the realised prototypes and how these artefacts will go on to deliver solutions to the research questions identified in RP1.

## Thesis structure

From a strictly structural point of view, this thesis is comprised of seven chapters. These chapters present the work that was undertaken, and the results in sequence according to the completed phases of the research plan developed at the start of the study.

This thesis is divided into chapters to include the current introductory chapter, a literature review, a chapter examining the methodological process applied in executing the research, a chapter that describes the identified generative artefacts, a chapter for evaluating and testing with the aid of metrics and concludes with a chapter dedicated pronouncing the conclusions of the study and one identifies areas of future research.

Chapter 1, has set the scene by introducing core concepts that are necessary for understanding the body of work and identifying where this research lies in relation to the wider body of research in the area. To that end, it started off by introducing the subject of design patterns. Using its advantages and disadvantages, it explored how the idea of design patterns relates to the subject of generative programming. With this, it was possible to identify the common denominator in these two concepts. This denominator refers to the idea of creating templates from which multiple solutions could be addressed in the case of generative and for design patterns, a solution that addresses a family of design problems. Having done this, the chapter goes on to introduce the subject of compound patterns as a first step in defining the particular problem area and the approach taken to address this problem.

This chapter goes on to identify the main objectives of this study, discuss the main assumptions that influence the direction of research and describe the methodological framework which guides the execution of the research.

In Chapter two (2), the thesis expands on some of the concepts introduced in the previous chapter. It further examines the literature on the subject of design patterns, their advantages and their limitations as dynamic artefacts. This leads to a discussion of the different forms of relationships exhibited by individual patterns amongst each other. This discussion looks at the works of Zimmer, Noble, and Vlissides in this regard and briefly discusses some of the documented relationship types. The chapter then goes on to make a case for the difference between pattern combination and pattern interaction, an argument that is crucial to understanding the main contributions of this research. This distinction feeds into the definition for generative artefacts and in essence generative design patterns. In defining a generative pattern, the chapter looks at Wilson’s work and singles out Wilson’s framework for defining a pattern as a generative artefact. In the process, the study hinges its findings on this framework.

Chapter Three (3) further delves into the methodical approach explained in Chapter 1 of this thesis. It relates this approach to the design science research methodology and formulates a justification for why this methodology is considered appropriate for conducting this research. Having done this, it explains the process via which this research is conducted and divides each phase into its constituent stages. Crucially, it explains the process of combination of individual design patterns and uses examples from literature to buttress this approach to pattern combination.

Chapter Four (4) presents the developed generative artefacts. The chapter, first of all, describes the constituent patterns involved in the combination and details both their advantages and their limitations. This would help identify the functionality these patterns bring to the generative artefact as the chapter goes on to describe the artefacts with the aid of Wilson’s generative pattern framework. This process will look at the advantages and disadvantages of the generative artefacts and discuss any emergent behaviours that are observed as a result of the said combination. Furthermore, the chapter would introduce a category of design patterns which do not meet the criteria required to be considered generative patterns and discuss why.

Chapter Five (5) evaluates the developed generative artefacts. To achieve this, the chapter discusses performance metrics as a tool for benchmarking the performance of the written code. The variance between measures of the same metric in different languages and by different platforms are considered and play a part in comparing the performance of the generative artefacts to their more static counterparts.

The thesis concludes with chapter six (6). Here, a summary of the contents of the study is given. This identifies the themes that are discovered during the research process and discusses how these themes relate to or address the problem areas discussed in chapter one.

Subsequently, Chapter seven (7) goes on to recommend potential areas of future study. These areas, it is argued could offer alternative information on the process of pattern combination or further lend credence to the claims of this research and in effect verify Wilson’s work by further fleshing out the subject area.

# Chapter 2

LITERATURE REVIEW

## Introduction

A cursory exploration of the subject of design patterns in software development literature confirms a point made in Chapter 1. This relates to the suggestion that the study of design patterns is no more at the forefront of research endeavours in the field of computing and is especially true concerning classical design patterns, which have now been absorbed into mainstream development practice for well over a decade. (Scalfani, 2016) argues that this is because software development no longer relies on the pure object-oriented world concepts that led to the development of these patterns in the first place. As a result, the few mentions and studies around the concept of design patterns relate to the processes that guide the decision-making activity that leads to product development or the discovery of design patterns in other avenues of the computing field such as the study of interfaces and networks.

This chapter will present the state of the art as relates to the study of generative design patterns and its development. This discussion will adopt a timeline approach that details the development of pattern artefacts, the relationships they exhibit, how these relationships influence the development of a generative process and the contribution these have had on developing generative pattern artefacts.

In exploring the subject, Section 2.2 gives a definition for “design patterns” in relation to problem-solving in software development. This definition forms an entry point into the discussion of the subject as concerns software development. This section will discuss why design patterns are an important aspect of the software development paradigm and would go on to identify the benefits of these patterns and the influence they exert on the development process. The section concludes by shedding light on the problems a number of researchers have discovered in their studies of design patterns and explain how these limitations necessitate amongst other things, the development of more flexible or dynamic artefacts.

Having defined the concept of design patterns in Section 2.2, the chapter goes on to present a time-lined description of the state of the art as concerns the subject. Section 2.3 examines the relationships that exist amongst individual patterns. The discussion focuses on the works of Zimmer (1995) and Noble (1998). These sections are necessary to understand the work of Wilson (2008) later in the chapter as they influence the direction of research and some of the assumptions made during the course of this study. Furthermore, this section is important as it later influences the choice of pattern combinations that are selected for study and testing. This is a crucial element in determining which patterns could work together generatively with emphasis placed on both the “uses” and the “combines” relationships which by definition, seem to suggest the most obvious system of pattern interaction that could result in a generative compound artefact.

In Section 2.4, we identify clear distinctions between what constitutes pattern interaction and pattern combination and, in the process, allude to one of the major contributions of this research. To do this, the subject of UML notation is briefly examined especially as relates to depicting some of the fundamental principles or pillars of object-oriented programming which are crucial to realising efficient pattern combination. This discussion focuses on the difference between the influence of inheritance and composition on pattern combination and the impact this has on the development of generative compound pattern artefacts.

Section 2.5 then introduces the subject of a generative design pattern. It argues that the idea behind these patterns resulted from the observable limitations inherent in the classical design patterns as mentioned in Section 2.2. This section thoroughly examines and discusses Wilson’s (2008) work on generative artefacts further extending the determined timeline of studies that lead to this research. The section would explore his ideas on pattern combination and conclude with the framework he developed for defining such a pattern.

Section 2.6 concerns itself with the process of creating a generative artefact. The section discusses the works of Menkya (2007) in relation to his observations of the different degrees of modification involved in combining two or more patterns.

The chapter concludes with Section 2.7, which summarises the state of the art as concerns software metrics. These metrics serve as benchmarks or litmus tests via which the performance of the developed artefacts are measured and comparisons made.

## Design patterns

A design pattern is, in its simplest form, a well-defined solution to a particular category of problems that occur within the bounds of a given context (Gamma et al, 1995). This definition recognises that for a pattern to exist, it must possess entities that are related within a problem domain. The problem domain referenced here describes in its simplest terms, the specific problem a developer intends to solve, while the process of resolving this problem identifies the relationships between the entities needed to solve it and also invariably provides more information about the space as well as addresses it. These entities are the pattern’s name, its intent, the problem it solves and solution or how it goes about solving the problem (Alexander, 1979). Design patterns are further explained by Hanmer (2012; 2013) who argues that design patterns are an attempt to encapsulate general solutions to the wide-ranging problems that are encountered by software developers during software development practice into recognisable and reusable structures. Such a representation invariably introduces a set of rules, which prescribe how particular design and development problems could be addressed to give an expected solution. This explanation is summarised by Shalloway & Trott (2010) who maintain that design patterns as understood within the wider development community represent a documentation of the best practices as observed by veteran object-oriented software developers. This distilled knowledge, of “best solutions” to recurring development problems, was obtained via trial and error by developers until Gamma, Helm, Johnson, and Vlissides presented them as recognisable artefacts in the book, “Design Patterns: Elements of Reusable Object-Oriented Software”. The use of these artefacts has proven invaluable as a mechanism for capturing design structure, communicating design and development knowledge and solving recurring problems. The existence of design patterns permits the restructuring of code into reusable modular constructs which implicitly contain particular architectural design themes and sit well within the confines of object-oriented programming observes Freeman, Sierra, and Bates (2004).

Design patterns, for all the advantages they possess, are saddled with several limitations. These limitations have become more apparent with the growing complexity of development projects and revolve around three main themes. These are:

* Design patterns are tailored to solve problems of a particular family of problems and as such, it is difficult to apply a single body of code to executing similar actions or solving the same problems.
* In addition, design patterns are highly rigid and as such, it is difficult to edit or modify them after they have been implemented.
* Furthermore, the absence of common representation via a tool independent environment of identified design patterns makes it difficult to compare and combine them.

This has resulted in a demand for more dynamic development patterns that define the solutions to code problems by presenting an architecture that can be used to manufacture systems. These dynamic patterns would not only pose viable solutions but also describe how these solutions would come about and share resources.

## Pattern relationships

With the proliferation of patterns came the inevitable questions concerning how to best employ and deploy them. This was envisaged by the authors of the GoF pattern handbook and influenced their decision to include a section on pattern relationships which they termed the “related patterns” section in their GoF catalogue.

The lack of discussion in this section has prompted the questions about how these related patterns are related and what these relationships mean for the design and development process. These questions boil down to queries about how systems wherein multiple patterns are implemented would work? And how to identify these relationships.

Several attempts have been made to address this issue and even the authors of the gang of four patterns, taking a page from pattern definitions in architecture included sections detailing possible relationships amongst the described design patterns. These relationships are at best loosely described argues Zimmer (1995) and this prompted him to propose a new classification mechanism for patterns based on the relationships they exhibited with other patterns.

### 2.3.1 Overview of Zimmer’s work

Zimmer (1995) in his work on the relationship between the classical design patterns observed there were questions arising from the relationships one pattern exhibited with other patterns in the catalogue. These relationships, although mentioned in the “see also” section of the gang of four pattern catalogue does little to inform developers on how they could be implemented or come to exist. As a result, developers struggled with understanding and identifying scenarios that were best addressed with the aid of design patterns and how different patterns could work together within the context of a particular code problem. In addressing these issues, Zimmer put forward a classification mechanism that classifies pattern relationships into three main forms. These are:

#### 2.3.1.1 Similar relationship (x “is similar to” y)

This category refers to a group of patterns that provide solutions to relatively similar kinds of problems (Zimmer, 1995). This should not be mistaken to mean they proffer a similar kind of solution as the way by which they solve these similar problems could be vastly different but the underlying problems they attempt to solve are innately similar argues (Noble, 1998). This category of pattern relationship is exemplified by the builder and strategy patterns. The strategy pattern, for example, wraps up different parallel algorithms and exposes them via a universal interface allowing each to be called independently depending on context while the builder pattern separates the process of constructing a complex object from its representation. This has the effect of allowing the same construction process to be used in the creation of different representations. These functions, which seem different are in fact both variant management problems and hence both patterns solve a similar problem but do it in different ways.

#### 2.3.1.2 Uses relationship (x “uses” y)

This category of relationship is the more common relationship category exhibited by design patterns observes (Zimmer, 1995). Noble (1998) argues that this is observed when a pattern X co-opts or makes use of another pattern Y to solve a part, possibly a sub-problem of the problem it sets out to solve. This scenario has been described by Zimmer (1995) as a situation where “the solution of Y (e.g. class structures) represents one part of the solution of X.” This can be observed in the relationship between the composite pattern and the flyweight pattern. Here the flyweight is used to implement the shared leaf nodes of the composite.

#### 2.3.1.3 Combines relationship (x “combines” y)

This category of relationship has been explored in detail in many attempts to combine design patterns (Wilson, 2008). This category refers to a situation whereby both pattern X and pattern Y are involved in solving a particular design problem (Noble, 1998). A popular representation of this is the combination of both the composite and the iterator patterns. Whereas the composite affords a tree-like structure for the constituent parts of a code object and the iterator traverses over a list or collection of objects of possibly diverse types. The combination of both allows the traversal over the composite structures (Zimmer, 1995).

In explain how these relationships are represented across the GoF’s pattern catalogue, Zimmer presented a graphical representation of the relationships he discovered amongst the patterns as shown in figure (2.1).

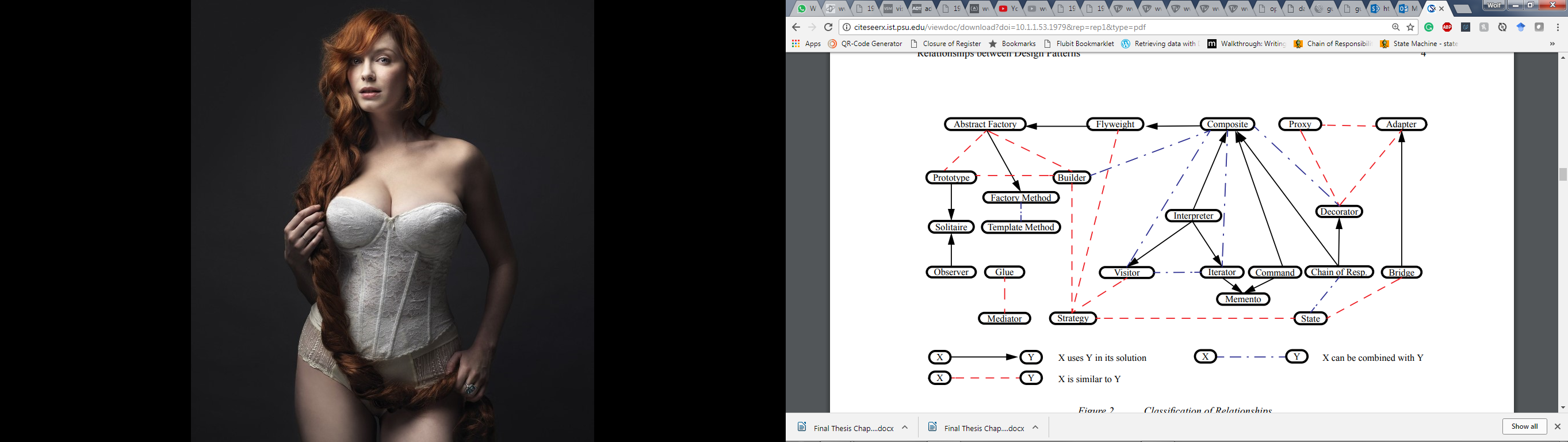


Figure 2.1: Relationships between the GoF patterns (Zimmer, 1995)

Going forward, he factored these relationships into a restructuring of the GoF’s design pattern catalogue which he used to shed light on the different layers at which these patterns operate and how these layers influence the overall behaviour of the developed system. Figure (2.2) depicts this arrangement.

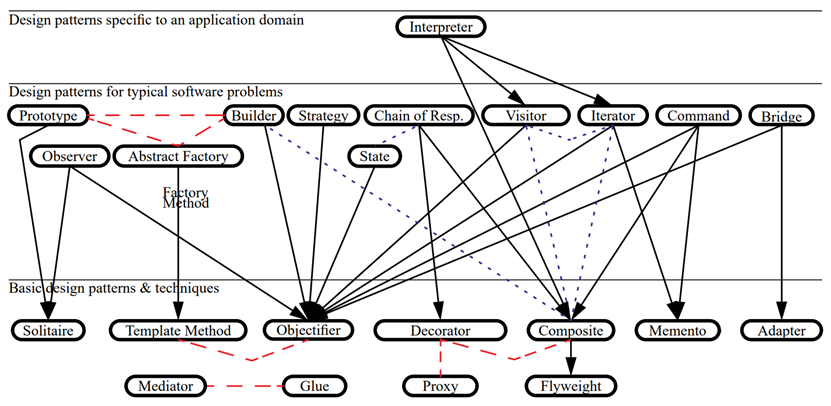


Figure 2.2: Zimmer's operational layers showing the identified pattern relationships (Zimmer, 1995)

In exploring other arrangements, Zimmer (1995) notes that it is theoretically possible to develop layers or categories that focus on the combinations different design patterns have with others in the catalogue and argued that this was of great significance to future work or studies concerning design patterns. This was borne out of the realisation of the shortcomings of design patterns in industry. Here, it was becoming ever clearer that patterns by their very nature as solutions to problems, did not operate in isolation and it was therefore, crucial for developers to understand how different patterns would work together to resolve a problem or achieve particular objectives.

### 2.3.2 Overview of Noble’s work

While the focus of Zimmer’s work on design patterns was the identification of categories into which the existing patterns could be classified in an attempt to understand and extend the functionality of design patterns, Noble (1998) focused on refining the relationships these patterns exhibited with one another. He argued that the relationships exhibited by GoF design patterns were variants of three primary relationships. This meant that there were three fundamental relationships amongst patterns and any other observed relationships could be broken down into some variant of these primary ones. The relationships presented by Noble are as follows:

#### 2.3.2.1 Primary relationships

* **Uses**: According to Noble (1998), this is a situation whereby a large-scale pattern uses a small-scale pattern in solving a particular problem. He argued that this is the case when a larger more global pattern makes use of other patterns in addressing localised problems within its body or scope of operations. He exemplified this relationship with the relationship between the mediator pattern and the singleton pattern. Here the mediator pattern uses the singleton pattern’s ability to restrict or limit class instantiation to one instance, to ensure that mediators are not unnecessarily duplicated in the body of code.
* **Refines**: Noble maintains that this relationship exists when a more specialised pattern refines a more general pattern. He maintains that this refinement takes place when a pattern’s full description extends that of another more general pattern. In this way, one pattern deals with a specialisation of the problem the general pattern is meant to solve. He makes the analogy that the refines relationship is akin to inheritance in object-oriented programming, and goes on to exemplify this with the case of the factory method pattern and the template method patterns. He argues that factory methods are in fact Hook methods deployed by subclasses when specifying the class of the object the template superclass creates upon instantiation, and therefore constitutes a refinement of the operations of the template method.
* **Conflicts**: Here, He argues that one pattern conflicts with other patterns that provide a different solution to the same or similar problems. This means different patterns could solve the same problem in diverse ways and since design patterns are about solving a family of problems this could constitute a clash of methodology.

#### 2.3.2.2 Secondary relationships

Noble (1998) also identified several relationships amongst patterns which he claims fall under a secondary category. That are pattern relationships that are derivatives of the primary relationships or variants of them. This secondary category of relationships includes:

* **Used By**: The “used by” relationship is in many ways a mirror image of the “uses” relationship and as a result, can be defined in the same manner as the “uses” relationship. For example, because the Mediator pattern uses the Singleton pattern, it could be said that the Singleton pattern is in fact being used by the Mediator pattern. The difference comes from the perspective of the designer or observer and is very subjective. The used by relationship is similarly observed in the case where the Iterator pattern is used by either the Interpreter pattern or the Visitor pattern.
* **Refined By**: The “refined by” relationship reverses the “refines” relationship and like the “uses” and “used by” relationships, can be viewed as a mirror image of the “refines” relationship. For instance, if the Factory Method pattern refines the Template Method pattern, then we can say the Template Method is refined by the Factory Method pattern and so on. This relationship like the used by relationship is also largely subjective and as a result is not documented or recorded in a majority of studies into the relationships between pattern forms. Instead, most existing pattern forms choose only to detail the primary refines relationship if they are to consider this type of relationship at all.
* **Variants**: Noble (1998) observes that some problems and solutions types occur more frequently than others, meaning the manner of implementation of particular patterns are more commonplace than others. He argues that these different implementations of the same solution should be stand-alone patterns in their own right and as such should constitute a unique pattern relationship that is separate from their more frequently applied versions or implementations. He goes on to make distinctions between different forms of variants and in the process, identifies two primary categories. These are the solution variants, which put forward varying solutions to a particular problem type and the problem variants, which he argued, define a common solution to varying problems.
* **Variant Uses**: This relationship is an extension of the uses relationship defined by Zimmer (1995). Noble (1998) opines that this relationship should be a stand-alone relationship type and not be defined as a derivative of the “uses” relationship as it facilitates a better understanding of the organisation of patterns. He argues that this is because the relationship defines a scenario where a pattern uses a variant (problem variant or solution variant) of a pattern it maintains a “uses” relationship with.
* **Similarity**: The similarity relationship is explored by Zimmer (1995) and is a derivative of the “related” section of the GoF pattern’s definition. Noble argues that this relationship is a catch-all category for all other pattern relationships not explored by Zimmer. While he agrees that this relationship could refer to patterns that solve similar problems in diverse ways he criticises Zimmer's catch-all approach and claims that there are numerous other pattern relationships that exist within this category and as such indicates that similarity category could be broken down.
* **Combines**: The “combines” relationship mirrors Zimmer's definition of a “combines” relationship. Here two or more patterns come together to solve a problem they cannot individually address. An example of this scenario can be found in the combination of the composite and the builder patterns as implemented by Wilson (2008).
* **Requires**: In this relationship, a pattern leverages the capabilities of another pattern to solve a particular type of problem. This relationship exists when the required pattern is a prerequisite to achieving a solution to the problem the first pattern intends to solve. This relationship closely mirrors the uses relationship with the difference being the necessity of one pattern before another pattern could even be thought to address the specific problem being targeted.
* **Tiling**: This relationship defines a scenario where a particular pattern or group of patterns is applied repeatedly to solve a problem. This relationship is explored extensively by Lorenz (no date) who suggests that the tiling actually makes use of a system of overlaying a particular group of patterns with its variants repeatedly. The interpreter and the visitor patterns can be tiled in this manner to implement a reflexive system.
* **Sequence of elaboration**: This relationship, according to Noble is borne out of the implementation of a sequence of patterns in a manner that progress from low-level patterns to larger more complex patterns. He argues that this relationship is often represented in a stand-alone collection of patterns or the fragments of a particular pattern language. He exemplifies this relationship with the do it yourself reflection, a pattern he argues progresses from the simpler property list pattern to the more complex object system pattern.

Having discussed design patterns and the relationships between individual patterns, it is important to consider the literature on patterns working together as this directly relates to the core deliverables of this study.

## Patterns co-operation (composition) vs pattern combination (compound)

A generic definition of interaction defines it as mutual or reciprocal action or influence (Oxford Dictionary) and implies the existence of two or more entities acting in a manner that sees the actions or operation of one influence the actions or operations of the other. As concerns software, it is a system of action and reaction between the outputs of a particular code entity and the inputs of another which creates a system or a composite of relating patterns. This is because a composite is widely considered as an entity that is comprised of inter-related entities. Combination, on the other hand, refers to an amalgamation or merging of various parts or qualities of separate entities that results in a new entity within which the component elements are no more individually distinct (Yang, Liang, and Avgeriou, 2016). This results in a compound entity whose main contribution is the resultant effect of combining multiple entities and agrees with the views of soapatterns.org (2017) which argues that these pattern entities are considered compound patterns because the relationships they exhibit are directed internally. This means that for compound patterns, emphasis is placed on the relationship the involved patterns acting as a whole, exhibit within a system. This relationship can only be understood from observing the effects the compound has on an external environment. Consequently, whether the involved patterns have dependencies amongst themselves which impact each other is immaterial when dealing with compound entities. As concerns software design patterns, this would see the decoupling of classes in a manner that eliminates the action-reaction phenomena observed when individual patterns work together, allowing for an intrinsic dependence between the classes that constitute the new pattern.

The relationship between these two phenomena is best understood by considering instances of these two forms of pattern interaction. Here we consider the differences between pattern co-operation and pattern combination with the aid of two popular design patterns, the abstract factory pattern and the prototype pattern.

According to Cooper (2000), the abstract factory pattern describes an interface meant for the creation of families of related program objects without the need to specify concrete classes. This allows the program to output specific objects by delegating instantiation calls to its concrete classes whereas the prototype pattern as explained by Bishop (2008), dictates the type of objects that can be created by enforcing the use of a prototypical instance of said object. Any new objects are thus borne from the replication of this initial prototype. This means that the prototype pattern clones an existing object to achieve object creation and as a result of the procedure, the newly created objects are decoupled from the systems that implement them. The class diagrams for both the abstract factory pattern and the prototype pattern are presented in figure 2.3 and 2.4 respectively.

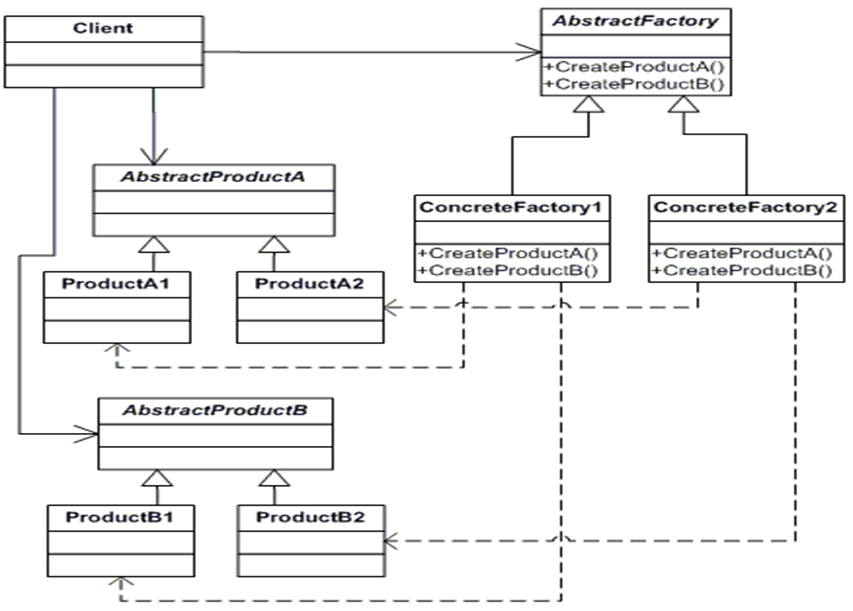


Figure 2.3: Abstract Factory Design Pattern (Bevis, 2012)

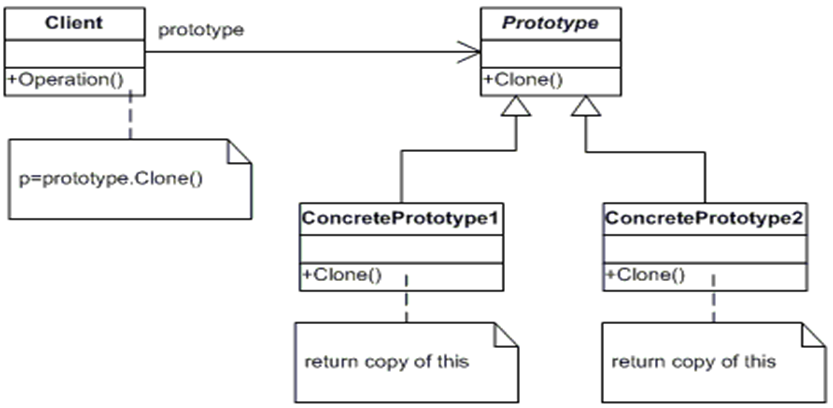


Figure 2.4: Prototype Pattern (Bevis, 2012)

For pattern co-operation, it is possible to observe the influence one pattern (Prototype method) has on the other (abstract factory pattern). Here, the operations of both patterns are very different, and both can be implemented without the other or with limited dependency on the other. In the case of the factory, its main concern is the creation of objects while the prototype acts like a façade pattern and limits interaction with the outside environment to the created copies. Although this presents as an added layer of security for the created objects as they have no awareness of the external environment they operate in, the resulting system is convoluted and difficult to maintain. Figure 2.5 presents a graphical interpretation of the interaction between both patterns.

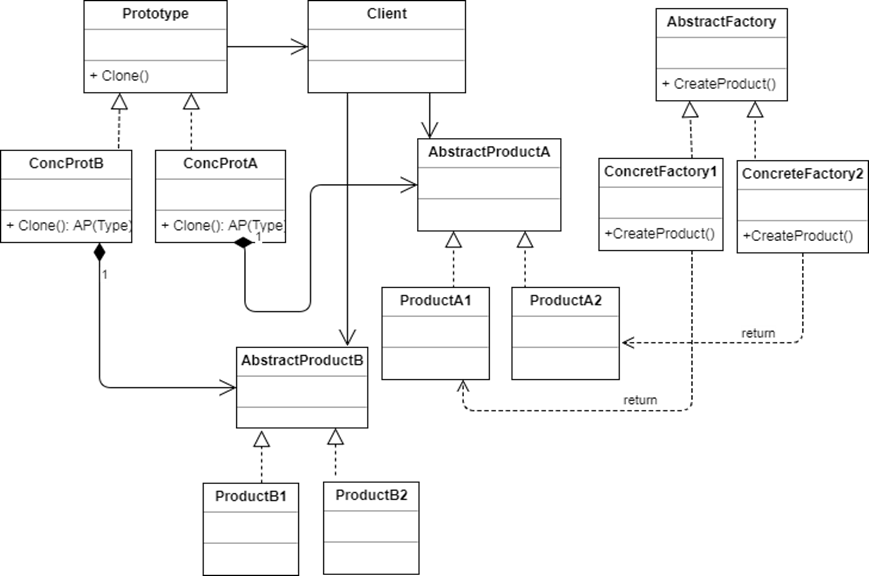


Figure 2.5: Abstract Factory pattern working with prototype Pattern

For pattern combination, the resultant pattern dubbed the “Pluggable Factory” by Vlissides (1999) allows the client vary product types by maintaining a single ConcreteFactory instance and varying the prototypes it copies. This results in a simpler, more maintainable structure but as a consequence, the run-time relationships between the factory and its product classes are slightly more complex.

The “Combination” of both patterns sees the reduction of the abstract factory into a single class that returns copies of the products to be created rather than sub-factories. This maintains the level of decoupling achieved by the abstract factory as it insulates clients from concrete classes and their instantiation while also allowing the created objects to be varied by changing the ConcreteFactory’s prototypes. The class diagram for the pluggable factory is given in figure 2.6 below.

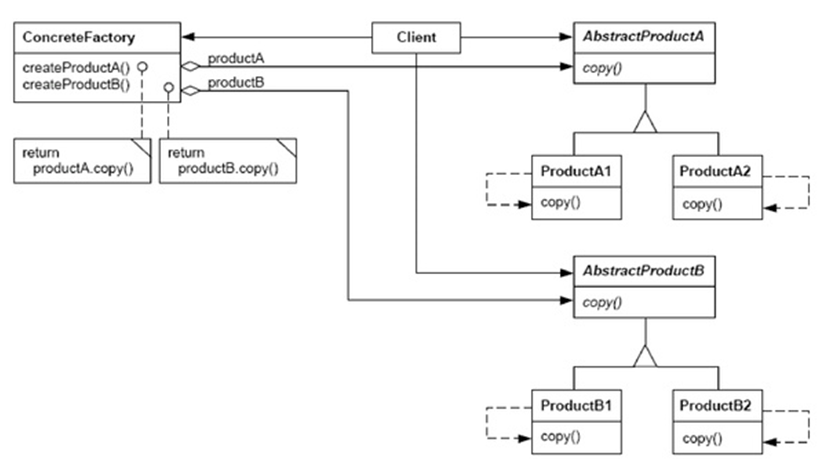


Figure 2.6: Pluggable Factory Pattern-Abstract Factory and Prototype Patterns Combined (Vlissides, 1999)

The abstract factory-prototype scenario described shows that the resources of the abstract factory are co-opted by the prototype’s prototype method resulting in a single class. The entire system of relating patterns evolves into a singular structure or pattern that is more than just two patterns communicating but rather, two patterns making up a whole with one pattern unable to function without the other functioning simultaneously. It is important to mention that Vlissides was one of the creators of the GoF patterns and argued that compound patterns like the pluggable factory were in fact generative structures.

Having discussed the differences between composed patterns and compound patterns it is important to identify what differentiates a compound pattern from a generative pattern. To do this, this thesis presents the difference between a popular compound pattern, the model, view controller framework and a generative compound pattern composed of the strategy pattern and the factory method.

### 2.4.1 Model-View-Controller

The Model-View-Controller is an industry-standard web development framework that is arguably one of the more frequently used compound patterns in the software engineering field. It is an architectural design pattern that separates the modelling of an application into three primary artefacts which are commonly referred to as the model, the view, and the controller. (Esposito, 2014; Deck et al, 2014) observes that each component in the MVC structure is intended to address specific functional aspects of an application.

This allows for the realisation of separation of concerns, one of the primary tenets of software development and as a result, has found significant use in the creation of scalable and extensible projects.

Although the pattern is commonly accepted as a stand-alone framework, a closer inspection reveals it is a compound pattern that is composed of the observer pattern, the strategy pattern and the composite pattern (Freeman et al, 2004).

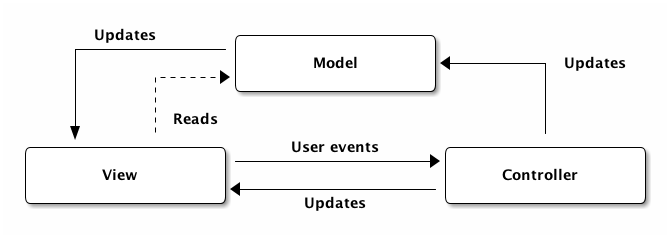


Figure 2.7: The Model-View-Controller Framework (Freeman & Sanderson, 2011)

* The Model: The Model component of the MVC relates to the definition of all data-related logic the system needs to function or is used by the system’s users (Deck et al, 2014). This includes any information that is transmitted between the View and Controller components, and any other data needed for implementation of the business logic. A typical example is transaction information in a commerce application. Such information can be retrieved from a database by both a customer object or a user object and then modified before being pushed back into the database or displayed to the component handling the request. Keeping the system updated is therefore typically achieved with the aid of the observer pattern observes (Freeman et al, 2004). Its ability to monitor the operations of all related components and update dependencies as required without coupling the components to one another allows for the implementation of both single responsibility and open and closed principles of software development.
* The View: The View component of the MVC framework handles all the user interface logic. It determines how information from the database or model is rendered to the user (Amuthan, 2014). It, therefore, outputs pages that are composed of numerous components within the system based on the requests passed to it. The composited nature of the output can be achieved with the aid of GoF’s composite pattern. Consequently, different UI elements can be composed into varying structures based on the varying implementations of the composite pattern. A typical transaction history page in a website or application could be composed from text boxes, tables with transaction information, drop-down menus and links to particular transactions. These elements are components a user would interact with on the application’s front-end.
* The Controller: The controller is in essence to the framework as the internal bus is to the computer in computer architecture. It acts as an interface between Model and View components of the framework to process and manipulate data and business logic, incoming requests and relays these to the view component. (Freeman et al, 2004) observe that the strategy pattern is mostly deployed to achieve this as it permits logic request to determine what algorithms to execute and what views will be built. For example, the Transaction controller could handle all the interactions and inputs from the transaction View to the transaction model. This allows any and all updates to the model to be implemented and if need be, call the view to display information about a particular transaction.

### 2.4.2 Strategy Pattern combined with Factory Method

Another popular example that explains the difference between the generic compound pattern and the generative compound pattern can be expressed with the aid of the strategy pattern and the Factory method as shown in figure (2.8).

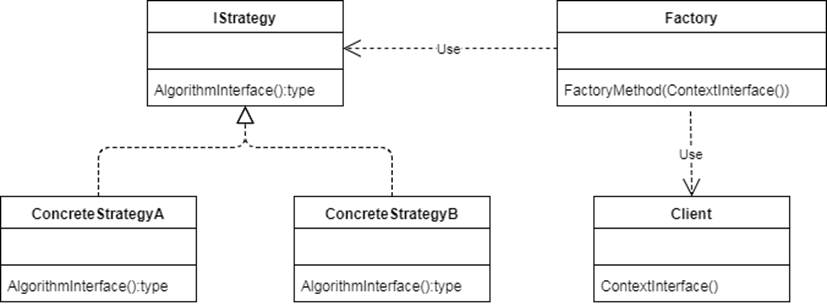


Figure 2.8: Strategy pattern and Factory method combined

This allows the factory method to focus on the instantiation of the appropriate classes, whereas the strategy implementation encapsulates the algorithm that must be executed for a particular scenario. The purpose of the Factory method is to instantiate the appropriate “concrete strategy” class based on its type. Consequently, for the combination, the factory method encapsulates the logic that is required to decide which concrete implementation of the strategy will be supplied. This implementation allows the resulting combination to adhere to the single responsibility principle, as it does not do anything other than instantiate the appropriate classes.

The code implementation for figure 2.8 above can be found under Appendix A in the appendix provided at the end of this thesis.

Looking at both the MVC design pattern and the strategy-factory combination, it is possible to observe that in the former, even though the three combined patterns have become a while, they are still invariably identifiable and the points of combination or overlap merely serve to extend the functionality with the introduction of each pattern whereas, with the strategy factory combination the operations of the factory are more closely interwoven with the operations of the strategy. From the resulting class diagrams, we observe that the strategy-factory combination is more dynamic and would be easier to apply to a varied array of problem scenarios while the MVC is a lot more specialised and rigid allowing for a limited set of actions versus results. This abstractness of the strategy-factory combination even though it is also a compound pattern like the MVC pattern is what makes it generative, and differentiates between a regular compound pattern and a generative pattern.

Although we currently possess several design pattern repositories or pattern catalogues such as the gang of four (GoF) which detail what constitutes a design pattern, information on how these patterns would be used in combination with other patterns is unavailable. Literature reveals the absence of extensive formal specification, detailing or mandating the use of particular design patterns along with their related patterns.

This difference, as exemplified by the case of the strategy and factory method already mentioned, highlights the key features of a generative design pattern and influenced the works of Wilson as concerns the idea of a generative design pattern. He goes on to posit that the combination of the composite and decorator patterns, the composite, command and builder patterns, and the composite, command, decorator and builder patterns fulfil the requirements of a generative construct. Figure (2.9) shows Wilson’s interpretation of a generative combination of the composite and decorator patterns.

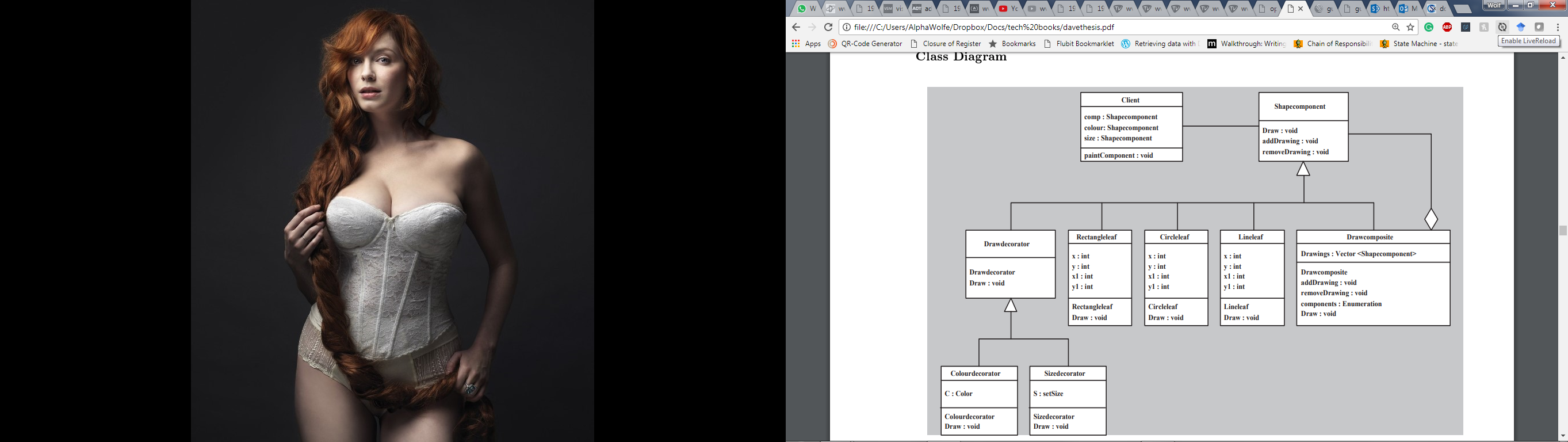


Figure 2.9: Composite and decorator (Wilson, 2008)

## Generative Design Patterns

Generative design patterns as an ideology have been around just as long as the premise of a design pattern. Starting with Alexander, he opined that generative patterns are the abstract definitions of physical patterns which we hold in our minds. The rules we bring to bear in applying these definitions dictate how we envision what constitutes a pattern, what we do with them, how we apply them, what they go together with and the circumstances under which we create them. This implies that the rules that define physical patterns are dynamic and have a consequence. Adapting these ideas into the field of software design, it can then be understood that generative patterns are those patterns which by their very definition detail how they can be created, used and how to use them. In one respect, it is understood that classical design pattern and generative patterns are both patterns, but the primary difference is in the information on the process that defines how these patterns come about, are created and implemented. So, whereas classical design patterns merely exist generative design patterns are dynamic and their definitions and implementation fluid (MacDonald et al, 2002). This agrees with Wilson (2008) who observes that each generative pattern artefact is, in essence, a compilation of rules that describe what must be done to generate the entity which it defines.

Furthermore, ever since Alexander (1977) defined the premise of a pattern as an idea which although useful or applicable in one practical context could be applicable or useful in others, the idea of the generation of such a pattern has taken centre stage in development circles. Citing Alexander’s premise, Welie (2004) submits that the process of traversal from the highest-level patterns to the lowest level patterns informs the idea of “generativity” and although design elements are not strictly hierarchical in a geometrical sense, they could be defined to represent design problems which can be interpreted via hierarchical connotations. This idea expanded on the notion that generative patterns are not only intended to shape system architecture by creating or generating whole or part systems while enforcing the characteristics of a good system, but also teach how such systems are realised (Fant, 2011). In other words, generative patterns help generate other patterns.

This idea of the generativity of design patterns was taken up by a number of researchers, and chief amongst them is Wilson who observed that like the constructs described by Alexander, generative design patterns revolve around the ability of multiple design elements to come together to form a newer element. He goes on to describe this combination scenario with the aid of his generative design framework as discussed in section 2.5.1

### 2.5.1 Overview of Wilson’s work

Wilson (2008), in exploring what it means for a pattern to be generative, suggests that generativity is borne out of the combination of two or more classic design patterns. These combinations would have to be expressed in a manner that details how they are realised and therefore it is crucial that the methodology for defining patterns had to evolve to accommodate their function as generative constructs. He realised that such combinations inevitably changed the physical representations of the patterns themselves but observes that this was not particularly a detriment to the resultant artefacts, as their implementation in several literatures was not an accurate depiction of the initial patterns as put forward by Gamma (1995). This subjectivity in the manner design patterns are implemented has been observed by other researchers.

He goes on to describe this combination scenario with the aid of his generative design framework. This framework presents the structure of a generative design pattern as seen in Figure 2.10 below:

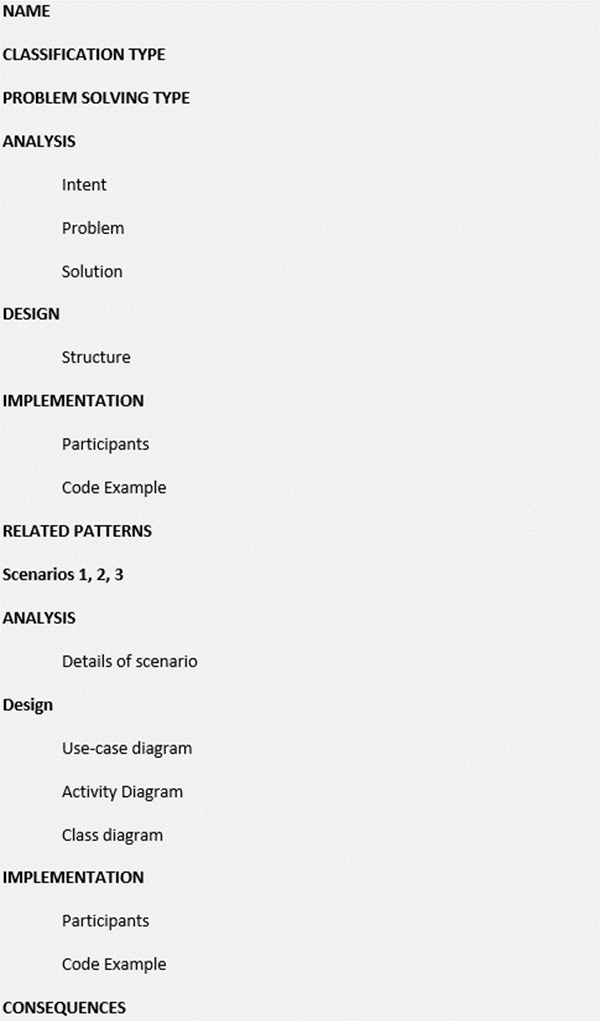


Figure 2.10: Structure of a generative design pattern (Wilson, 2008)

From this framework, it is then possible to identify certain features that any generative design construct’s notation must possess. These are:

* Describe a pattern lifecycle process
* Prominently describe the problem and solution the construct is designed to address.
* At its core, describe the modus via which multiple patterns would work together to solve the particular problems the construct is intended to address

This notation, according to Wilson (2008), adequately describes the emergent system that results from the coming together of different patterns. In this regard, a generative pattern not only solves a particular body of problems but also describes the manner via which the problems are solved.

## Developing a generative design pattern

In attempting to combine different patterns, it has been observed that there is a connection between the structural representations of classical GoF object-oriented design patterns and the manner via which they can be combined with other GoF design patterns. This is because the pictorial representation of these patterns is a crucial part of identifying them in code. A number of researchers have attempted to disentangle this pictorial representation of a pattern from the functional working of the patterns themselves (Gifford et al, 2008) and while this line of discussion is not without its merits especially in light of studies that show that in implementing these patterns, a number of code excerpts do not directly translate into the pictorial forms of the patterns (Wilson, 2008) and the general idea that design patterns are meant to serve as guides or rules of thumb to follow and not concrete representations or instances of code design.

The influence of pattern structure on combinability is extensively explored by Menkyna (2007) in his work on combining aspect-oriented design patterns. He asserts that the combination of aspect-oriented design patterns is considerably influenced by their structural type, meaning it is possible to make statements about the manner in which one pattern could be combined with other patterns based on its structural depiction.

His findings reveal that depending on structure type, combining one design pattern with another might either require extensive modification of the original constituent patterns or might require very little by way of modification. He cites the combination of a “pointcut” design pattern with another “pointcut” as an example of a scenario whereby there is little or no modification to the original combinatory patterns while the combination of a “pointcut” design pattern with an “advice” pattern would require significant modification of the constituting pattern resulting in a completely new pattern structure.

This alludes to the difference between pattern co-operation and pattern combination discussed earlier in this chapter wherein in pattern co-operation, the constituting patterns are wholly represented and can be observed to be interacting. This implies very little modification of the constituent patterns whereas, in pattern combination, there is an intrinsic merging of the composing patterns resulting in an entity with possible new emergent behaviour and would require extensive modification of the constituent patterns. Chapter 5 would explore the topic of combination versus co-operation further as it evaluates the developed generative artefacts.

Another key but subtle aspect of generative design pattern development can be observed in the manner object-oriented principles are applied. A thorough inspection of the programming code generated for both instances of pattern interaction, reveals minute variances in how composition, aggregation, and association are implemented which separate them from one another. These variances come in the form of the enforcement of a composition (has-a) relationship between the class that defines the factory method and those that make up the strategy pattern. This scenario is best understood with the aid of the distinctions as denoted by the UML notation in software development.

### 2.6.1 UML notation

Unified Modelling Language (UML) is a standardised modelling language that enables developers to define, visualise, construct, and document artefacts of a software system (Larman, 2005). Thus, UML makes these artefacts scalable, secure and robust in execution and is a valuable tool in rationalising the decisions involved in object-oriented software development. UML makes use of a graphic notation to create visual representations of software systems. These models could be static, meaning it comprises of both class and composite structure diagrams that emphasise the static structure of systems by depicting a system’s objects, attributes, operations and relations or it could be dynamic, which is used to represent the interactions or collaboration among objects and the changes to their internal states as shown via sequence, activity, and state machine diagrams.

#### Association

Association refers to a group of links having a common structure and common behaviour. It defines a relationship amongst classifiers that indicates that instances of classifiers could either be linked to one other or combined into an aggregate. The “Association” relationship details the relationships between objects of one or more classes. The link referred to here, refers to an instance of an association relationship. The Degree of association denotes the number of objects involved in a connection. This relationship implies that the objects can exist independent of one another with their own independent lifetimes. Consequently, any translation of data is due to the cooperation of the classes.

#### Aggregation

Here, classes are loosely coupled to another principal class. They define part-whole relationships where the parts can exist independently of the whole. This means the classes or objects involved in this relationship can exist independent of one another but belong to a principal class. Therefore, class B is a part of class A but can subsist without class A.

#### Composition

Composition refers to an object-oriented software development principle that asserts that classes or objects could acquire polymorphic behaviour and code reuse by encapsulating instances of other objects that implement a desired functionality rather than inheriting from them (Prehofer, 2001). This means that composition describes part-whole relationships where the life cycle of the part objects is solely dependent on the lifecycle of the whole object. Therefore, class B is an integral part of class A and cannot subsist without class A. This principle plays a significant role in the combination of design patterns as it allows for the implementation of multimodal behaviour in the interface classes of the developed generative constructs. Multimodal behaviour achieved via composition allows for the enforcement of the single responsibility rule even though these classes are required to undertake or implement actions that are indicative of the patterns that make up the system.

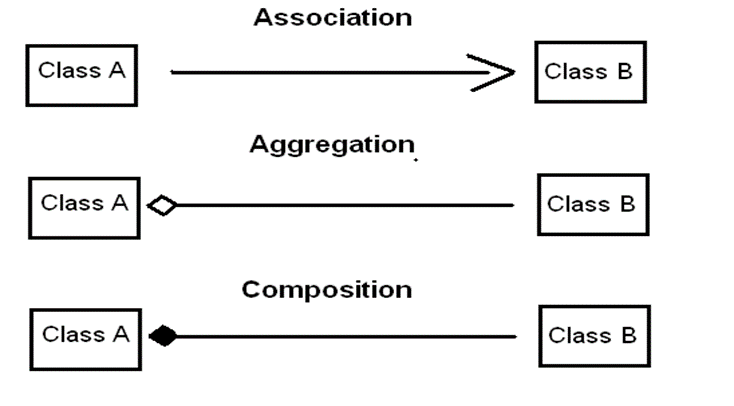


Figure 2.11: UML Notation for Association, Aggregation, and Composition (Gomaa, 2011;2010)

## The state of the art as concerns metrics

A software metric has been defined as a standard of measure that represents the degree to which a software system or process possesses some identifiable property (Olsen, Haug & Bergman, 2001). This definition is necessary to bridge the different but related concepts of metrics and measurement.

Measurement according to (Lockhart, 2012) represent values that indicate an observable quantity or size of an entity while a metric refers to a function that induces a topology on a defined set (Pfleeger, 2008). In essence, the application of a metric outputs a number by which an entity is quantified. These two concepts are often used interchangeably observes (Angood, 2013). Quantitative measurements are an indispensable aspect of almost all sciences and as such, there exists ongoing research efforts by computer scientists and scholars to translate comparable quantifying mechanisms to software development. This would facilitate the recording of objective, reproducible and quantifiable measurements, which may have numerous valuable applications in quality testing, software debugging and performance optimisation (Shepperd & Ince, 1993).

Software development is a complex and subjective process, with a high variance on both the available methodologies and objectives. Consequently, the process of defining or measuring software qualities and identifying quantities necessary for determining a valid measurement metric is difficult. This is especially true when making predictions about the efficacy of a particular design.

In addition, there are the problems associated with determining which metrics carry the most significance in light of the subjective nature of the field. As such, the practical utility of software measurements is determined by the platform used for development purposes. A majority of these platforms, in an effort to focus attention on those metrics they consider the most significant, limit measurable quantities to derivatives of the following function domains

* Scheduling
* Software size
* Code complexity
* Software development effort estimation
* Software quality

These domains, are borne out of an integration of different measurable quantities. Some of the more popular measurable quantities are shown in the table 2.1 below:

|  |  |  |  |
| --- | --- | --- | --- |
| Program size | Number of classes and interfaces | DSQI | Coupling |
| Number of lines of code | Weighted Micro Function Points | Instruction path length | Cohesion |
| Program load time | Instruction path length | Maintainability index | Bugs per line of code |
| Code coverage | Cyclomatic complexity | Program execution time | Halstead Complexity |

Table 2.1: Some Popular Measurable Quantities in Software Development

In response, a number of software developers have opined that simplistic measurements can and do cause more harm than good as they are mostly one dimensional and as such do not reveal the entire picture (Iversen & Nhwenyama, 2006; Clark, 2002). This, they argue is particularly important in a field like software engineering where these quantifiable metrics work together to determine the quality and efficacy of the resultant software product.

Those against metrics also cite numerous studies that suggest that metrics promote negative behaviours in developers. These studies that draw attention to the negative impact of metrics and other measurement paradigms on a developer’s psychology especially as concerns the negative effects on performance due to stress, performance anxiety, and efforts towards cheating the metrics.

Also, they argue that the definition of many quantifiable entities is vague and sometimes open to interpretation. As a result, it is often unclear how these measurement tools arrive at a particular metric or result. This is particularly obvious when one considers the variance in the calculation of particular metrics across different platforms or development environments. For example, lines of code, one of the simpler code metrics are interpreted differently depending on the language and development environment a developer is using.

Others (Basili, Briand & Melo, 1996; Gilb & Cockburn, 2008) seem to indicate that any metric quantity is better than none and as such the current attitudes towards software metrics, sets a standard with which quality can be determined. These proponents argue that they find the proliferation of software metrics to have a positive impact on the value developers place on their work. This prevents developers being undervalued in the industry.

Furthermore, these group draws on some core beliefs that are held at the core of most physical science disciplines. This belief makes the argument that one cannot control that which one cannot measure, a sentiment espoused by James Harrington (as cited in Tzanakakis, 2013).

In recent years, metrics and their associated quantifiable measures have become vital to the software development process. The debate surrounding the efficacy of software metrics aside, evidence presented by Gilb & Cockburn (2008) indicates that software metrics as a tool for determining the efficiency of code has been adopted widely across all manner of institutions from government agencies to academic institutions and as such could be relied on to give an indication of the degree of efficiency any developed artefacts provide.

## Conclusion

This chapter examines the state of the art as concerns the study of generative design patterns. To do this, the chapter first presents a definition for design patterns. It hinged its definition of design patterns in popular literature noting that they are standard ways of addressing common programming problems with an identified and well-defined solution. It observes that these solutions are based on the experience of other programmers and are usually recognised as a good solution to that problem. In defining design patterns, the chapter identifies a number of limitations inherent in traditional design patterns and recognises that as the complexity and scope of development efforts scale, there is a need for more dynamic artefacts, which take into consideration how patterns could work together and the emergent properties that could result from different variants of these combinations.

Recognising that the core of this study’s argument resides in an understanding of the relationships patterns exhibit amongst each other, this chapter discusses the evolution of pattern relationships. In doing this, the chapter discusses the work of Zimmer who first identified and classified pattern relationships into the “uses”, “similar” and “combines” categories in his attempt to describe the relationships traditional design patterns exhibit.

The chapter follows this with an examination of the work of Noble (1998) who refined the classification of pattern relationships and described the direction of these relationships. This direction influenced his classification of these relationships into the used by, refines, conflicts and similar relationship.

Having discussed pattern relationships, Section 2.4 discusses how these relationships influence the discussion and development of generative constructs. To discuss this, it was crucial that parallels are drawn between pattern interaction and pattern combination. We expand on these differences with the aid of core object-oriented principles in section 2.5 and this leads to the definition of generative design patterns in 2.5.

The section discusses the work of Wilson (2008) in defining a generative design artefact. It places emphasis on the pattern notation he developed and discusses how this accurately described the dynamic nature of the generative artefact.

Section 2.6 then describes the process of development of a generative artefact. The review looks at the works of Menkya (2007) and his discoveries about how patterns combine and the effect this combination has on the patterns identifiable structure and the emergent behaviour it exhibits.

The chapter concludes with an examination of the state of the art as concerns development metrics. These are a measure of performance used to benchmark written code. The section defines the different metrics that directly influence the comparisons the study makes between generative artefacts and static combinations of patterns. It also discusses the variances in how different development platforms measure these metrics.

# Chapter 3

METHODOLOGY

## Introduction

Previous research in the field indicates that there are several methodological challenges to contend with in developing generative design constructs (Wilson, 2008; MacDonald et al, 2002). These challenges revolve around the identification and definition of classical patterns that would work together in a manner that addresses specific design problems while meeting the requirements of the generative patterns notation.

In defining patterns, there is a tendency to seek out elegant, repeatable, statistical studies on pattern identification. This, according to Oram and Wilson (2011) is due to the mathematical and scientific origins of computer science which strives to achieve a level of statistical confidence via the simplification of complicated circumstances. This simplification exercise invariably leads to an increase in the frequency of assumptions made concerning the behaviour of systems and constructs observed (Ng, 2016). This is the case with classical design patterns and explains why in numerous instances, the implementation of a particular pattern fails to match with its pictorial representation. This positivist perspective of design abstracts away important features or elements of the pattern resulting in the static nature of the classical design patterns that pervade the field (Eide et al, 2002). To counter this, this study rigorously explores the process of combination amongst patterns, suggesting that this is a step in the development of dynamic patterns whose implementation details can be observed from their definition.

The process via which these artefacts would come about is equally as important as the resultant artefact and necessitates the adoption of research methodologies that cater to the testing and evaluation of both the produced outputs and the process of production of same.

This chapter looks at the methodical framework that guides the execution of this study. This methodology guides the process of data collection, analysis and the eventual development of the arguments this thesis puts forward.

After this, the chapter delves into the theory of the design science research methodology, the chosen research method employed for this research. It, first of all, defines it, then goes on to explain that the design science methodology is implemented with the aid of three cycles which significantly influence the way the three (3) phases of the project is carried out. Emphasis will be made on the fact that the three-cycle view of DRS matches and has a one to one relationship with each phase of the approach to the project as detailed in chapter one and thus guides the translation from one phase to another.

Thirdly, section 3.4 takes each phase in turn detailing the design science process and explaining the outputs of these stages in relation to this research.

The chapter concludes by justifying the chosen methodology and the reason behind its implementation variant.

## Methodical Framework

The methodical framework employed in conducting this research was introduced in Chapter 1. It presents a general overview of how the research activity was conducted and shows that in developing the presented generative artefacts, several iterative phases, RP1 through RP3 will be undertaken. These phases form a very abstract view of the manner of research and contained within each phase is a rigorous implementation of the design science research methodology, the main methodology that guides this research. These phases are structured to ensure that the outputs of each phase, refines, contributes or reinforces those of the phase which came before it. Furthermore, limiting the process to three phases agrees with the three-cycle view of design science research. This, in an abstract sense, allows for a sort of coupling between particular phases and the cycles they encapsulate. How this works in practice will be discussed later in Section 3.4. A visual reference is presented in fig (3.1) to aid understanding and details how three-cycle view of design science research works with the three-phase approach employed in conducting this study.

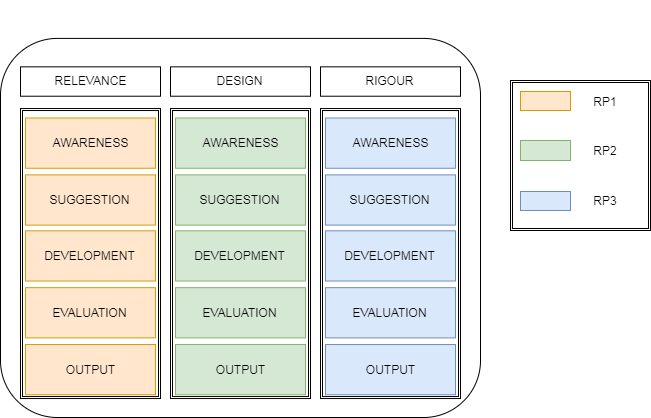


Figure 3.1: Relationship between research phases and the three-cycle view of Design Science Research

## Design Science Research

Design science research is widely regarded as an outcome-based information technology research method, it offers and emphasises specific guidelines for the evaluation of research projects and production iteration within the research. According to both Vaishnavi and Kuechler (2004) and Hevner (2007), design science research is yet another set of analytical techniques for conducting research. It is a particularly useful research tool in the field of Information Systems as it is agreed that as a methodology, it emphasises the more technical aspects of the subject. This is an area which has historically been put to the side as information sciences moved towards more managerial and organisational issues (Orlikowski and Iaconono, 2001). It focuses on the development and performance of developed artefacts with the explicit intention of improving the artefact’s functional performance. The category of artefacts to which design science research is typically applied includes algorithms, human/computer interfaces, design methodologies (including process models) and languages (Benbasat and Klein, 2003). As such, the body of this study which focuses on the development of dynamic artefacts which offer more flexibility to the design process falls squarely within its scope.

As a research methodology, it primarily concerns itself with the processes involved in developing novel artefacts and the usage and performance of these with the aim of understanding and eventually improving on particular aspects of an information system.

### 3.3.1 The Three-Cycle View of Design Science Research

Text Design science research has been argued to embody three (3) closely related cycles of activities (Hevner, 2007). These cycles are:

* **The relevance cycle**: This initiates the design science research process. It focuses on providing an application context that not only identifies the requirements of the research as inputs but defines the acceptance criteria that would be used in the design cycle of the research. These inputs are based on the environment the research is to be conducted in, the problem domain to be considered and the resources available and also define specific acceptance criteria for the eventual evaluation of the research’s results or outputs (Hevner, 2007).
* **The rigour cycle**: This affords the process an awareness of past knowledge relating to the particular research area. Thereby ensuring the project’s objectives are innovative. This cycle is contingent on the ability of the researchers to meticulously investigate and reference the knowledge base. Affording the project some measure of a guarantee that resultant outputs are genuine research contributions and not routine results based upon the application of well-known processes (Hevner, 2007).
* **The design cycle**: This cycle iterates over the core activities of developing and evaluating the produced artefacts. This stage progressively aligns the outputs with the objectives and processes of the research defined in the relevance cycle (Simon, 1998). According to Simon (1996), this cycle is the process of developing design alternatives. These alternatives refer to the different approaches to implementing the numerous inputs from the relevance cycle of the research. This cycle is also responsible for evaluating the outputs of this approach, ensuring that they match the project’s requirements until a satisfactory artefact is attained.

Consequently, Peffers et al (2008) are of the opinion that design cycle is the most important of the three cycles. There is an innate need to maintain equilibrium between the efforts spent in fabricating or developing the outputs and that used in evaluating and evolving the design artefacts.

Figure (3.2) depicts the discussed three-cycle view of the design science methodology.

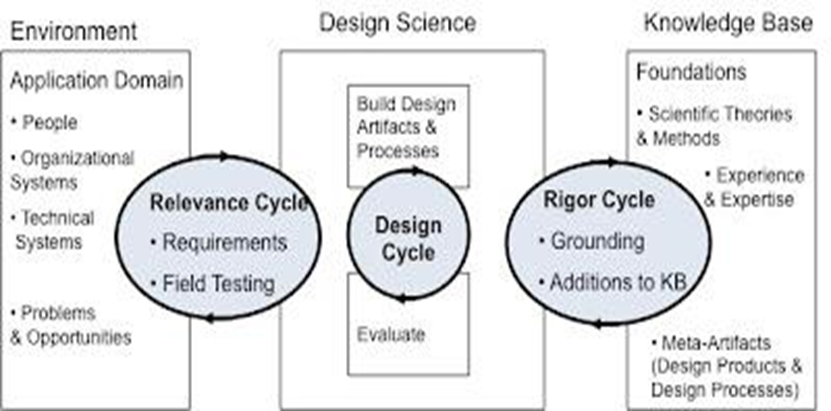


Figure 3.2: The Three-Cycle view of DSR (Hevner, 2007)

## Design Science Research Methodology

Having introduced the concepts that constitute the ideology of design research, this chapter goes on to detail the process for conducting design science research. This is done with the aid of the DSRM model given in fig (3.3).

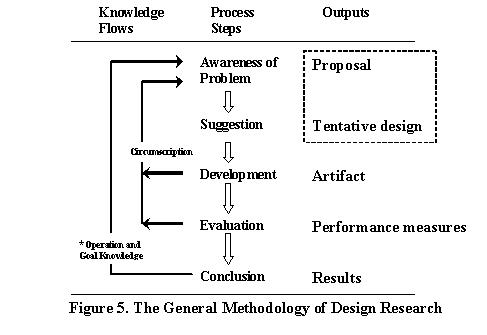


Figure 3.3: Design Science Research Methodology (DSRM) (Kuechler & Vaishnavi, 2008)

The model indicates that the process begins with an acute awareness of the problem to be solved. For the purposes of this study, this aspect of the design research methodology borrows from the intent of the research phase which it is a part of. Thereafter, suggestions for addressing the problem are considered, having been inductively formulated with the aid of available existing knowledge in the subject. This is then followed by the development of an artefact that implements a proposed solution. This is the development stage and results in the creation of prototype artefacts which can then be tested against certain performance metrics. The implemented solutions are then evaluated, detailing any emergent behaviour that has been observed and relating this to the performance expectations in terms of the problems the solution is meant to solve. The stages of Suggestion, Development and Evaluation are frequently performed iteratively, to better deliver more appropriate or accurate solutions. The conclusion indicates the termination of a project. New knowledge production is shown in the figure by arrows labelled Circumscription and Operation of Knowledge and Goal. The Circumscription process is relevant in design science research because it outlines the importance of construction to gain understanding.

Having discussed the process via which the design science research methodology is implemented, this chapter goes on to detail the process of implementation as concerns this particular study.

### 3.4.1 Research Phase one (RP1)

This phase initiates the study. It is primarily concerned with gaining an understanding of design patterns, compound patterns and generative design patterns as a subject area and formulating an action plan that would allow for achieving the objectives of the project. This phase of the study is related to the relevance cycle of design science research in that it asks questions regarding what problems the study is trying to solve. It does this by critically looking at the subject landscape and factoring in variables that might help to more accurately define the nature of the subject being investigated. In discussing this phase, this study looks at the different steps that went into its execution in line with the DSRM discussed in 3.4

#### Problem awareness

An awareness of the existence of a problem to be solved is the foundation of most if not all research endeavours argues Belk (2006). Such an awareness could be borne from observable limitations in the industry or even experience.

As concerns, this study, the initial incursion into the subject area revealed that there exists a demand for dynamic design constructs that actively describe how a problem is solved especially when such a solution necessitates the implementation or at the very least co-operation amongst existing classical design patterns. This demand stems from observable limitations in software development which present static constructs cannot address. These limitations fall into three major themes:

* Inflexibility: Design patterns ascribe a set of solutions to a family of related design problems and it is difficult to generate a single body of code that satisfactorily addresses each problem in the family.
* The lack of a tool-independent representation: A common tool-independent representation mechanism could lead to a shared repository which could make more patterns accessible. Although studies exist that attempt to address this, research in the area has stalled and such notations are not as extensive or populated with enough artefacts to garner them widespread appeal.
* Incompleteness: In defining a design pattern it has been observed that popular notations do not describe how such patterns work in concert with related patterns leaving developers to constantly guess at the implementation mechanics of such cooperative constructs.

#### Suggestions

Having identified the specific information on the nature of the problem to be solved, the next step of the DSRM involves mapping out the route to workable solutions. The process of identifying these workable solutions is largely influenced by the knowledge gained from researching the problem area. This is because a thorough examination of the literature on the subject as conducted in the awareness stage would help formulate direction on how to address the problem. In a situation where there are competing theories that could be pursued, the one which to a higher degree, lends itself to achieving the goals or objectives of the study takes precedence. As a result, a key aspect of the suggestion phase is ensuring that the suggested solutions are measurable as per the predetermined success criteria.

The suggestion stage of RP1 involves identifying the state of the art as relates to previous research in the field. This review of existing literature informed on a number of assumptions that will later guide the execution of RP2. Secondary research on the nature of patterns revealed several attempts to tackle the static nature of design patterns in literature. These attempts focused primarily on the development of new pattern libraries without addressing the limitations of the existing libraries but shed light on the alternative ideologies concerning pattern extension and consequently, the idea of generative constructs that described not only a more dynamic representation of a pattern but also how such patterns collaborate with related patterns within the confines of an application. An example of this is seen in the literature. The works of Zimmer and Noble attempt to classify the relationships they observe are inextricable from the practical implementation of the patterns identified by Gamma. Their approaches to these relied on a restructuring of the pattern catalogue to show these relationships but details about how this restructuring affects the implementation of the patterns are not extensively discussed.

Also, there is an obvious absence of discussion on how these restructuring efforts relate to code dynamism and generativity.

#### Development

The development stage for RP1 was mostly involved with ratifying the discovered ideologies. Emphasis was placed on defining a structure that would guide the execution of the project. The main output of this stage of research was the research proposal which indicated the mode of research, the expected outputs and how these would be measured.

Furthermore, crucial to the execution of the research was an identification of the methodology to be employed in carrying out the research. Design Science research methodology was chosen due to the relationship its process has to the objectives of this particular study. This methodology focuses on the creation of artefacts and how these artefacts would perform which is the core of this research in its simplest form.

#### Evaluation

The evaluation stage was not a major focus in RP1. It was important to understand that the chosen methodology would result in the required outcome and this was understood from the initial literature review that was the core part of the suggestion phase of RP1. This review also influenced the decision to use the performance metrics that will be discussed in chapter 5 as one of the evaluation criteria.

#### Conclusion

Research Phase 1 (RP1) resulted in a blueprint that detailed how the research would be executed. This blueprint is akin to a recipe in that it represents the overall design of the research project and does not go into detail on specific aspects of the research. This allows for a flexible approach to research phase implementation, affording the process an almost agile-like development quality. The various stages of this phase were iterated multiple times to focus and filter the resources required and accurately define the eventual outputs of the phase.

This research phase adhered to the demands of the environment view of the three-cycle view. It allowed this research to identify the application domain, the particular research questions and the objectives that needed to be met to address these problems.

The main output of this research phase is a well-rounded identification and definition of the objectives of the research, the research framework and the methodology to be employed.

### 3.4.2 Research Phase two (RP2)

Having identified the main objectives of the research and how these would be achieved in the evaluation phase of research phase one, it was then important to implement the designs that were identified. This phase will see the identification of the combinatory and non-combinatory patterns and prototype the code examples to show the variance between interacting patterns and combined patterns. This phase is mainly concerned with both the rigour and design cycles of the three-cycle view of design science research. The distinct stages of the phase are presented as follows:

#### Problem awareness

This stage as discussed above is concerned with proffering a specific definition of the research problem and understanding the value of the proposed solution. In this phase, the specific problem to be tackled is the process of implementing the mechanisms for achieving the identified objectives of phase one and how this implementation represents an accurate depiction of the mechanisms involved. The resources coming into this phase are the identified objectives of the research, the identified methodology and an understanding of the premise of the subject of design patterns and Wilson’s approach to design pattern generativity.

#### Suggestions

It has been established that design patterns are an attempt to proffer solution to commonplace design problems. With this understanding, to attempt to combine design patterns, it is crucial to identify commonalities in their definition that would lend themselves to an effective combination.

These commonalities can be identified by examining the problem a pattern is meant to solve or its statement of intent. This can be understood when one considers the compound pattern “Pluggable Factory” as defined by Vlissides (1999).

The pattern is made up of a combination of both the abstract factory pattern and the prototype pattern. Examining these two patterns, it is possible to highlight certain commonalities amongst the pattern

This is shown in Table (3.1) below:

|  |  |  |
| --- | --- | --- |
| Pattern | Abstract Factory | Prototype |
| Problem | Code Modularity and portability is a major issue in modern day programming practice and as such, for an application to be portable, there is a need for it to encapsulate any and all platform dependencies. | There is an emphasis on savings as relates to computer resource usage. This is because modern computer programming more or less considers the cost of creating a new object as high and resource intensive. Hence, application developers are constantly looking for ways of improving code performance. |
| Intent | -Provide a mechanism that supports the creation of families of related or dependent objects without needing to specify each object’s concrete class.  -Function as a hierarchy that encapsulates any and all possible platforms which makes the construction of a suite of product objects to be platform independent and hence modular and portable.  -Eliminate the need for the “new” operator which is considered detrimental to code performance. | -Stipulate what object types are created with the aid of a singular prototypical instance which is cloned each time a new object is to be created.  -This process co-opts an instance of a class, converting it into a breeder class that is used in the creation of future instances.  -The elimination of the need for the “new” operator which is considered detrimental to code performance. |
| Commonality | --Avoidance of the “new” operator  --They are both creational patterns | |

Table 3‑1: Comparing the Abstract Factory Pattern against the Prototype pattern

After comparing the two patterns and identifying commonalities that can be leveraged, it is possible to the address the issue of combination by matching these commonalities.

For instance, in its definition, the intent of the pluggable factory developed by Vslidues (1999) is stated as follows

“Specify and change product types dynamically without replacing the factory instance”.

This connotes a problem whereby the outputs of an object creation process need to be varied in some manner without having to instantiate a new variant of the object.

This directly maps to the identified commonality of the “avoidance of the new operator” and is an indication of the required commonality that must exist between or amongst combined patterns for them to exist.

With this in mind, patterns which exhibited already identified relationships were then compared in an attempt to discover commonalities that could be exploited.

Having identified a relationship between the definition of design patterns and the potential for combination, a number of pattern pairs were shortlisted for possible integration.

This shortlist provided under Appendix B of the appendix was based on the following criteria:

* Identified relationships the patterns exhibit between each other
* Commonalities between the patterns

Further studies into literature revealed the parallels between generative programming, design patterns and compound patterns. This relationship helped to put Wilson’s framework in context. It was, therefore, possible to identify a timeline of events which have significantly influenced the development of the theory and allow for an understanding and an affirmation of Wilson’s work. These developments refer to the works of Zimmer, Noble and Menkya in relation to the relationships unique design patterns exhibit and the works of Vlissides (1999) in relation to compound patterns. It was important to identify how these relationships contributed to the development of the ideology of generativity.

#### Development

At this stage, a review of some pertinent contributions to the subject of generative design patterns is presented. Emphasis is placed on the work of Wilson in his framework of a generative design pattern, the developed artefacts and the process of development that was employed. This singular study serves as a direct precursor to the research area and significantly informs the development stage of this thesis. This stage is particularly important to this study as it serves to position this study amongst the wider subject area, emphasising that this approach proffers a viable argument in the wider discussion and addresses some of the fundamental problems observed in classical design patterns.

Based on the outputs of the suggestion phase, it was possible to break down the typical design patterns into the following:

**Pattern**: This refers to the overall definition of the pattern as given in the GoF handbook. It considers the problem to be solved and how a particular pattern solves that problem.

**Pattern method**: It was possible to identify a main method that embodies the function or purpose of each specific pattern. This main method is responsible for the behaviour, the pattern, when considered as a whole, exhibits. For example, in the prototype pattern, the abstract class encapsulates a clone method which returns objects of the prototype class and makes it possible to call the varying clone methods. Figure (3.4 & 3.5) depicts this:

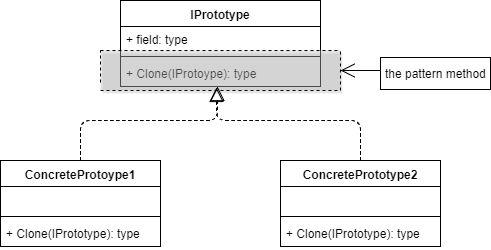


Figure 3.4: Prototype pattern showing the pattern method

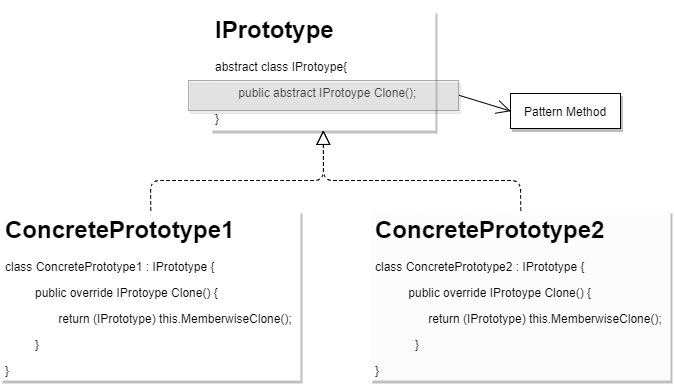


Figure 3.5: Prototype Pattern Showing the Pattern Method

**Output**: The output refers to the result of implementing the pattern. It is not always possible to define the output pictorially as is the case with the visitor pattern and template pattern. In these cases, the output refers to the effect the solution has on the system.

Having identified the components of a classical design pattern, combination would see an integration of the pattern methods present in each pattern. This would allow for the constituent patterns to merge.

In the case of the prototype and abstract Factory, the prototype’s clone method subsumes the factory method.

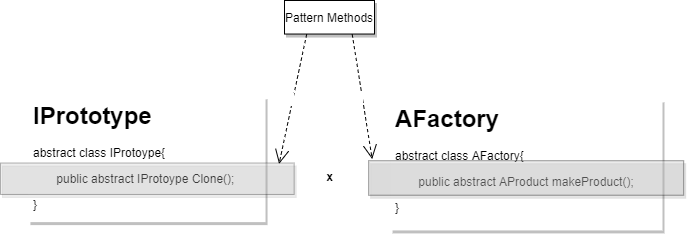


Figure 3.6: Prototype pattern combines Abstract Factory

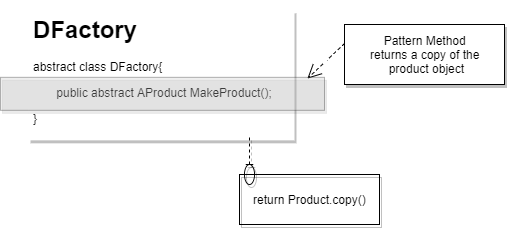


Figure 3.7: Pluggable factory (combination of prototype and Abstract Factory Patterns

#### Evaluation

The evaluation stage of DSRM is crucial to ensuring that the developed outputs are on track to deliver an accurate solution to the identified problems (Venable et al, 2011). In this stage, the observational method was used to initially determine the degree of cohesion of the developed compound artefact. To achieve this, the class diagrams of the resulting compound artefacts are studied, ensuring that there is an observable merger of the abstract classes of the constituent patterns.

In the case of RP2, the defined problem of this stage is the implementation of combination mechanisms to realise the compound patterns. To this end, the shortlisted design patterns are coded in such a manner that merges their functionalities by combining the principal methods attendant to each pattern as discussed in the development stage.

As explained in the definition of the DSRM, the evaluation stage is iterative, with each successive iteration further streamlining and integrating the constituent design patterns.

Having combined certain pattern pairs, the limitations of this combination mechanism was revealed. These have to do with the interaction and use of object types within the developed patterns. This will be discussed in detail in Chapter 5 of this thesis.

Another important observation was the difficulty of identifying the main method of certain patterns due to their manner of design or operation. The visitor pattern which will be looked at in Chapter 4 is one such pattern. This is because its operations are spread over two algorithmic structures making it difficult to achieve integration.

#### Conclusion

In this phase, the process of identifying combinable artefacts was used to evolve the combinations of the template method pattern with three of the creational pattern in the GoF handbook. These creational patterns are the factory method, the prototype pattern and the builder pattern respectively. Through this process, it was possible to realise that the method of combination suggested was incompatible with those patterns whose code representation made it difficult to identify a primary or main method. An example of such a pattern is the visitor pattern. This was the first indication that there could exist a category of patterns which could not be combined with other patterns to result in generative compound patterns. This category of patterns would be discussed in detail in chapter 4 of this thesis.

### 3.4.3 Research Phase three (RP3)

This research Phase takes its bearing from the design cycle phase of the three-cycle view of design science research. Here, the identified potential pattern combinations are prototyped and evaluated against their more static counterparts. The steps involved in this phase are more iterative as they apply each individual artefact that is developed. The process is detailed below:

#### Problem awareness

The problem awareness stage of RC3 is unique in that attention moves away from identifying mechanisms for combination to development and evaluation of the possible generative artefacts. Having identified combinatory static patterns and filtering the forms of relationships they exhibit; this phase is concerned with the realisation of metrics that prove or disprove the efficacy of the developed patterns. The problems that needed to be solved in this phase included:

* Development of code constructs that implement the combinatory mechanism identified in RC2
* Identification of metric systems for effective analysis.
* Identification of platform and language dependencies that might affect the evaluation process

#### Suggestions

This phase of RC3 makes use of the assumptions derived from the RC1. These assumptions influenced the manner of combination and the constituent patterns which were chosen for integration to become compound structures. In this phase, generative artefacts are developed and suggestions on how new functionality comes about based on novel configurations of existing and or new elements are explored. This phase shortlists numerous attempts at pattern combination which are in turn borne out of “assumption 3”. These assumptions stem from observed trends in the literature which suggest particular “combinatory pairs”. The breadth of these is discussed in chapter 2 which discusses the state of the art as concerns the development of generative or compound design patterns.

Theory, on the implementation of design science research methodology, indicates that this phase encapsulates a period of iterative development which sees the translation of the outputs of both RC1 and RC2 into measurable artefacts.

In this stage, the results of the filtration process undergone in RC1 and RC2 are implemented. Recursive and iterative attempts to combine different pattern types based on the findings observed in literature are performed with each artefact broken down and refactored to get the best combinatory pair possible.

Chapters 4 and 5 detail the result of iterating over the suggestion, development and evaluation stages of the DSRM. The result is a number of artefacts which this thesis argues are generative design patterns. This is followed by the introduction and discussion of a category of patterns which cannot be combined with others to give a generative construct by virtue of their operating behaviour or structure. Through these chapters, it is proposed that these artefacts appropriately populate a possible Wilson’s generative design pattern catalogue and present implementations which are critiqued with the aid of metrics against similar instances of the cooperating patterns interacting.

#### Evaluation

The evaluation phases define a stage whereby the performance of a constructed artefact is assessed based on previously defined criteria. The criteria are selected in a manner that compares the measured performance values with those expected and helps to confirm the validity of the research hypothesis. It has been argued that the main difference between DSR and other natural science methodologies is obvious here as the phase exposes the epistemic fluidity if the methodology, which lies in contrast to the stricter interpretation of positivist stances. This is because, in more positivist approaches, the research endeavour ends with subsequent discussions shedding light on areas of possible research but here, the results of any evaluation mechanism are fed back into the model at the suggestion stage allowing for iterative development on what has already been discovered.

This is the case in RP3 where possible combination pairs are then compared with their more static counterparts. Here, taking the results of the evaluation procedure as the assumptions, the DSRM is restarted.

#### Conclusion

At the end of this research phase, identified knowledge has been thoroughly distilled and documented. This knowledge is then either fed back into the system to refine the developed artefacts or documented as one of the phase’s outputs and goes into the research’s findings. As concerns this study, the results of the comparisons between the generative artefacts and their static counterparts are prescribed as proof of the viability of the research’s outputs.

## Conclusion

This chapter introduced and discussed the choice of design science research as an appropriate research methodology for this study. It argues that this is because of inherent similarity between the intent of DSR and the objectives of this research. This similarity revolves around the need to generate measurable outputs which can be assessed to determine the validity of the study.

In achieving this, this chapter began by reviewing the research framework used in this research. From there, it introduced design science research, the methodology that was employed in executing this study. Here it identified the three-cycle view of design science research and related this to the three-phased approach of the methodical framework.

Thirdly, the process of implementing the methodology was looked at. Of significance are the definitions of the different steps involved namely:

The problem awareness, which takes input about the problem domain to formulate the specifics of a problem to be solved. The suggestion, which attempts to provide direction on how the problems should be solved. The development, which is the process of using the identified knowledge to generate a concrete output or solution and finally the evaluation, which assesses the efficacy of the developed solution and outputs them in the conclusions of the process.

The following chapter, Chapter 4 goes on to present the developed artefacts.

# Chapter 4

FINDINGS

## Introduction

This chapter, discusses the core artefacts developed as a result of the design science research method applied to this study. It presents as outputs of the design science research methodology, a number of artefacts that have resulted from the process and are recognised as generative design patterns due to the manner in which they have been combined. This combination involves a merger of the main or primary methods of each design pattern in order to achieve some advantageous behaviour.

Having identified the generative constructs, this chapter will present each one using Wilson’s generative pattern framework. The framework will discuss the problem the pattern solves, the pattern’s intent, analyse its structure, present scenarios that show how the pattern can the employed and discuss emergent behaviour that results from the combination operation.

With the combinatory artefacts discussed, the chapter would introduce the subject of non-combinatory patterns. These refer to patterns that cannot form combined structures with other design patterns. To exemplify this category of patterns, the visitor pattern would be looked at.

It will be shown that a combination of the visitor pattern and any of the other patterns in the GoF handbook cannot be considered a generative or compound construct.

In conclusion, the chapter will summarise the main discoveries of the study, detailing the two categories of patterns, generative and non-combinatory.

## Generative patterns

Generative patterns have been described as design patterns that are themselves the result of a combination of two or more classical design patterns in a manner that results in advantageous behaviour which might not have been previously available to the patterns if they were working independently or co-operatively within a system.

### 4.2.1 Pattern Combinations

The first combinatory patterns to be identified are those formed from a combination of the “Template Method” with the creational patterns of the GoF’s 23 design patterns. The “Template Method” is a behavioural pattern which subscribes to the idea that any invariant components of a system or code’s behaviour should be implemented only once, and any variant parts should be deferred to the subclasses for implementation (Kanjilal, 2017).

Thus, the “Template Method” pattern manages the bigger picture of task semantics and refines the implementation of both the sequence of methods and the details of their selection (Kanjilal, 2017). This is because anytime the client invokes the methods of the template pattern’s external interface, the pattern in turn calls its defined abstract methods (its internal interface) as required to invoke the algorithm.

As a result, execution of the abstract methods is completely reliant or controlled by the template method while the implementation of the algorithm is left to the subclasses. This is the reason for the pattern’s main feature of enhanced expressive power, decoupling and greater degree of freedom.

Furthermore, the template method's abstract class could also define hook methods that may be overridden by subclasses, further extending the pattern’s extensibility.

This characteristic of the template method could be exploited by other patterns especially those responsible for object creation. Thereby allowing the system to hide the methods responsible for the actual creation process and as a result, afford the resultant construct more security and significant decoupling. This is the case with the first three patterns developed.

To show how this combination is implemented, the study presents a catalogue style definition of the developed generative artefacts. This definition would closely follow the format given by Wilson (2008) in his work on “a framework for generative design patterns” with a few organisational modifications to enhance clarity. The format will start with a look at the primary pattern followed by a secondary or related pattern in the relationship and then, a discussion on the generative artefact formed.

#### 4.2.1.1 Templated Factory as a generative design pattern

The first generative combination of patterns to be identified during the course of this research is that of the combination of both the factory method and the template method. This combination is one that combines both the behavioural and the creational characteristics of the two constituent patterns by allowing behavioural activity to be implemented by the subclasses of a creational pattern’s abstract class. The constituent patterns are unique in the way they operate in that they target characteristics of the subclasses of a particular base class. This common behaviour in the manner the patterns operate lends itself to fluid combination and reinforces Alur, Crupi & Malks (2003) observation that the factory method is to object creation as the template method is to algorithm implementation. The focus on the attributes of the subclasses is better understood when one considers the definition of both patterns.

##### Factory Method

The factory method as defined by the GoF handbook describes an interface for the creation of objects but permits its derived classes to decide which class to instantiate per circumstance (Gamma et al, 1995). This in practice means a superclass postulates all typical or common behaviour and, in this case, uses virtual "placeholders" to define the steps necessary to realize object creation. These steps are then delegated to the class’s subclasses, which in turn make the decision to create based on the nature of information supplied or requested by the client.

This can be explained by the operation of libraries. A library uses its abstract classes to define and maintain the relationships between and amongst the objects it encapsulates. These relationships define the responsibilities of the library including the responsibility of object creation if necessary. Consequently, the library understands that it needs to create objects but not necessarily the type of object it should create. The information about the object type to be created by the library is dependent on the application making the create request of the library. This makes a system more customisable even if a little bit more complicated due to the creation of extra classes. A detailed catalogue definition of the factory method is given as a precursor to defining the generative artefact later in this chapter.

|  |  |  |
| --- | --- | --- |
| PRIMARY PATTERN | | |
| **NAME**: Factory Method | | |
| **CLASSIFICATION TYPE**: Creational - creational design patterns are those design patterns whose main focus is the provision of object creation mechanisms. They concern themselves with trying to create objects in a manner suitable to the context within which they are implemented. The basic process of object creation could result in either the development of any number of design problems or the introduction of added complexity to the design of a system and as such, creational design patterns solve this problem by controlling this process. | | |
| **SOLUTION TYPE**: Variant Management - Tichy (1997) observes that variant management patterns are largely dependent on the features of object-oriented programming. They factor out areas of common functionality and this allows for the uniform treatment of different but related objects. | | |
| ANALYSIS | | |
| **INTENT**: The “Factory Method” defines an interface for the creation of program objects but allows its subclasses determine which class to instantiate. Also referred to as a virtual constructor, the “Factory Method” permits a class to defer object instantiation to its derived classes by defining a common constructor interface across object families. As abstract methods, these are to be overridden by the appropriate subclasses. | | |
| **PROBLEM**: A framework needs to homogenise the architectural model for a variety of applications but permit each distinct application to define its own domain objects and provide for its own instantiation. Such a scenario requires the introduction of inversion of control containers to maintain the modularity of the system and manage the flow of control. This can be explained as follows:   * The system cannot anticipate the type of objects required of it beforehand. * A class requires its subclasses to specify the objects it creates. * It is necessary to localize the logic for instantiating a complex object. | | |
| **SOLUTION**: An increasingly prevalent characterisation of the factory method views it as the static method that returns an object of its class's type. But unlike a typical constructor, the actual object that is returned might be an instance of a derived class or in some cases, an existing object might be reused instead of a new object created. Also, unlike typical constructors, factory methods generally possess different and more descriptive names. Consequently, the program’s client is completely decoupled from the implementation details of the derived classes which makes polymorphic creation feasible. The advantage of a Factory Method lies in its ability to return the same instance or a subclass of an expected object type multiple times.  Recent iterations of the “Factory Method” has seen several advocates recommend that (as a matter of language design, or failing that, as a matter of style) all constructors should be made private or protected. They argue that it is of no concerns to the client whether a class manufactures a new object or recycles an old one, and so the processes involved in the creation of objects should be hidden from view.  Furthermore, there is a difference between requesting an object and creating one. Creation used to primarily be achieved with the aid of the “new” operator which is considered damaging to code performance with extensive use. This is because, the “new” operator creates program objects but fails to encapsulate object creation. Therefore, the “Factory Method” is used to enforces encapsulation and lets an object to be requested without inextricably coupling the request to the act of creation. | | |
| DESIGN | | |
| **STRUCTURE**: In recent times, the Factory Method is commonly regarded as standard practice for object creation even in situations where the instantiated object would never change, or when instantiation takes place in an operation that derived classes can effortlessly override. This shows the classic description of the Factory Method as given in contemporary design pattern catalogues. | | |
| Figure 4.1: The Factory Method (dofactory, 2017) | | |
| PARTICIPANTS   * AbstractFactory (Framework): This creates the product objects. It declares the factoryMethod (), which when called, returns a Product object. * ConcreteFactory (ApplicationOne and ApplicationTwo): Overrides the generating method for creating ProductOne and ProductTwo objects * Product: Defines the interface for the objects created by the factorymethod(). * ConcreteProduct (ProductOne and ProductTwo): Implements the Product interface and embodies the output of a particular concrete factory | | |
| IMPLEMENTATION | |  |
| class Program    {      static void Main()      {        Creator[] creators = new Creator[2];        creators[0] = new ConcreteCreatorA();        creators[1] = new ConcreteCreatorB();          // Iterate over factories and create products        foreach (Creator creator in creators)        {          Product product = creator.FactoryMethod();          Console.WriteLine("Created {0}",            product.GetType().Name);        }        Console.ReadKey();      }    } | | CLIENT |
| abstract class Product{} | | PRODUCT |
| class ConcreteProductA : Product{} | | CONCRETE PRODUCT |
| class ConcreteProductB : Product{} | | CONCRETE PRODUCT |
| abstract class Creator   {  public abstract Product FactoryMethod();   } | | FACTORY |
| class ConcreteCreatorA : Creator  {      public override Product FactoryMethod()       {        return new ConcreteProductA();       }  } | CONCRETE FACTORY | |
| class ConcreteCreatorB : Creator  {  public override Product FactoryMethod()      {        return new ConcreteProductB();      }  } | CONCRETE FACTORY | |

Table 4.1: Factory Method

##### Template Method

In this pattern, a class is created that defines the steps of a designed algorithm. Those steps that are considered invariant are implemented inside this base class while those that are known to be variant, are assigned default implementations in the classes that inherit their operations from this base class. As a result, at run-time, the individually defined variant steps are supplied by the concrete subclasses that derive from the base class. The GoF handbook defines this process as the template method and asserts that the pattern allows for the definition of the skeleton of an algorithm in a base class whose details are then deferred to its subclasses (Kanjilal, 2017).

The template method is classified as a behavioural pattern because it manages the bigger picture of task semantics and the details of their selection, implementation and sequence. The larger picture referenced here calls abstract and non-abstract methods for the task at hand (Pai & Xavier, 2016) and grants it complete control over any abstract and non-abstract methods defined within its scope and accounts for its expressive power and the degree of freedom it affords developers (Stevenson & Phillips, 2003).

The template method pattern is commonly employed in scenarios that require the use of polymorphic methods. This is due to the fact that in most cases, the existence of a polymorphic method is predicated on the use of big picture methods that are defined abstractly or in a manner whereby their particular operations need to be modified by subsequent subclasses (Bud, 2000).

Because the template method allows subtle variances in subclass behaviour to be implemented without changing the core structure of an algorithm, it is possible to adopt this feature into a system of subclasses where the decision on which object to instantiate is left to the subclass itself. This system is, in essence, a factory method.

|  |  |
| --- | --- |
| SECONDARY PATTERN | |
| **NAME**: Template Method | |
| **CLASSIFICATION TYPE**: Behavioural - behavioural design patterns are design patterns that identify and appreciate common communication patterns between objects within a pattern. This implies that behavioural design patterns are primarily concerned with how responsibilities are allotted amongst the objects of a system, or the process via which behaviour is encapsulated in an object and requests are delegated to it. By enforcing these concepts, the patterns increase flexibility and control of how this communication is executed. | |
| **SOLUTION TYPE**: Variant management - Like the factory method, the template method also provides variant management solution type. These patterns are largely dependent on the features of object-oriented programming as they factor out areas of common functionality and this allows for the uniform treatment of different but related objects (Tichy, 1997). | |
| ANALYSIS | |
| **INTENT**: The pattern defines the skeleton of an algorithm in an operation while conceding the implementation of some steps of that algorithm to the client’s subclasses. Template Method allows subclasses redefine certain parts of an algorithm without modifying the algorithm's core construction. This is achieved by declaring algorithm “placeholders” within a base class which can then be implemented or overridden by the derived classes. | |
| **PROBLEM**: Multiple different components possess numerous points of significant similarity but do not demonstrate the use of a common interface or implementation. In the event that a change or modifications common to all components becomes necessary, duplicate effort will be expended. This multiplies with the number of components that require modification and could give rise to errors. | |
| **SOLUTION**: In implementing the template method, the developer first has to decide which steps of an algorithm are invariant. This means differentiating between those actions that repeated or are the same across components and which are variant, customizable or maintain a different implementation dependent of specific requirements. After this, these invariant steps are implemented in an abstract base class, while the variant ones are either given a default implementation specific to them or no implementation whatsoever. The variant steps characterise "hooks" or "placeholders", that can or must be supplied by the component's client in a subclass. This implementation not only determines what the required steps of an algorithm are and the order in which these steps are called, but also enables the client to extend, customize or replace some steps at run time. The “Template Method” features prominently in the design of frameworks where each framework implements the invariant parts of a domain's architecture and defines "placeholders" for all required client customization options. In this manner, borrowing the analogy from the understanding of space, the framework becomes the "centre of the solar system" while the client customizations are simply the planets that comprise said system. This inverted control structure has been humorously termed "the Hollywood principle" or "don't call us, we'll call you". | |
| **HOW THE SOLUTION OCCURS**: The implementation of template method() follows the following sequence.   * Call stepOne() * Call stepTwo() * Call stepThree()   Where, stepTwo() is a "hook" method or a placeholder that is defined in the program’s abstract class and inherited by its subclasses. Large scale reuse infrastructures such as frameworks, make extensive use of the “Template Method” because it allows for all recyclable code to be defined in the framework's base classes and from which the framework’s clients, which are free to define customizations can access it as required. | |
| DESIGN | |
| STRUCTURE  \\neptune\res_data\u1179154\Downloads\templatemethod.png  Figure 4.2: The Template Method (dofactory, 2017) | |
| **PARTICIPANTS**   * **AbstractClass (FramworkClass):** Defines abstract primitive operations that the concrete subclasses override to implement the steps of an algorithm. This class holds a template method that defines the skeleton of the program’s algorithm. This template method is responsible for calling both the defined primitive operations and other methods defined in the base class of the object hierarchy. It could also contain hook methods. These methods espouse a default implementation that could be overridden by some classes. The main difference between hook methods and the concrete methods defined in the concrete class is that hook methods are purpos4efully created to be overridden while the concrete methods are not. * **ConcreteClass (ApplicationClassOne & Two):** implements the primitive operations to carry out subclass-specific steps of the algorithm. | |
| IMPLEMENTATION | |
| class Program    {      static void Main()      {        AbstractClass aA = new ConcreteClassA();        aA.TemplateMethod();          AbstractClass aB = new ConcreteClassB();        aB.TemplateMethod();        // Wait for user        Console.ReadKey();      }    } | CLIENT |
| abstract class AbstractClass   {     public abstract void stepOne();     public abstract void stepTwo();     public void TemplateMethod()     {       stepOne();       stepTwo();       Console.WriteLine("");      }  } | ABSTRACT CLASS |
| class ConcreteClassA : AbstractClass   {      public override void stepOne()      {        Console.WriteLine("ConcreteClassA.stepOne()");      }      public override void stepTwo()      {        Console.WriteLine("ConcreteClassA.stepTwo()");      }   } | CONCRETE CLASS |
| class ConcreteClassB : AbstractClass   {     public override void stepOne()      {        Console.WriteLine("ConcreteClassB. stepOne()");      }      public override void stepTwo()      {        Console.WriteLine("ConcreteClassB.stepTwo()");      }    }  } | CONCRETE CLASS |
| **RELATIONSHIP TYPE:** Combines - Research reveals that combination as concerns these two patters is highly dependent on context. This is because there are two points for insertion in the implementation of the factory method depending on the context and the design parameters of the system being built. In the first instance, the abstract product class could be combined or merged with the abstract class of the template method. This results in a superclass that defines not only the product but also its behaviour. In the second instance, the abstract factory class merges with the abstract class of the template method to create a super factory that determines the product class that is returned per query and implements any common behaviour of the factory classes. | |

Table 4.2: Template Method

##### Templated Factory

|  |  |
| --- | --- |
| **NAME**: Templated Factory | |
| ANALYSIS | |
| **INTENT**: The templated factory pattern is designed to hide the process of object creation from external viewers including the client class. Its main contribution is the extension of the powers of object creation. This means it permits the bundling together of any attendant processes that might be needed to support the creation process and exposes these via a single method. | |
| **PROBLEM**: One main problem has to do with the fact that client applications have to know or have an understanding of all the variant methods defined by a superclass. This becomes a problem in scenarios where there exist numerous support methods bundled up with object creation or where it is important to hide these methods. This results in multiple method calls by clients to the abstract factory class which negatively impacts resource overhead. | |
| SOLUTION | |
| **HOW THE SOLUTION COMES ABOUT**: The creation of enemy types is a perfect scenario for the factory method as “enemy type” can be spawned based on other variables which have nothing to do with the mechanics of spawning or the behaviour of the enemy objects after they are spawned.  The template method allows for the developer to model the behaviour of the spawned objects separate from the process of instantiation.  Overlaying both systems on each other would see a system where behaviour and instantiation are separated by the signature of methods that launch them. | |
| **PARTICIPANTS**: The classes involved in the conjoined process are as follows:   * **AbstractFactory:** This class defines the factory method that is the core of this patterns creationist abilities. It is the base class for the enemy object creation * **ConcreteFactory:** As in the factory pattern, these classes override the factory method allowing for the creation of both concreteProductOne and concreteProductTwo’s objects. This means this is the class responsible for the creation of a particular enemy object * **Product:** It is the base enemy object. This is where the combination occurs and is where the construct differs from that of the two patterns interacting. This class defines the interface for the objects created by the factory method as well as introduced the template method which serves as a singular call for all methods that are attendant to the created products. Here, the behaviour of the objects is defined, and the template algorithm is implemented. Allowing for the instantiated objects to perform common actions with differing results. In the case of the example provided, this is the “shoot()” method. * **Concrete Product 1 & Concrete Product 2:** Implements the Product interface. These are the actual enemy object that has been instantiated. | |
| **SCENARIO 1**  Here we consider a game environment where some user battles enemies. The game has two enemy types, the human enemy which does minor damage and the robot enemy which does considerable damage. The game rules that require the enemies to pick up weapon items on the stage, reload said weapons and then they can then fire. | |
| DESIGN | |
| STRUCTURE  C:\Users\AlphaWolfe\Downloads\templatedFactory.png  Figure 4.3: Scenario implemented via Templated Factory | |
| IMPLEMENTATION | |
| class Program  {  static void Main(string[] args)  {  EnemyTypeFactory[] factory = new EnemyTypeFactory[2];  factory[0] = new humanEnemyFactory();  factory[1] = new robotEnemyFactory();    foreach (var enemytype in factory)  {  EnemyType enemy = enemytype.spawnEnemyType();  Console.WriteLine("spawned {0}", enemy.GetType().Name);  enemy.shoots();  }  Console.ReadKey();  }  } | The Client |
| abstract class EnemyTypeFactory  {  public abstract EnemyType spawnEnemyType();  } | ABSTRACT FACTORY CLASS |
| class humanEnemyFactory : EnemyTypeFactory  {  public override EnemyType spawnEnemyType()  {  return new humanEnemy();  }  } | CONCRETE FACTORY A |
| class robotEnemyFactory : EnemyTypeFactory  {  public override EnemyType spawnEnemyType()  {  return new robotEnemy();  }  } | CONCRETE FACTORY B |
| abstract class EnemyType  {  public String name;  public String amtDamage;  public String getName()  {  return name;  }  public void setName(String newName)  {  name = newName;  }  public String getDamage()  {  return amtDamage;  }  public void setDamage(String newDamage)  {  amtDamage = newDamage;  }  public void shoots()  {  pickWeapon();  reloadWeapon();  Console.WriteLine(getName() + " shoots playercharacter and does " + getDamage());  }  public abstract void pickWeapon();  public abstract void reloadWeapon();  } | ABSTRACT PRODUCT CLASS (MODIFIED) |
| class humanEnemy : EnemyType  {  public humanEnemy()  {  setName("humanPatrol");  setDamage("15.0");  }  public override void pickWeapon()  {  Console.WriteLine("human enemy picked weapon");  }  public override void reloadWeapon()  {  Console.WriteLine("human enemy reloaded weapon");  }  } | CONCRETE PRODUCT 1 (MODIFIED) |
| class robotEnemy : EnemyType  {  public robotEnemy()  {  setName("DoomBot");  setDamage("25.0");  }  public override void pickWeapon()  {  Console.WriteLine("robot enemy picked weapon");  }  public override void reloadWeapon()  {  Console.WriteLine("robot enemy reloaded weapon");  }  } | CONCRETE PRODUCT 2 (MODIFIED) |
| **CONSEQUENCES**  Although the described construct returns positive results, there is an obvious violation of the single responsibility principle. This is because the new super class is responsible for not only delegating operations to the subclasses but is also responsible for defining the exact actions these subclasses would perform. These actions have been worked into the classes as attributes or variables in the example presented and it should be noted that this approach is not always feasible.  -**Advantages**: The pattern maintains the main advantages of the factory and template pattern. Allowing for customization hooks for the objects created which can easily replace the super object that creates them. Also, the deferment of object creation to the subclasses is maintained.  -**Disadvantages**: The resulting generative construct can only be implemented for scenarios that involve a family of similar objects. This implies objects that extend the same base class. | |
|  | |

Table 4.3: Templated Factory

#### 4.2.1.2 Prototyped Template as a generative pattern

Having realised the compound pattern templated factory, the research then attempted to build on this by identifying other patterns whose purpose could be extended. The natural progression was to try to extend to those patterns whose primary aim is creation. Such as extension could involve the addition of behavioural attributes to the created object at runtime. This line of thinking led to the development of the prototyped template pattern. This pattern is borne from the combination of the prototype pattern and the template method.

In defining this pattern, the constituent patterns would, first of all, be defined and then their combination would be presented as a generative construct with the aid of Wilson’s framework.

The template method has already been defined in Section 4.2.1.1 thus, a reference would be made to it but its definition will not be repeated in this section.

##### Prototype pattern

The prototype pattern as defined by Gamma et al’s (1995) GoF handbook allows one to stipulate what type of object is to be created with the aid of prototypical instances, which copy this prototype (Shalloway & Trott, 2010). A creational pattern, the pattern broadly concerns itself with object creation and could be used in scenarios where class instantiation can be time-consuming, complicated or otherwise expensive in terms of the resources required to achieve it observes (Shalloway & Trott, 2010). In these situations, instead of creating more instances of the class from scratch, the pattern allows for the creation of multiple copies of the object defined by the class and these, can then be modified as required.

It is especially important in scenarios where it could be disadvantageous to directly call an object's constructor. For there to be alternative avenues from instantiation, the ability to then clone a pre-existing object (a prototype) of the same class reduces resource costs. This is the case when there exist many subclasses that differ only in the kind of objects they create, and so requires the use of a prototype pattern which effectively reduces the number of subclasses accessed by cloning a prototype that is representative of all. In this manner, the prototype design pattern facilitates the reduction of the number of classes present in such a system.

Over time, prototype patterns have been absorbed into a number of object-oriented languages and are implemented with the “clone()” method in c# and the “IClonable” interface in Java. This means that for a “clone()” operation to be successful in java, classes that are required to support cloning must inherit from the “ICloneable” interface. This interface maintains a “clone()” method which can be overridden in its subclasses. It should be noted that the clone method could be implemented to varying degrees, this implementation could be deep which mandates that all objects are duplicated or shallow copy where only top-level objects are explicitly duplicated, and subsequent objects are referenced.

|  |  |
| --- | --- |
| **PRIMARY PATTERN** | |
| **NAME**: Prototype Pattern | |
| **CLASSIFICATION**: Behavioural Pattern: behavioural design patterns are design patterns that identify and appreciate shared communication patterns between objects within a pattern. This implies that behavioural design patterns are primarily concerned with how responsibilities are allotted amongst the objects of a system, or the process via which behaviour is encapsulated in an object and requests are delegated to it. In so doing, these patterns increase flexibility in carrying out this communication. | |
| **SOLUTION TYPE**: Control patterns concern themselves with the flow or control of program execution. This has to do with selecting the appropriate methods at the right time to achieve a stated objective or functionality. | |
| ANALYSIS | |
| **INTENT**: The prototype pattern is used in scenarios where the new operator is detrimental to proper management of resources. It specifies the type of objects to generate by making use of a prototypical instance from which other objects are cloned. This in effect hijacks a specific instance and forces it to act as an incubator for all possible instances that come after. | |
| **PROBLEM**: Application "hard wires" the class of object to create in each "new" expression. | |
| **SOLUTION**: This pattern works by implementing a prototype interface which creates a clone of the current object. This reduces the need for writing repetitive code, especially the code responsible for creating and then copying the object’s state to the new object. The prototype design pattern, therefore, handles the complex task of creating either deep or shallow copies of the primary object. | |
| **HOW THE SOLUTION OCCURS:** The prototype pattern resolves the problems associated with instantiating program objects via the “new” key word. It does this by:   * Define an abstract class that specifies a pure virtual “clone” method and maintains a dictionary of all “cloneable” concrete subclasses. Any class that needs the capabilities of a polymorphic constructor must, therefore, derive from the abstract base class. * Register its prototypical instance and implement the “clone()” method.   In the client code, this eliminates the need for multiple new operators and instead, every new object simply calls the “clone()” method on the abstract base class and supplies any necessary information which ties the object to a particular derived concrete class as required. | |
| DESIGN | |
| STRUCTURE  http://1.bp.blogspot.com/-UZ_vU0ghJhM/VofVwmk77qI/AAAAAAAACJg/ju_Bs-4Sj-Q/s400/22.png  Figure 4.4: The Prototype Pattern (dofactory, 2017) | |
| **PARTICIPANTS**   * Prototype: This creates the product objects. Defines an interface for cloning itself. * ConcretePrototype (ImageOne and ImageTwo): implements the operation for cloning itself * Client: asks a prototype to clone itself. | |
| IMPLEMENTATION | |
| class Program    {      static void Main()      {        // Create two instances and clone each        ConcretePrototype1 p1 = new ConcretePrototype1("I");        ConcretePrototype1 c1 = (ConcretePrototype1)p1.Clone();        Console.WriteLine("Cloned: {0}", c1.Id);          ConcretePrototype2 p2 = new ConcretePrototype2("II");        ConcretePrototype2 c2 = (ConcretePrototype2)p2.Clone();        Console.WriteLine("Cloned: {0}", c2.Id);          // Wait for user        Console.ReadKey();      }    } | CLIENT |
| abstract class Prototype    {      private string \_id;      public Prototype(string id)      {        this.\_id = id;      }      public string Id      {        get { return \_id; }      }      public abstract Prototype Clone();    } | PROTOTYPE |
| class ConcretePrototype1 : Prototype    {      public ConcretePrototype1(string id)        : base(id)      {      }      // Returns a shallow copy      public override Prototype Clone()      {        return (Prototype)this.MemberwiseClone();      }    } | CONCRETE PROTOTYPE |
| class ConcretePrototype2 : Prototype    {      public ConcretePrototype2(string id)        : base(id)      {      }      // Returns a shallow copy      public override Prototype Clone()      {        return (Prototype)this.MemberwiseClone();      }    } | CONCRETE PROTOTYPE |

Table 4.4: Prototype Pattern

##### Prototyped Template

|  |  |
| --- | --- |
| **NAME**: Prototyped Template | |
| ANALYSIS | |
| **INTENT:** The prototyped pattern permits the appending of behavioural attributes to similar object instances at runtime. This reduces the resource-intensive nature of object creation especially when those objects have similar but different operations defined within them. | |
| **PROBLEM**: Similar to the templated factory, the prototyped template is designed to reduce the need for the client to be aware of all methods defined by the abstract class. As this invariably leads to duplication of code and the need to effect modification in numerous places. Using the template as a gateway method allows the developer to only expose innocuous methods to the client affording the system more security overall. | |
| **SOLUTION:** The combination both these patterns makes for a rather complex if not convoluted system. The template method here presents a singular access point via which the client can access the cloning process be that a deep clone or a shallow clone. The pattern is extensible in that any additional functionality need only be implemented as methods within the body of code. | |
| **SCENARIO 1**  For this scenario, we consider a piece of code that requires that a system runs through the test sequence for vehicles. The tests include starting the vehicles, operating the vehicles and stopping them at the end of the test run. For this system, we are required to test two plane types, Jet and Boeing and two car brands, Celica and Corolla. The caveat is that the system doesn’t have enough resources for us to instantiate all vehicle brands and so we are presented with only the base plane and car classes respectively. | |
| DESIGN | |
| STRUCTURE  C:\Users\AlphaWolfe\Downloads\prototyped template (1).png  Figure 4.5: Scenario implemented via the Prototyped template | |
| **PARTICIPANTS**  **AbstractClass**: --this class maintains the template method that will determine the behaviour exhibited by the system but also declares an interface for cloning itself to allow multiple instances of itself be generated  **ConcreteClassA & B**: --These inherit from the base abstract class and therefore supply the methods to be implemented the abstract methods declared in the base class. This abstract method includes the clone method which allows the created object to clone itself.  **Client**: --Asks a prototype to clone itself in order to create a new program object. | |
| IMPLEMENTATION | |
| abstract class vehicleBehaviour  {  public string Name { get; set; }  public abstract vehicleBehaviour clone();  public void act(vehicleBehaviour productName)  {  start();  move();  stop();  }  public void start()  {  Console.WriteLine("{0} is started", Name);  }  public void stop()  {  Console.WriteLine("{0} just stopped", Name);  }  public abstract void move();  public abstract String GetDetails();  } | ABSTRACT CLASS (MODIFIED) |
| class Plane : vehicleBehaviour  {  public override vehicleBehaviour clone()  {  return (vehicleBehaviour)this.MemberwiseClone();  }  public override string GetDetails()  {  return string.Format("this plane is a {0} ", Name);  }  public override void move()  {  Console.WriteLine("{0} is flying", Name);  }  } | CONCRETECLASSA |
| class Car : vehicleBehaviour  {  public override vehicleBehaviour clone()  {  return (vehicleBehaviour)this.MemberwiseClone();  }  public override string GetDetails()  {  return string.Format("this car is a {0} ", Name);  }  public override void move()  {  Console.WriteLine("{0} is being driven", Name);  }  } | CONCRETECLASSB |
| class Program  {  static void Main(string[] args)  {  vehicleBehaviour planetype1 = new Plane();  planetype1.Name = "jet";  Console.WriteLine(planetype1.GetDetails());  planetype1.act(planetype1);  vehicleBehaviour planetype2 = (planetype1).clone();  planetype2.Name = "boeing";  Console.WriteLine(planetype2.GetDetails());  planetype2.act(planetype2);  vehicleBehaviour carType1 = new Car();  carType1.Name = "celica";  Console.WriteLine(carType1.GetDetails());  carType1.act(carType1);  vehicleBehaviour carType2 = (carType1).clone();  carType2.Name = "corolla";  Console.WriteLine(carType2.GetDetails());  carType2.act(carType2);  Console.ReadKey();  }  } | CLIENT |
| **CONSEQUENCES**  It can be observed that although the “Prototyped Template” like the “Templated Factory” allows for the introduction of variable behaviour to the instantiated classes, there is also a violation of the single responsibility principle. This violation is in the form of allowing a class to determine both the manner of instantiation and the behaviours it defines through its methods. Consequently, the base class is more aware of its subclasses than it needs to be.  -**Advantages**: The pattern, in this case, maintains the advantages of the template method but allows one to create multiple instances of the object that that will make use of the algorithm defined in the template. This is achieved without having to rely on more or other resources asides those already used in spawning the primary object or instance.  -**Disadvantages:** Like the templated factory, this pattern can only be used when the required objects exhibit the same behaviours or extend the same base class. | |

Table 4.5: Prototyped Template

#### 4.2.1.3 Templated Builder Pattern as a generative Pattern

The third artefact developed is the template builder pattern. The pattern stems from the combination of the builder pattern with the template method. This section would look at the constituent patterns in detail. The template pattern would be omitted as it has been defined in section 4.2.1.1.

##### Builder Pattern

Unlike the factory patterns (abstract factory and the factory method) whom are primarily concerned with enabling polymorphism, the main objective of the builder pattern is to resolve the issue of the telescoping constructor anti-pattern that occurs when an increase in the number of object constructor parameter combinations leads to an exponential list of objects. This means that the more complex an application, the greater the complexity of classes and the number objects used. Complex objects are made of parts produced by other objects that need special care when being built observes (oodesign.com, 2017). An application might need a mechanism for building complex objects that is independent of the ones that make up the object. If this is the problem, then the builder pattern suggests a more appropriate solution than attempting to compose the objects implicitly at run-time.

Also, as relates to both the factory method and abstract factory, there is a tendency for a substantial amount of arguments to pass from the client program to the Factory class. These arguments are very error prone when parsed by humans and since the type of arguments could be the same as those from client side, it becomes difficult to maintain the order of the argument.

Furthermore, some of the parameters might be optional but in the case of the factory pattern, we are forced to send all the parameters, optional or otherwise and would need to get creative to force these optional parameters to result into Null. Consequently, if the object to be built is heavy and its creation is complex, then all that complexity will be part of the factory classes and potentially could be confusing.

|  |  |
| --- | --- |
| PRIMARY PATTERN | |
| **NAME**: Builder Pattern | |
| **CLASSIFICATION TYPE**: Creational Patterns - creational design patterns are those design patterns whose main focus is the provision of object creation mechanisms. They concern themselves with the creation of objects in a fashion suitable to the context within which they are implemented. The basic processes that define the creation of a program object could result in any number of design problems or a rise in the complexity of a system. As such, creational design patterns resolve this issue by invariably controlling this creation process. | |
| **SOLUTION TYPE**: Variant Management - These patterns are largely dependent on the features of object-oriented programming as they factor out areas of common functionality and this allows for the uniform treatment of different but related objects (Tichy, 1997) | |
| ANALYSIS | |
| **INTENT**: The builder pattern typically defines an interface for creating objects but permits these objects to be determined by the subclasses. This process, therefore, separates object representation from object construction. Especially as concerns complex objects that are an aggregate of other individual objects. As a result, the same construction procedure can be used to create multiple but different object representations. This means the builder pattern:   * Parses a complex representation and thus could be used to create one of several targets. * Refers to the newly created object through a common interface   It is important to note that the primary intention of the builder pattern is to address issues relating to the telescoping constructor anti-pattern. This is a scenario that results from the existence of an exponential list of constructors due to an increase in the number of constructor parameter combination. | |
| **PROBLEM**: An application is tasked with creating the individual elements of a complex aggregate. The specification for the aggregate resides in a secondary location but one of many the representations needs to be built in primary storage. | |
| **SOLUTION**: The builder pattern shields a “client” object from the details of a “product” object’s representation. This is because, the logic of the construction process is removed from the concrete steps involved in crafting the complex object and as a result, the process of constructing a product object can be replicated for numerous but varying objects. This happens because the client only has to specify an object’s type and content in order to construct it.  In addressing the issues of telescoping, the client, which could either be a separate object or the program’s client class calls the main() method of the pattern and initiates both the Builder and Director classes. The Director class’s constructor accepts a “Builder” object as a parameter from the Client and maintains responsibility for calling the appropriate methods of the Builder class.  In everyday practice, it is usually the case that the Builder class is defined as abstract as this provides the Client class with an interface for all concrete Builders it is aware of. This makes it possible to vary the type of complex objects created by defining the object’s structure and reusing the construction logic.  This enforces the decoupling as the Client needs only know what new types exist while the director concerns itself with the particulars of which Builder method to call and when. | |
| DESIGN | |
| STRUCTURE  C:\Users\u1179154\Downloads\Builder2.png  Figure 4.6: The Builder Pattern (dofactory, 2017) | |
| **PARTICIPANTS**   * **The Builder**: This class defines an abstract interface for the creation of the different parts of a Product object. * **The ConcreteBuilder**: This class implements the Builder interface. It describes and maintains the representations it creates while providing an interface for saving and retrieving the constructed product. * **The Director**: The director class uses the Builder interface to manufacture the complex product object. * **The Product**: This is the complex object that is being constructed. It refers to classes that define the constituent parts of the complex object created by the director. It is these individual product objects that will be assembled into the final complex product object. | |
| IMPLEMENTATION | |
| public static void Main()      {         Director = new Director();         Builder b1 = new ConcreteBuilder1();         Builder b2 = new ConcreteBuilder2();         director.Construct(b1);         Product p1 = b1.GetResult();         p1.Show();         director.Construct(b2);         Product p2 = b2.GetResult();  p2.Show();  // Wait for user  Console.ReadKey();  } | CLIENT |
| class Director  {      // Builder uses a complex series of steps    public void Construct(Builder builder)      {         builder.BuildPartA();         builder.BuildPartB();       }    } | DIRECTOR |
| abstract class Builder     {      public abstract void BuildPartA();      public abstract void BuildPartB();      public abstract Product GetResult();    } | BUILDER |
| class ConcreteBuilder1: Builder  {  private Product \_product = new Product ();    public override void BuildPartA()  {  \_product.Add("PartA");      }         public override void BuildPartB()       {         \_product.Add("PartB");       }         public override Product GetResult()       {          return \_product;       }  } | CONCRETE BUILDER1 |
| class ConcreteBuilder2: Builder     {  private Product \_product = new Product();    public override void BuildPartA()  {    \_product.Add("PartX");  }       public override void BuildPartB()       {         \_product.Add("PartY");       }       public override Product GetResult()       {         return \_product;       }  } | CONCRETE BUILDER2 |
| class Product    {       private List<string> \_parts = new List<string>();       public void Add (string part)  {  \_parts.Add (part);       }       public void Show ()       {         Console.WriteLine("\nProduct Parts -------");  foreach (string part in \_parts)           Console.WriteLine(part);       }  } | PRODUCT |

Table 4.6: Builder pattern

##### Templated Builder

|  |  |
| --- | --- |
| **NAME:** Templated Builder | |
| ANALYSIS | |
| **INTENT:** Manage the process of building complex objects while limiting direct awareness of the process itself. As a result, the client doesn’t need to know the type or content of the object. This is the major departure from the original builder pattern as there the client has to know the type of the object being created. | |
| **PROBLEM:** This pattern is useful in situations where there is a need to create a complex or composite object that is composed of objects of the same type. A primary observation is that the template builder is only applicable for objects of a homogeneous object type. | |
| **SOLUTION:** The solution comes from the definition of a template method within the abstract builder class. This template method abstracts away the builder methods and presents a singular method for the director to call. This affords the director some level of extensibility as there is no direct coupling between the two classes and any access is managed via the template method. | |
| **SCENARIO 1:** For this scenario, we consider a piece of code that manages the process via which vehicles are assembled in a vehicle factory. The factory operates a “PartsBuilder” system that constructs vehicles in a series of sequential steps. Choosing to streamline to manufacturing process, the factory intends to focus on vehicle type per season but still maintain the ability to build vehicles with differing components. Furthermore, since customers have access to the factory’s systems, it is critical that any and all client applications the factory’s system communicate with are kept in the dark about the inner workings of the factory’s build process. | |
| DESIGN | |
| STRUCTURE:  C:\Users\u1179154\Downloads\Untitled Diagram (1).png  Figure 4.7: template method combined with builder pattern | |
| **PARTICIPANTS**:  **AbstractPartsBuilder**: -- This class defines an abstract interface for the creation of the different parts of a Product object. The class also maintains a template method that functions like a gateway method to the build process. This template method is used to set the sequence of execution of the building of the different parts. For the systems current use, the build method is used as it hides the methods that build the different parts from the client.  **ConcreteBuilderA** **& B**: --These inherit from the base abstract builder and therefore supply the methods to be implemented the abstract methods declared in the base class. They describe and maintain the representations of the individual parts they create while providing an interface for saving and retrieving the constructed product.  Product: refers to the complex object that is being constructed. **ConcreteBuilders**: these construct a complex object’s (product) internal representation and determines process by which the product is assembled. This means the concrete classes typically include classes that define the constituent parts of a complex product and implement the interfaces responsible for the process of assembling the parts into the final result.  **Director**: builds an object using the Builder interface. | |
| IMPLEMENTATION | |
| class Program  {  static void Main(string[] args)  {  Director director = new Director();  PartsBuilder[] factories = new PartsBuilder[2];  factories[0] = new PartAFactory();  factories[1] = new PartBFactory();  foreach (var item in factories)  {  director.Make(item);  Vehicle V = item.GetResult();  V.Show();  }  Console.ReadKey();  }  } | CLIENT |
| abstract class PartsBuilder  {  public abstract void BuildPartA();  public abstract void BuildPartB();  public abstract Vehicle GetResult();  public void Build()  {  BuildPartA();  BuildPartB();  }  } | ABSTRACTBUILDER |
| class PartAFactory : PartsBuilder  {  private Vehicle \_product = new Vehicle();  public override void BuildPartA()  {  \_product.Add("PartAA");  }  public override void BuildPartB()  {  \_product.Add("PartAB");  }  public override Vehicle GetResult()  {  return \_product;  }  } | CONCRETEBUILDERA |
| class PartBFactory : PartsBuilder  {  private Vehicle \_product = new Vehicle();  public override void BuildPartA()  {  \_product.Add("PartBA");  }  public override void BuildPartB()  {  \_product.Add("PartBB");  }  public override Vehicle GetResult()  {  return \_product;  }  } | CONCRETEBUILDERB |
| class Vehicle  {  private List<string> \_parts = new List<string>();  public void Add(string part)  {  \_parts.Add(part);  }  public void Show()  {  Console.WriteLine("\nProduct Parts -------");  foreach (string part in \_parts)  Console.WriteLine(part);  }  } | PRODUCT |
| class Director  {  public void Make(PartsBuilder partsFactory)  {  partsFactory.Build();  }  } | DIRECTOR |
| **CONSEQUENCES**  The use of the template method to encapsulate the builder methods allows the build process to be shielded from external viewers. This introduces a level of decoupling into the system while also affording the system a level of extensibility. This extensibility can be used in cases where other methods that are not part of the builder process need to be called to support the process.  Furthermore, this introduces a level of flexibility to the build process which can be overridden by the subclasses of the builder. This is because these subclasses determine exactly how to best implement the different build methods they inherit.  -**Advantages**:   * The pattern, in this case, maintains the advantages of the template method but allows one to create multiple instances of the object that that will make use of the algorithm defined in the template. This is achieved without having to rely on more or other resources asides those already used in spawning the primary object or instance. * It simplifies code maintenance because low-level changes are effectively decoupled from the larger framework as the interaction the director class’s object has is with the template method and not the build methods it encapsulates.   -**Disadvantages**:   * The decoupled nature of the application could adversely affect code readability, especially when attempting to understand the function of each component of the system. | |

Table 4.7: Templated Builder

## Non-Combinatory Patterns

### 4.3.1 Visitor as a non-combinatory pattern

The Visitor pattern uses a visitor class to modify the execution algorithm of a given element class. This pattern is a behavioural pattern because it allows for an algorithm to vary with the type of visitor that is supplied to it. This means that the visitor pattern affords a developer a mechanism via which the execution of an algorithm can be separated from the object structure on which it operates. For the visitor to work, element objects within the program have to declare an accept method that permits the visitor object to handle an operation on the element object.

|  |
| --- |
| NAME: Visitor Design Pattern |
| **CLASSIFICATION TYPE**  Behavioural Pattern: These design patterns identify mutual communication patterns between a program’s objects and thus are primarily responsible for the well-organized and effectual assignment of operational behaviour amongst them. As a result, this category of design pattern increases and improves code flexibility. |
| **SOLUTION TYPE**  Variant Management: Variant management patterns factor out any commonality amongst dissimilar but related program objects and uses this commonality as a basis for enforcing uniform treatment of the program’s objects. To achieve this, the variant management patterns are largely influenced by features found in object-oriented programming languages. |
| ANALYSIS |
| **INTENT**  The visitor pattern allows one to define operations that target the elements that constitute an object structure without changing the element’s classes. This means the visitor allows us to vary the execution algorithm of an element class. Also, this permits the definition of new operations without having to change the classes of the elements on which they would operate. |
| **PROBLEM**  One of the more common problems in software development revolves around how collections are handled. In most cases, collections are heterogeneous meaning they encapsulate objects of multiple data types. This poses a problem to developers when an operation is to be performed on these objects irrespective of their type. Before the visitor pattern, there was a need to query the type of each object so that it could be cast to the correct type before any operations could be performed on it. This is a time and resource intensive endeavour that complicated written code and negatively impacted software performance.  The issue is further compounded when multiple or unrelated operations have to be performed operations as it multiplies the casting process by as many operations as required |
| **SOLUTION**  The Visitor pattern’s primary purpose is the abstraction of functionality which can be applied to an aggregate hierarchy of "element" objects. The approach facilitates the construction of lightweight Element classes because the created elements are specialised and only focus on those functionalities that are specific to them. This is because all processing functionality is eliminated from the element’s list of responsibilities and is handled elsewhere in the code. Consequently, any new functionality can effortlessly be added to the original inheritance hierarchy by creating a new Visitor subclass. This is very much in line with the single responsibility principle of object-oriented programming  The “Visitor” implements "double dispatch" as opposed to the “single dispatch” that is routinely manifest in object-oriented messaging. This is because the executed operations of the visitor are dependent on three factors. These are the name of the request, the “type” of the Visitor object and the “type” of the receiver or acceptor object while single dispatch is wholly dependent just the first two mentioned.  Implementing the visitor pattern typically follows the following process:   * Create a “Visitor” class hierarchy that defines in its base class, virtual visit() methods for each “concreteElement” class in the aggregate hierarchy. These visit() methods should each accept a single argument that serves as a pointer or reference to the base “Element” class. * Each visit() method is overloaded in the appropriate derived subclass by allocating the "type query - cast" code. * An accept() method is defined in the base “Element” class. Mirroring the visit() methods, the accept() method also receives a single argument that references the visitor hierarchy. * With this, all “concreteElement” classes will inherit the accept() method and by calling the visit() method on the concrete derived instances of the Visitor hierarchy in the client, it passes it’s "this" pointer as its sole argument.   If and when a client requests for an operation to be performed, it creates an instance of the “Visitor” object and calls the “accept()” method defined in each element object before passing it the visitor object.  The accept() method allows the flow of control to locate the appropriate element subclass, so that upon invocation of the visit() method, the flow of control is vectored to the correct Visitor subclass. This is where the visitor pattern gets its double dispatch moniker as the accept() dispatch plus the visit() dispatch equals double dispatch.  The Visitor pattern makes adding new functionality to a program relatively easy as this is achieved by simply introducing new Visitor derived classes for each bit of functionality required. This requires some level of stability in the subclasses in the aggregate node hierarchy else synching the visitor subclasses could become tedious, if not difficult.  A recognised objection to the use of the “Visitor pattern” stems from that view that embodies a regression to functional decomposition. This implies the separation of an algorithm from its data structures and while this is a legitimate interpretation of the visitor’s operations, it has been argued that perhaps a better interpretation can be found in its goal of promoting non-traditional behaviour to full object status. |
| **PARTICIPANTS**  **Visitor** – This participant declares the “visit” methods which takes particular elements as arguments depending on class of each element. Typically, the name of the operation is the same and the operations and are differentiated by the method signature: The input object type decides which of the “visit” methods is called at run-time.  **ConcreteVisitor** – This class derives from the “visitor” class and implements its “visit” methods. Each type of visitor is responsible for different operations and as a result, implements only a part of the algorithm operating on the object structure. Consequently, to ensure that the entire algorithm is affected, all the visit methods declared in abstract visitor, must be implemented.  **Visitable** – This refers to the master class for those classes that declare the “accept” operation. The accept method is the entry point via which class objects can be "visited" by the visitor object. For collections, each object within the collection would have to implement this “accept” method in order for them to be affected by the visitor method.  **ConcreteVisitable** – These classes inherit from the “Visitable” interface or class as a result also define an “accept” operation. The visitor object is passed to this object using the accept() operation.  **ObjectStructure** – The object structure is that class which defines all the objects that can be visited by the visitor method. It could be a collection or a composite and offers a mechanism via which the visitor could iterate through all the elements it encapsulates. |
| DESIGN |
| Structure  \\neptune\res_data\u1179154\Downloads\visitorPattern.png  Figure 4.8: The visitor pattern (Guizzo & Vergilio, 2016) |
| IMPLEMENTATION |
| class MainApp  {  static void Main()  {  ObjectStructure o = new ObjectStructure();  o.Attach(new ConcreteElementA());  o.Attach(new ConcreteElementB());  ConcreteVisitor1 v1 = new ConcreteVisitor1();  ConcreteVisitor2 v2 = new ConcreteVisitor2();  o.Accept(v1);  o.Accept(v2);  Console.ReadKey();  }  } |
| abstract class Visitor  {  public abstract void VisitConcreteElementA(ConcreteElementA concreteElementA);  public abstract void VisitConcreteElementB(ConcreteElementB concreteElementB);  } |
| class ConcreteVisitor1 : Visitor  {  public override void VisitConcreteElementA(ConcreteElementA concreteElementA)  {  Console.WriteLine("{0} visited by {1}", concreteElementA.GetType().Name, this.GetType().Name);  }  public override void VisitConcreteElementB(ConcreteElementB concreteElementB)  {  Console.WriteLine("{0} visited by {1}", concreteElementB.GetType().Name, this.GetType().Name);  }  } |
| class ConcreteVisitor2 : Visitor  {  public override void VisitConcreteElementA(ConcreteElementA concreteElementA)  {  Console.WriteLine("{0} visited by {1}", concreteElementA.GetType().Name, this.GetType().Name);  }  public override void VisitConcreteElementB(ConcreteElementB concreteElementB)  {  Console.WriteLine("{0} visited by {1}", concreteElementB.GetType().Name, this.GetType().Name);  }  } |
| abstract class Element  {  public abstract void Accept(Visitor visitor);  }  class ConcreteElementA : Element  {  public override void Accept(Visitor visitor)  {  visitor.VisitConcreteElementA(this);  }  public void OperationA()  {  }  } |
| class ConcreteElementB : Element  {  public override void Accept(Visitor visitor)  {  visitor.VisitConcreteElementB(this);  }  public void OperationB()  {  }  } |
| class ObjectStructure  {  private List<Element> \_elements = new List<Element>();  public void Attach(Element element)  {  \_elements.Add(element);  }  public void Detach(Element element)  {  \_elements.Remove(element);  }  public void Accept(Visitor visitor) |
| {  foreach (Element in \_elements)  {  element.Accept(visitor);  }  }  } |

Table 4.8: The Visitor Pattern

The Visitor pattern cannot be combined in a manner that lends itself to “generativity”, as defined within the context of this study. This is due to the way the process of “visitation” works and the problems the pattern is designed to solve.

The pattern is meant to address the issues of double dispatch. This is an extension of the single dispatch ability that is prevalent in most object-oriented programming languages. The single dispatch, commonly found in the virtual methods of object-oriented languages, is a form of “dynamic dispatch” wherein a method is selected at runtime based on a single characteristic such as “object type” as opposed to at compile time as is the case with non-virtual functions.

**Scenario**

In explaining this, we consider the following scenario:

A game with two vehicle and coin type assets requires that the different vehicle types gain points by colliding with particular coin types. The points are awarded based on the vehicle type vs coin type interaction, hence, a regular vehicle gains 100 points for colliding with a regular coin and 1000 for colliding with a gold coin, while the armoured vehicle variant gains 80 points for colliding with a regular coin and 800 for the gold coins.

In the case of single dispatch, we see that only the type of a particular object is chosen to determine the method that is executed. The code example that follows demonstrates this:

|  |
| --- |
| public class Coin  {  public virtual string GetCoinType()  {  return "Coin";  }  }  public class GoldCoin : Coin  {  public override string GetCoinType()  {  return "GoldCoin";  }  } |

The coin objects are instantiated as follows:

|  |
| --- |
| Coin = new GoldCoin();  Console.WriteLine(coin.GetShipType());  THE OUTPUT  GoldCoin |

|  |
| --- |
| public class Vehicle  {  public virtual void CollideWith(bronzeCoin coin)  {  Console.WriteLine("Vehicle gained a 100pts");  }  public virtual void CollideWith(goldCoin coin)  {  Console.WriteLine("Vehicle gained a 1000pts ");  }  };  public class ArmouredVehicle : Vehicle  {  public override void CollideWith(bronzeCoin coin)  {  Console.WriteLine("ArmouredVehicle gained 100pts");  }  public override void CollideWith(goldCoin coin)  {  Console.WriteLine("ArmouredVehicle gained 1000pts ");  }  }; |

In executing this code, the vehicle and coin objects will be created as follows:

|  |
| --- |
| Vehicle myVehicle = new Vehicle();  ArmouredVehicle myArmouredVehicle = new ArmouredVehicle();  bronzeCoin aBronzeCoin = new bronzeCoin();  goldCoin aGoldCoin = new goldCoin();  calling the following methods, would result in:  myVehicle.CollideWith(aBronzeCoin);  myVehicle.CollideWith(aGoldCoin);  myArmouredVehicle.CollideWith(aBronzeCoin);  myArmouredVehicle.CollideWith(aGoldCoin);  THE OUTPUT  Vehicle gained a 100pts  Vehicle gained a 1000pts  ArmouredVehicle gained 100pts  ArmouredVehicle gained 1000pts |

But if the following were executed:

|  |
| --- |
| Vehicle theArmouredVehicleRef = new ArmouredVehicle();  bronzeCoin theGoldCoinRef = new goldCoin();  theArmouredVehicleRef.CollideWith(theGoldCoinRef);  THE OUTPUT  ArmouredVehicle gained 100pts |

This result is very different from the expected “ArmouredVehicle gained 1000pts” and is because the method selected to execute the code is solely based on just “theArmouredVehicleRef and not both “theArmouredVehicleRef” and “theGoldCoinRef”.

This limitation in execution capability is what the visitor pattern is designed to address, and it does this by distributing hook methods into those classes to be visited allowing resources to be modified by its own methods. This has the semblance of pattern interaction discussed earlier as the output of the preceding or in this case, acceptor object is separate from that returned after modification by the visitor object.

This separation of concerns cannot be easily extended, and thus defeats the purpose of combination as defined within this thesis.

## Conclusion

This chapter has defined the developed generative compound artefacts with the aid of Wilson framework starting with the template factory which was revealed to result from a merger of the factory method and the template method. This pattern is revealed to abstract the methods responsible for object creation with the aid of a template method. This provides some level of invisibility for the pattern while reducing the degree of awareness a client must have to interact with all the functions defined by the pattern’s abstract class.

After this, the Prototyped template pattern was presented. This pattern allows for dynamic appending of additional methods to the clone objects created by the patterns clone method. The pattern is also shown to still maintain its primary function of reducing the overhead costs that accrue as a result of using the new operator.

Thirdly, the chapter defined the template builder. This generative pattern was shown to facilitate the management of the numerous methods involved in building a complex object. This also affords a level of security to the individual build methods, further enforcing the single responsibility principle.

Finally, the chapter introduced the idea of patterns whose combinations with other patterns could not be defined generatively. These patterns, such as the visitor pattern were shown to operate in a manner that makes combination difficult within the context of this research. These patterns fall under the category of non-combinatory patterns.

# Chapter 5

EVALUATION AND METRICS

## Introduction

The term evaluation typically connotes methods and processes via which judgements are made about the quality, adequacy and suitability of a subject matter in relation to a defined environment (Weiss, 1972). This is the case in Design Science Research (DSR) where the term relates to methods and processes via which judgments are made concerning the outputs of the design science research process, be they theory or artefacts. Hevner et al (2004) opine that the process of evaluation of the outputs is as vital to the design science research methodology as is the process of creation of the outputs. As such, they suggest that it is important for researchers to prove the utility, quality, and efficacy of a produced output using multiple equally rigorous evaluation methods. This opinion agrees with Vaishnavi and Kuechler (2004) who suggest that there exists an obligation to ensure that any developed outputs are analysed in a manner that explains their immediate use and any observed performance. They argue that this process makes it possible for any changes and possible improvements in the behaviour of the system, people, and or organization to be implemented effectively.

Consequently, a number of design science researchers choose to tightly couple the evaluation of the outputs they realise with the design process itself (Peffers, 2012; Baskerville, Kaul, & Storey, 2015). The main argument for this, stems from the observable impact iterative evaluation has on a designer’s thinking (Cleven, Gubler, & Hüner, 2009). This is especially true for scenarios that are characterised by the rapid “build” and “evaluate” cycles that in this case, constitute the design process itself.

Although it is largely agreed that evaluation is crucial to the effective execution of DSRM (Cronholm & Göbel, 2016), the manner of evaluation is a highly contested subject in both general information system research and in design science research (Cleven, Gubler and Huner, 2009). The contention revolves around the premises adhered to by two main schools of thought. One school, the ex-ante perspective, maintains that design science research outputs are evaluated before they are evolved or implemented while the other, the ex-post perspective, believes that the outputs that result from the DSR process should be evaluated in relation to a chosen system or technology after they are evolved or implemented (Klecun and Cornford, 2005).

This thesis agrees with the ex-post perspective because the developed artefacts only come into being because of the DSR process and can only be evaluated in relation to their non-generative counterparts. To that end, this chapter corresponds to the evaluation stage of the design science research methodology discussed in Chapter Three and it details the process via which the developed generative artefacts are evaluated to ensure that they address the research problems stated in Chapter One.

To do this, the chapter will firstly present an evaluation strategy, this strategy will look at the when, what and why as concerns the evaluation of the developed artefacts. In articulating this strategy, this research will employ the FRAMEWORK FOR THE EVALUATION IN DESIGN SCIENCE (FEDS). This tool will highlight the two dimensions explored in evaluating the developed generative constructs discussed in chapter four.

This process will see the chapter look at each evaluation dimension in turn. The first dimension will see the individual outputs generated by the research endeavour compared with their static representations. Static representations, in this case, refer to structures that result from instances of the constituent patterns working together but not generatively. The investigation will take the constructs, in turn, starting with the “templated factory” followed by the “prototyped template” and finally, the “templated builder”.

After this, the second half of the evaluation strategy’s dimensions will be discussed. Here, the focus of this section would be the empirical performance of the generative patterns versus that of their static counterparts. Consequently, the subject of metrics will be discussed, and a number of popular metrics will be defined and subsequently calculated for each artefact. This will make it possible to identify areas where the generative constructs fare better than their static versions and paint a more empirical picture of the performance of the generative patterns in relation to their static counterparts.

## Evaluation strategy

Section (5.2) indicates an awareness of the rigorously contested nature of the process of evaluation in DSR. This can be associated with the obvious lack of well-defined frameworks for evaluating a prescribed output by either evaluation perspective (Venable, Pries-Heje and Baskerville, 2016). Design science researchers are, as a result, left to adopt or invent evaluation strategies which suit their individual narratives provided they stem from either the ex-ante or ex-post perspectives. This has seen the application of wide-ranging evaluation tools ranging from positivist approaches as suggested by Hevner et al (2004) and Walls et al (1992) to more experimental approaches such as those suggested by Tichy (1998). The approach of this thesis aligns more with March and Smith’s (1995) approach. March and Smith (1995) define evaluation as either of the two primary activities in design science. These activities are “build” and “evaluate”. This means that the evaluation strategy not only looks at the criteria by which the realised artefact or artefacts would be assessed but also determines if the process of assessment of the developed artefact’s performance is in line with stated objectives. This latter part refers to the process of obtaining the artefact’s performance values in comparison to certain defined criteria values. Therefore, this approach goes beyond simply establishing that an artefact worked or didn’t work but also identifies how and why it works or does not.

For this thesis, formulating a criterion to be used in the evaluation strategy requires an investigation of the differences between the proposed generative artefacts and their static counterparts.

To recap, static counterparts, in this case, refer to scenarios where pattern interaction expounds a co-operative relationship rather than a combined one. Mathematically, this can be described as follows:

“Co-operative or static relationship:

Pattern A + Pattern B = Pattern A + Pattern B”

While generative artefacts refer to structures where pattern interaction expounds a combined or compound relationship that sees the integration of the main methods that characterise a classical design pattern. Mathematically, this can be depicted as:

“Generative relationship:

Pattern A \* Pattern B = Pattern AB”

Having discussed the evaluation strategy, section 5.3 would articulate this strategy with the aid of the FEDS tool.

## Framework for Evaluation in Design Science (FEDS)

The FEDS tool is a result of the works of John Venable, Jan Pries-Heje and Richard Baskerville. As a design science research evaluation tool, it asks the question of how to best approach the design of a robust evaluation strategy in DSR and thus, presents itself as a roadmap with which DSR researchers could make appropriate decisions concerning the choice of what activities to leverage in evaluating design science research outputs.

Venable et al (2014) maintain that the framework was created by relating the different extant evaluation methods to the overarching goal of DSR’s evaluation process. This approach positions the goals of DSR’s evaluation phase as varying objectives of evaluation as an activity while the methods via which the activity is performed are the means to achieving those activities.

The FEDS framework itself is made up of four major steps which serve to present a two-dimensional characterisation of DSR’s evaluation episodes.

These steps are meant to:

* clarify the goals of the evaluation
* select appropriate evaluation strategy or strategies
* Identify what properties to evaluate
* design the individual evaluation episodes

As concerns, this research project, explicating the goals of the research correlates with providing answers to the questions of what, why, how and for what purpose?

* What is evaluated? The evaluated artefacts are those presented in Chapter 4. They represent constructs borne from the combination of two stand-alone patterns in a manner that merges their functionality and results in advantageous emergent behaviour. This advantageous behaviour is observed in both the utility and the performance of these artefacts in relation to their more static counterparts. In respect of this, the key evaluands are firstly the utility possessed by the developed artefacts and secondly, their individual performance in relation to their static counterparts.
* Why it is evaluated? From DSR literature, it is possible to identify the reasons for rigorous evaluation within the purview of design science research. These reasons include:
  + Determining the degree to which a developed artefact achieves its expected environmental utility. This entails ascertaining the artefacts main purpose. As concerns this research, this implies demonstrating the utility of generative compound patterns as relates to their performance and dynamism.
  + The substantiation of design theory in terms of the quality of the knowledge outcomes (Baskerville et al, 2007; Kuechler & Vaishnavi, 2012). This implies showing evidence that the proposed theory leads to some developed artefact that will be useful for solving a problem or bringing about some improvement. The main idea for developing generative patterns is the combination of individual design patterns into generative constructs by merging their main methods. This in effect, merges the main function of the constituent patterns and results in the emergence of advantageous behaviour. This result is advantageous because it solves the problem of rigidity observed in classical design patterns.
  + Comparing the performance of the developed artefact with the performance of other established ones with a view to identifying whatever improvements the artefact provides. This is crucial in establishing the similarities amongst the developed generative design patterns. For example, the combination of the template method with the creational patterns will be shown to improve decoupling.
  + The identification of undesirable side effects which are unaccounted for in the design stage of DSR (Venable, 2006). The process of evaluation makes it possible to observe potential side effects of combining the individual patterns as this is critical to establishing or demonstrating their efficacy.
* How is it being evaluated? The manner of evaluation is primarily formative and as such, relies on a comparison of the utility and performance metrics returned from the operations of the individual generative artefacts versus their static counterparts.
* When is the evaluation done? The question of when to evaluate harkens back to the two perspectives introduced in Section 5.2 The ex-ante perspective refers to “predictive evaluation which is performed in order to estimate and evaluate the impact of future situations” (Stefanou, 2001). This implies that evaluation in this context precedes the formulation and development of models or artefacts. In DSR, the ex-ante approach informs on the decision to acquire or develop a technology or determines which of several competing technologies should be acquired or adopted.

The ex-post perspective, on the other hand, is a determination of the value of the implemented system on the basis of cost and other defined criteria (Stefanou, 2001). This implies that ex-post evaluation approach succeeds the selection, development and or adoption of a particular technology or artefact. The particular flavour of the ex-post evaluation approach is commonly derived from descriptive, interpretive (Stockdale & Standing, 2006) or critical analysis (Klecun & Cornford, 2005).

The primary distinction between these two perspectives arises from the timing of the defined evaluative events and seems to suggest that ex-post evaluations are innately summative while ex-ante and intermediate evaluations are always formative. However, ex-ante and ex-post refer only to timing. Venable et al (2014) observes that a “summative evaluation may be required on an ex-ante or intermediate basis” such as for the elicitation of continuation approval while ex-post evaluations could also have formative purposes.

Section (5.2) signals that this thesis aligns itself with the ex-post perspective and as such the core evaluation activities occur after a realisation of the generative artefacts.

* The relationship to research objectives? One of the primary aims of this research is to populate a generative design catalogue as this could help bring the concept of generative patterns to the forefront of pattern research. As such, the results of an evaluation of these artefacts could serve as proof of concept that shows that these artefacts meet Wilson’s criteria for consideration as generative patterns.

### Application of FEDS

As introduced in Chapter 3, FEDS allows the research endeavour to articulate the answers to the primary questions posed by the premise of evaluation within the context of design science research.

As concerns this research, the FEDS process offers a mechanism to justify the evaluation design decisions by creating a bridge between the identified aims of the research and the evaluation strategy employed. To this end, two crucial dimensions as determined by the literature review are identified. These are:

* The functional purpose of the evaluation.
* The paradigm of the evaluation (artificial or naturalistic).

These two dimensions form the basis of the presented FEDS framework.

#### 5.3.1.1 Dimension 1

This attempts to distinguish between formative and summative approaches to the evaluation process. As such, this dimension considers any identifiable differences between the functional purpose of the developed artefacts and their static counterparts. This dimension contributes to improving the outcomes under evaluation and determines if the functional purpose of the developed artefacts match expectations.

Consequently, to fully explore the functional purpose of the developed artefacts, the individual intents and purposes of the constituent patterns have to be compared with the purpose of the resultant generative artefact.

##### Templated Factory

This generative pattern is a result of the combination of both the factory method and the template method design patterns. The resulting artefact allows for the individual actions to be performed by the factories to be protected, thereby offering increased levels of security to the program.

|  |
| --- |
| Factory Method |
| **Intent**:  This design pattern defines an interface for creating objects, but lets the factory’s subclasses determine which classes to instantiate per time. This effectively defers the processes of object instantiation to subclasses. |
| **Purpose**: Variant Management |
|  |
| Template Method |
| **Intent**:  This pattern defines the skeleton of an algorithm in an operation, deferring some steps to client subclasses. This in effect permits the subclasses to redefine certain steps of an algorithm without changing the algorithm's structure. |
| **Purpose**: Variant Management |
|  |
| Templated Factory |
| **Intent**:  It defines the skeleton of an interface for creating objects while hiding the innate details of the process from prospective clients. |
| **Purpose**: Variant Management/ Decoupling |

Table 5.1: Templated Method functional evaluation

The purpose of the template factory generative pattern is twofold. On one hand, it matches with that of the constituent patterns and with the expectation especially as the introduction of the template method into the abstract class of the factory method doesn’t break the flexibility the factory method offers but reinforces it. This means the templated factory generative pattern emphasises the process of deferment of action to the factory’s subclasses especially as it forces any access to the factories to happen via the template method, and as a result improves decoupling. Furthermore, it makes it easier to streamline the behaviour of the factories especially in scenarios where there exist multiple calls to different operations. This effectively reduces the overhead in the time taken to execute the code.

On the other hand, it further disentangles the relationship or awareness clients have of the processes that lead to the creation of object since any access the algorithm is only possible via the template method that encompasses the individual factory methods.

**Scenario**: Here we consider a game environment where two (2) character objects are spawned for the player character to interact with. The characters are limited to a certain number of each per session and so the system has to maintain a count of how many of each character type is created.

Breaking this down into the participating components, it is possible to identify two primary functions. These are:

* Creation of character objects
* Logging each creation cycle for each object type

A typical static representation would see the use of both the factory method and the template method. Here, the factory method pattern would be tasked with handling the process of object creation while the template method pattern would concern itself with maintaining an awareness of the number of times the creation process for each product type is completed. Figure (5.1) depicts the class diagram for a static implementation of the scenario.

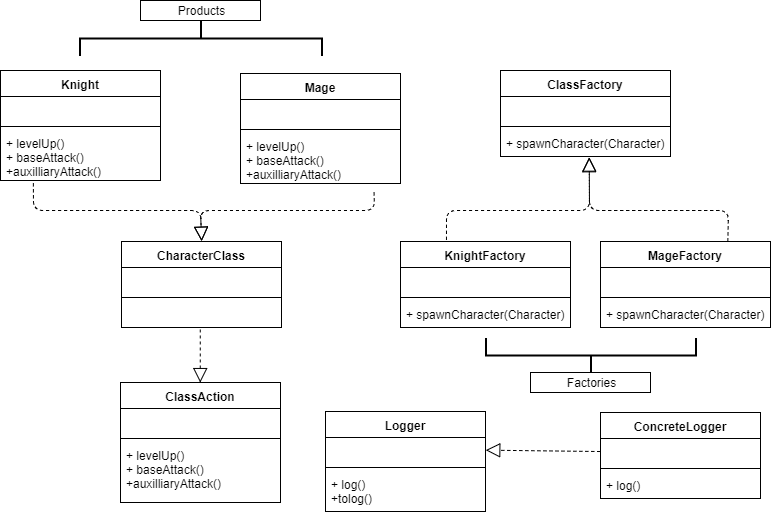


Figure 5.1: Static representation of the template and the factory method working together.

From the class diagram, the individual pattern components can be identified. These are as shown in fig (5.2) and fig (5.3).

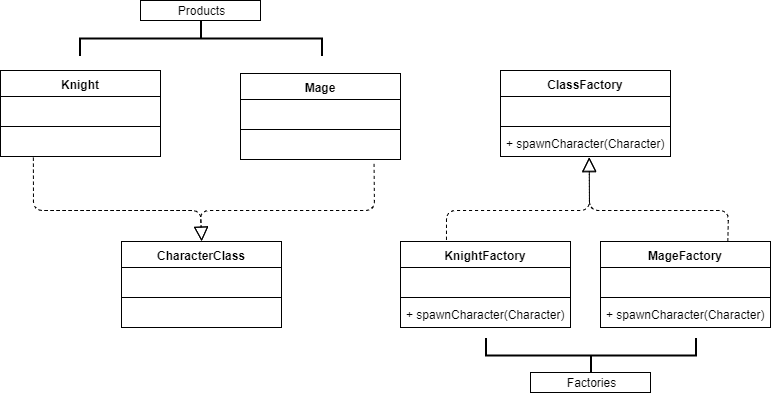


Figure 5.2: Class diagram for the factory component



Figure 5.3: Template method showing the logger component

Table (5.2) and table (5.3) show the code implementation of these patterns as relates to the scenario given.

|  |
| --- |
| abstract class Logger  {  public abstract void log();  public void tolog()  {  log();  }  }  class Concretelogger : Logger  {  public override void log()  {  Console.WriteLine("logged character creation”);  }  } |

Table 5.2: The logger component implemented with a template method pattern

|  |
| --- |
| abstract class ClassFactory  {  public abstract CharacterClass spawnCharacter(Logger obj);  }  class KnightFactory : ClassFactory  {  public override CharacterClass spawnCharacter(Logger obj)  {  obj.tolog();  return new Knight();  }  }  class MageFactory : ClassFactory  {  public override CharacterClass spawnCharacter(Logger obj)  {  obj.tolog();  return new Mage();  }  } |

Table 5.3: The character creation component implemented with the factory method pattern

From table (5.3), it is possible to observe the actions of the factory method “spawnCharacter()”. This operation is responsible for creating the individual instances of the product objects “knight” and “Mage”. It achieves this by permitting the “classFactory” class’s subclasses to override its implementation in their respective classes.

The log function is defined within a template method and is passed to the factory method as an argument allowing the client to instantiate it whenever a call is placed to the individual factories. This method encapsulates the log() function of the logger class and as such maintains its code structure regardless of any modifications that might need to be made to the component at a future date such as the addition of other functions to the logger class that are deemed immutable.

This allows the client class to be implemented in a manner that forces the logging function to be called anytime a particular object's factory is accessed as shown in table (5.4)

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  ClassFactory[] factory = new ClassFactory[2];  factory[0] = new knightFactory();  factory[1] = new MageFactory();  foreach (var item in factory)  {  Logger lgr = new Concretelogger();  CharacterClass Xclass = item.spawnCharacter(lgr);  Xclass.baseAttack();  Xclass.auxilliaryAttack();  Xclass.levelUp();  }  Console.ReadLine();  }  } |

Table 5.4: The client class for the static representation of template and factory methods working together.

The generative counterpart, on the other hand, differs from the static representation due to the fact that the logging function is defined as an operation within the “classFactory” class. This makes it possible to introduce a template method “generate()” into the class which subsumes the main factory method “spawnCharater()”. This effectively reduces the number of classes required to achieve the same result as the static representation while also reducing high-level code duplication. High-level code duplication refers to a scenario whereby multiple methods which are only differentiated by element comparison are present within the code (Deissenboeck, Hummel & Juergens, 2010). Because the log() method and the spawn method are shared by both factories, creating a template method eliminates the need to duplicate this operation.

The creation of a template method to handle both the “log()” function and the “spawnCharacter()” function enforces the open and closed principle. This means that any subsequent changes to code such as the addition of a new character class would only require the creation of a new subclass. Figure (5.4) depicts the class diagram for the template factory.

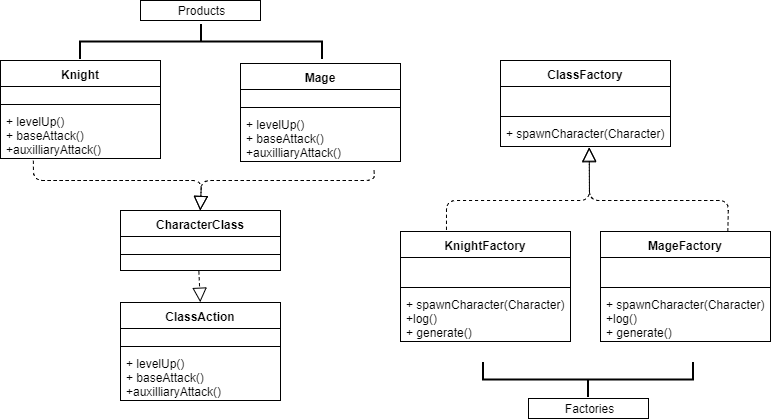


Figure 5.4: templated factory

Consequently, this eliminates the need for the instantiation of a logger class in each factory in the client class as shown in table (5.5) below.

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  ClassFactory[] factory = new ClassFactory[2];  factory[0] = new knightFactory();  factory[1] = new MageFactory();  foreach (var item in factory)  {  CharacterClass Xclass = item.generate();  Xclass.baseAttacl();  Xclass.auxilliaryAttack();  Xclass.levelUp();  }  Console.ReadLine();  }  } |

Table 5.5: Client class for the templated factory

##### Prototyped Template

An examination of the purpose of the template method seems to suggest that it could be used to encapsulate any sequence of methods and this led to tests that attempted to match the template method with all the other creational patterns.

The second pattern with which the template method pattern was combined with is the prototype pattern. As the name suggests, this compound artefact came about from the combination of the template method with the prototype method and attempts to add behavioural control to the creation of objects.

From the gang of four, it is understood that the template method helps define the structure of an algorithm (Shalloway and Trott, 2010) while the prototype method creates object instances without recourse to the limited resources used up during object creation (Freeman et al, 2004). It is, therefore, possible to bundle the clone method of the prototype inside a template method. This not only affords the resources of the object an added layer of security, due to the client having no direct knowledge of the clone method but instead interacts with it via the template method that encapsulates it, but also, affords the developer the opportunity to define processes involving multiple operations where cloned instances of an object are required.

|  |
| --- |
| Prototype Pattern |
| **Intent**:  Vilifies the use of the new operator but affords the opportunity to create new objects by cloning a prototypical instance of the main object which serves as a breeder of all future instances. |
| **Purpose**: State handling |
|  |
| Template Method |
| **Intent**:  This pattern defines the skeleton of an algorithm in an operation, deferring some steps to client subclasses. This in effect, permits the subclasses to redefine particular steps of the algorithm without altering the algorithm's structure. |
| **Purpose**: Variant Management |
|  |
| Prototyped Template |
| **Intent**:  Bundle a set of operations that require new instances of an object together. Reduce the awareness subclasses have of their parent especially when the subclasses implement different versions of the clone operation. |
| **Purpose**: Variant Management/ Decoupling |

Table 5.6: Prototyped Template Functional evaluation

The prototyped template also matches with expectations. This is because of the introduction of a template method to act as a gateway or proxy method to the clone operation which for the sake of simplicity and security should be hidden from the client class. This is buttressed by the fact that the process of cloning an object is rather complicated in a number of object-oriented programming languages especially when the object to be cloned maintains a circular reference to other classes within the program.

As explained in previous artefacts, the template method’s presence in the prototype’s abstract class affords it the use of the Hollywood principle. This grants the template method complete control of the object creation process thereby decoupling the subclasses from their parent.

**Scenario**: In order to evaluate the prototyped template, we consider a scenario in which a fictional online colour palette with which new colour objects can be created from pre-defined colours is held on a secure server. The process of defining new colour sets from the pre-existing ones requires that users clone or copy the existing sets and modify them as required. To access the server, the system is presented with a unique connection string which has to be submitted to the server every time the client or user makes a request to create a colour object.

Figure (5.5) and (5.6) depict the class diagrams for both the static and generative structures of this artefact.

Static (Template method + Prototype pattern)

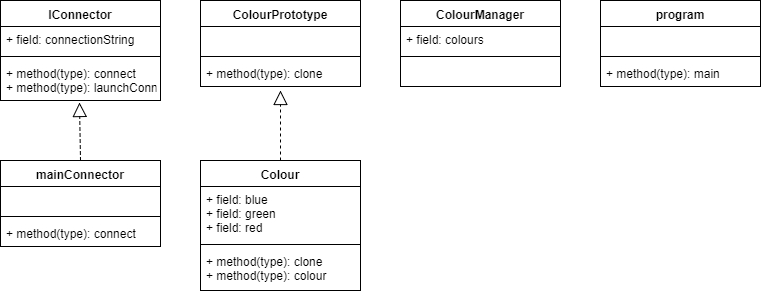


Figure 5.5: Static template method working with prototype pattern

Generative (Template method \* Prototype Pattern)

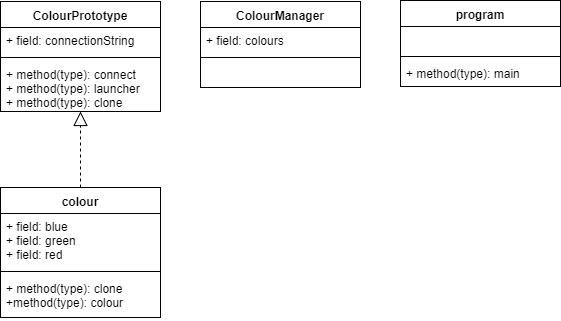


Figure 5.6: Generative artefact (Prototyped Template)

The static scenario, sees the rigid enforcement of the separation of concerns principle. As a result, the process of connecting to the online palette is implemented with a template method while the creation of colour objects is achieved with the prototype pattern. This is a very flexible design in that it allows each class to focus on the particular aspects of the process it is responsible for. Consequently, the client has to explicitly define a connector object outside of the other functions. This invariable uses up more resources as it necessitates the use of the “new” operation. Also, the different components have to explicitly reference each other in order to access the functionality they implement which invariably leads to more lines of code even though the system is more decoupled.

Furthermore, the system maintains a running connection for all calls to the palette as opposed to instantaneous calls which the scenario demands.

The client class for the static implementation is given in table (5.7)

|  |
| --- |
| Client class for the prototype and Template |
| class Program  {  static void Main(string[] args)  {  ColourPaletteManager colourPManager = new ColourPaletteManager();  IConnector palletteConnector = new mainConnector();  palletteConnector.LaunchConnection();  // Initialize palette with standard colours  colourPManager["red"] = new Colour(255, 0, 0);  colourPManager["leaf-green"] = new Colour(0, 255, 0);  colourPManager["sky-blue"] = new Colour(0, 0, 255);  // User adds personalized colours  colourPManager["Yellow"] = new Colour(255, 54, 0);  colourPManager["White"] = new Colour(255, 255, 255);  colourPManager["Orange"] = new Colour(211, 34, 20);  // User clones selected colours  Colour bloodRed = colourPManager["red"].Clone() as Colour;  Colour offWhite = colourPManager["White"].Clone() as Colour;  Colour Burgundy = colourPManager["Orange"].Clone() as Colour;  // Wait for user  Console.ReadKey();  }  } |
| Client class for the prototyped Template |
| class Program  {  static void Main(string[] args)  {  ColourPaletteManager colourPManager = new ColourPaletteManager();  // Initialize palette with standard colours  colourPManager["red"] = new Colour(255, 0, 0);  colourPManager["leaf-green"] = new Colour(0, 255, 0);  colourPManager["sky-blue"] = new Colour(0, 0, 255);  // User adds personalized colours  colourPManager ["Yellow"] = new Colour(255, 54, 0);  colourPManager ["White"] = new Colour(255, 255, 255);  colourPManager ["Orange "] = new Colour(211, 34, 20);  // User clones selected colours  Colour Crimson = colourPManager["red"].Launcher() as Colour;  Colour cream = colourPManager["White"].Launcher() as Colour;  Colour Burgundy = colourPManager["Orange"].Launcher() as Colour;  // Wait for user  Console.ReadKey();  }  } |

Table 5.7: Client code for both static and generative implementations of the template and prototype

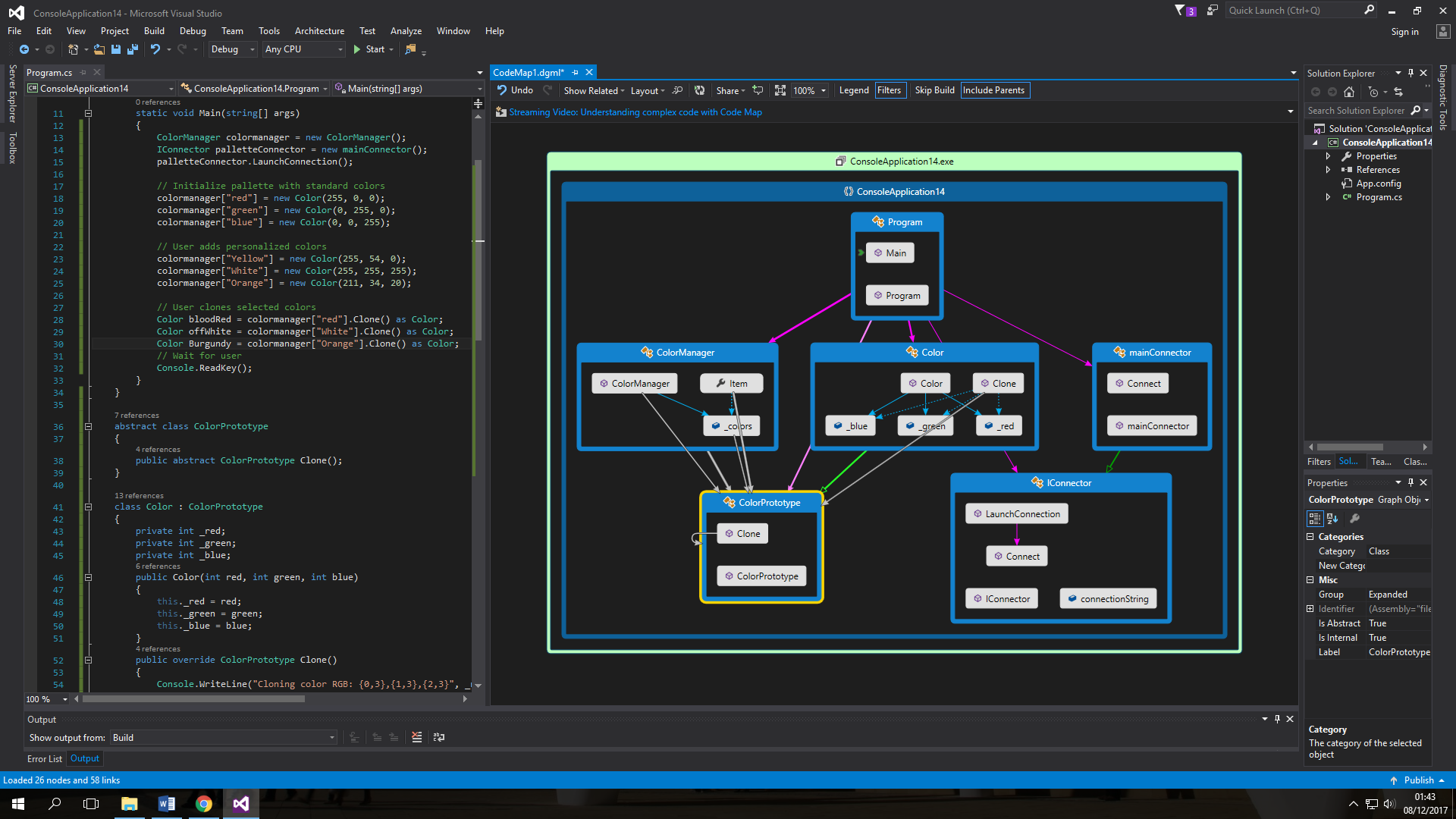


Figure 5.7: code map for the prototype and Template

In the generative scenario, the “colourPrototype” class is responsible for the connection to the online palette. This is achieved by defining the connect method inside the “colourPrototype” class. The implementing of a template method “launcher()” allows for all requires functions to be executed with one method call. This means that the system only accesses the online palette anytime the user makes a call to it without maintaining an always on instance.

This significantly violates the single responsibility principle in object oriented programming but enforces one of the requirements of the system as given in the scenario.

Also, the presence of the template method “launcher” means the clone process and any other sensitive operations can be hidden from the client by declaring such operations as protected. This significantly improves security within the system as exposure to the client is then only possible via the template method.

When considered as a whole, the generative artefact significantly reduced the number of classes required to achieve the same result. The code map for the generative prototyped template is given in figure (5.8).

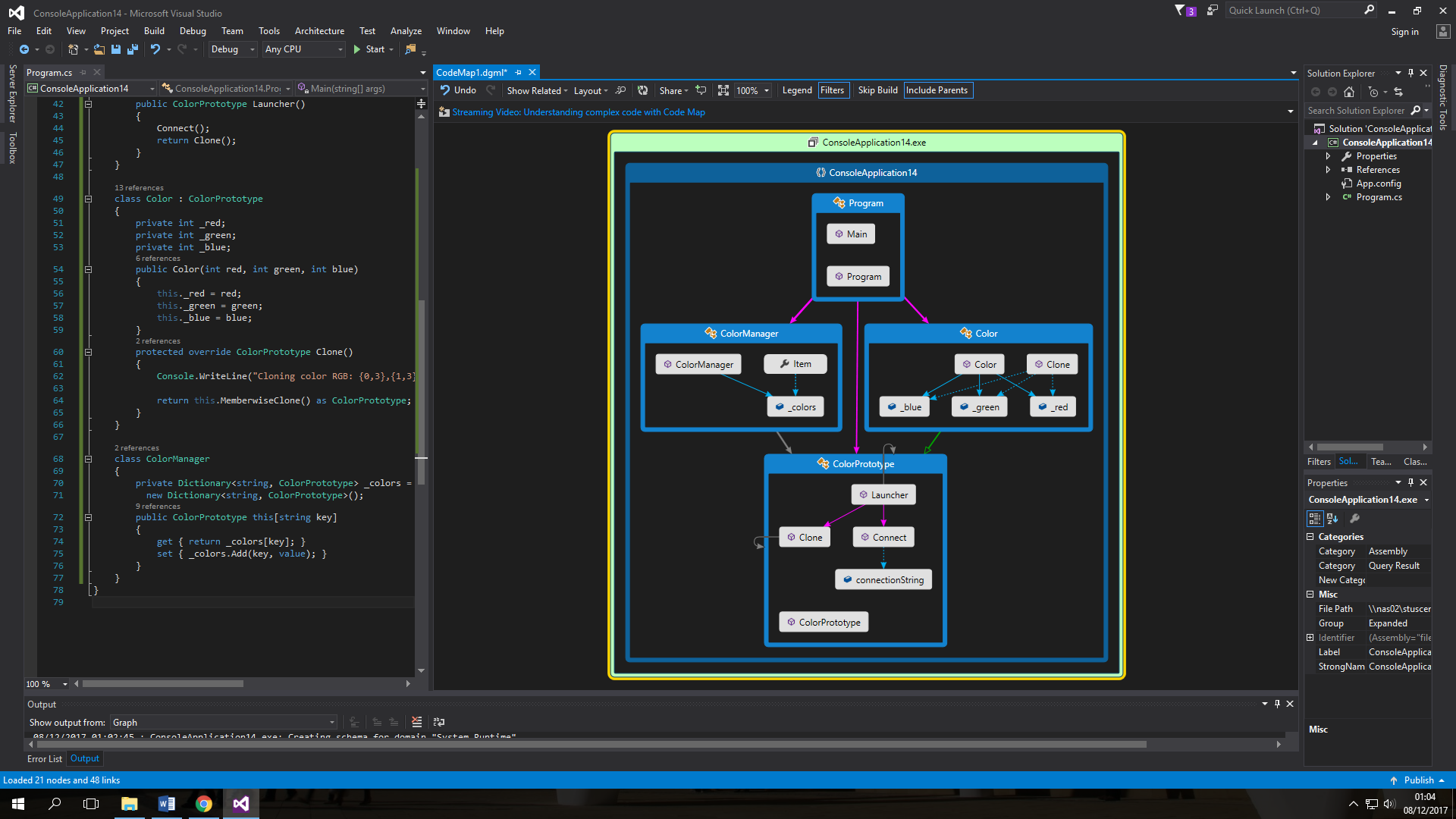


Figure 5.8: Code map for the Prototyped Template generative pattern

##### Templated Builder

Continuing with the trend set in the prototyped template, the next artefact is the templated builder. This sees the combination of the builder pattern with the template method. Though the builder pattern is a more complicated design pattern, its setup still follows the same configuration of a subject with an encapsulated main method and objects. In this case, the builder’s main methods are the set of methods responsible for creating the different objects that the abstract builder’s subclasses create.

These methods are the ones that will be overridden any number of times depending on the specific nature of the object being built and called by the director class in order to compose the required object.

For the builder pattern to work with the template pattern, these methods would be encapsulated within a template method. This limits the director class’s access to these methods and by extension their parent classes at the point of exposition effectively decoupling the system.

Also, the director class then only has to call the template method instead of each builder method, thereby reducing the amount of code that would be written and any future changes to the builder classes primary methods would need to be implemented just once.

The advantages of this artefact lie in the decoupling abilities it affords the system. Effectively reducing or limiting intimate knowledge of the contents of the builder and allowing all communication or calls to the builder class be executed via its template method. As a result, the low-level classes are much less dependent on the high-level ones.

|  |
| --- |
| Builder Pattern |
| **Intent**:  Isolates the process of constructing a complex object from its representation in order for the same construction process to be used to generate different representations. |
| **Purpose**: Variant Management |
|  |
| Template Method |
| **Intent**:  This pattern describes the skeleton of an algorithmic operation, conceding some steps to the pattern’s subclasses. In effect, this permits the subclasses to redefine the deferred steps without effecting significant alterations to the algorithm's structure. |
| **Purpose**: Variant Management |
|  |
| Templated Builder |
| **Intent**: streamline the construction of a complex object that is comprised of objects assembled in a pre-known sequence. Reduces the awareness client class has of the assembly process. Control the process of building complex objects. |
| **Purpose**: Control |

Table 5.8: Table showing the purpose of the templated builder

**Scenario**: For this scenario, we consider the creation of meals at a fast food restaurant. The restaurant only serves breakfast and dinner to cater to busy customers in a highly competitive area. The Typical meal is made up of main order, a portion of sides and a drink. As a result, regardless of meal type, breakfast or dinner, the process of construction is constant. But because the restaurant is in competition with others it is crucial that the process of making their meals are kept secret from the general staff.

Mapping this scenario to the participants defined in the templated builder in Chapter 4, we have the following; the AbstractBuilder corresponds to the kitchenMeals class, the concreteBuilders refer to the two meal types and the director maps to the salesperson class.

Figure (4.7) depicts the class diagram that results from the templated builder. The scenario looks at the differences between the class diagram that can be constructed for the generative pattern versus that for a more static representation of the code.

The introduction of a template method into the builder pattern’s abstract class greatly reduces the awareness the director class has of the build process. This in effect promotes the open and closed principle, reduces the flexibility the builder affords the system and improves the degree of decoupling within the system.

In the static depiction, the builder methods of the Abstract builder are set to public. This means the director class maintains an innate and complete awareness of the process of constructing the product or in this case, the meals served by the establishment. This poses a security risk for them in case the individual decides to trade this information.

The implementation of the static AbstractBuilder and director classes are shown in table (5.9) below:

|  |
| --- |
| public abstract class kitchenMeals  {  public Meal meal = new Meal();  public abstract void addDrink();  public abstract void addMain();  public abstract void addSide();  public abstract Meal getMeal();  } |
| public class SalesPerson  {  public Meal doMeal(kitchenMeals cheffsMeals)  {  cheffsMeals.addDrink();  cheffsMeals.addMain();  cheffsMeals.addSide();  return cheffsMeals.getMeal();  }  } |

Table 5.9: Static AbstractBuilder and Director classes

It shows the existence of significant duplication of code in both the director and client classes. This is manageable in scenarios where there is a very limited number of builder methods required to compose the final object but could become tedious in scenarios with large numbers of builder methods.

The class diagram and code map from the static representation is given in figure (5.9) and (5.10) below.

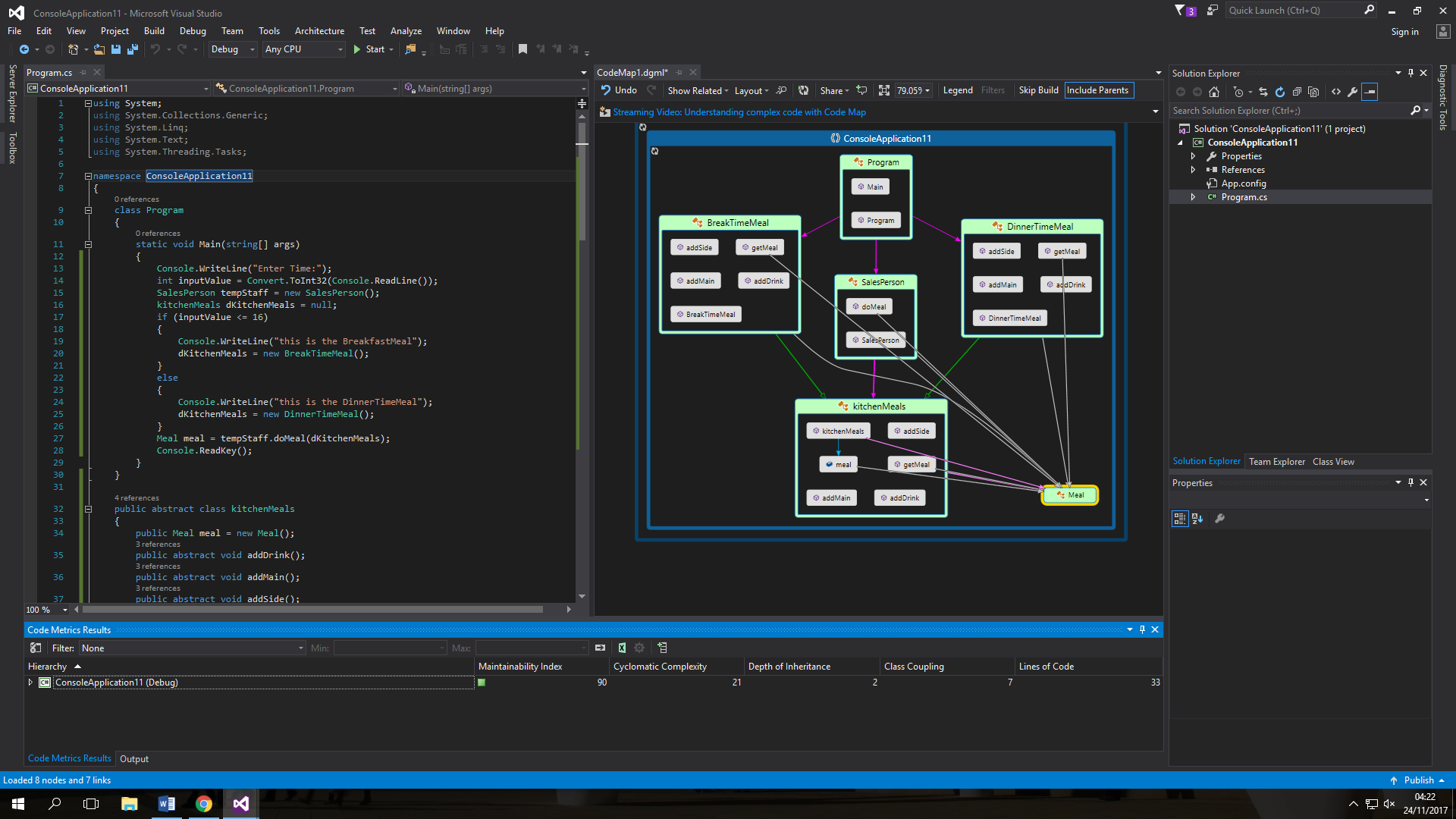


Figure 5.9: Code map for static representation

In order for the business to reinvent itself and secure its recipes, a template method is introduced. This template method encapsulates the process of building the product which can then be set to private. As a result, the director class is decoupled from the creation process and maintains limited awareness of the recipe for each meal type. Table (5.10) depicts the modified abstract builder and director classes.

|  |
| --- |
| public abstract class kitchenMeals  {  protected Meal meal = new Meal();  protected abstract void addDrink();  protected abstract void addMain();  protected abstract void addSide();  public void makeMeal()  {  addDrink();  addMain();  addSide();  }  public abstract Meal getMeal();  } |
| public class SalesPerson  {  public Meal doMeal(kitchenMeals cheffsMeals)  {  cheffsMeals.makeMeal();  return cheffsMeals.getMeal();  }  } |

Table 5.10: The Generative AbstractBuilder with its protected methods and Director classes

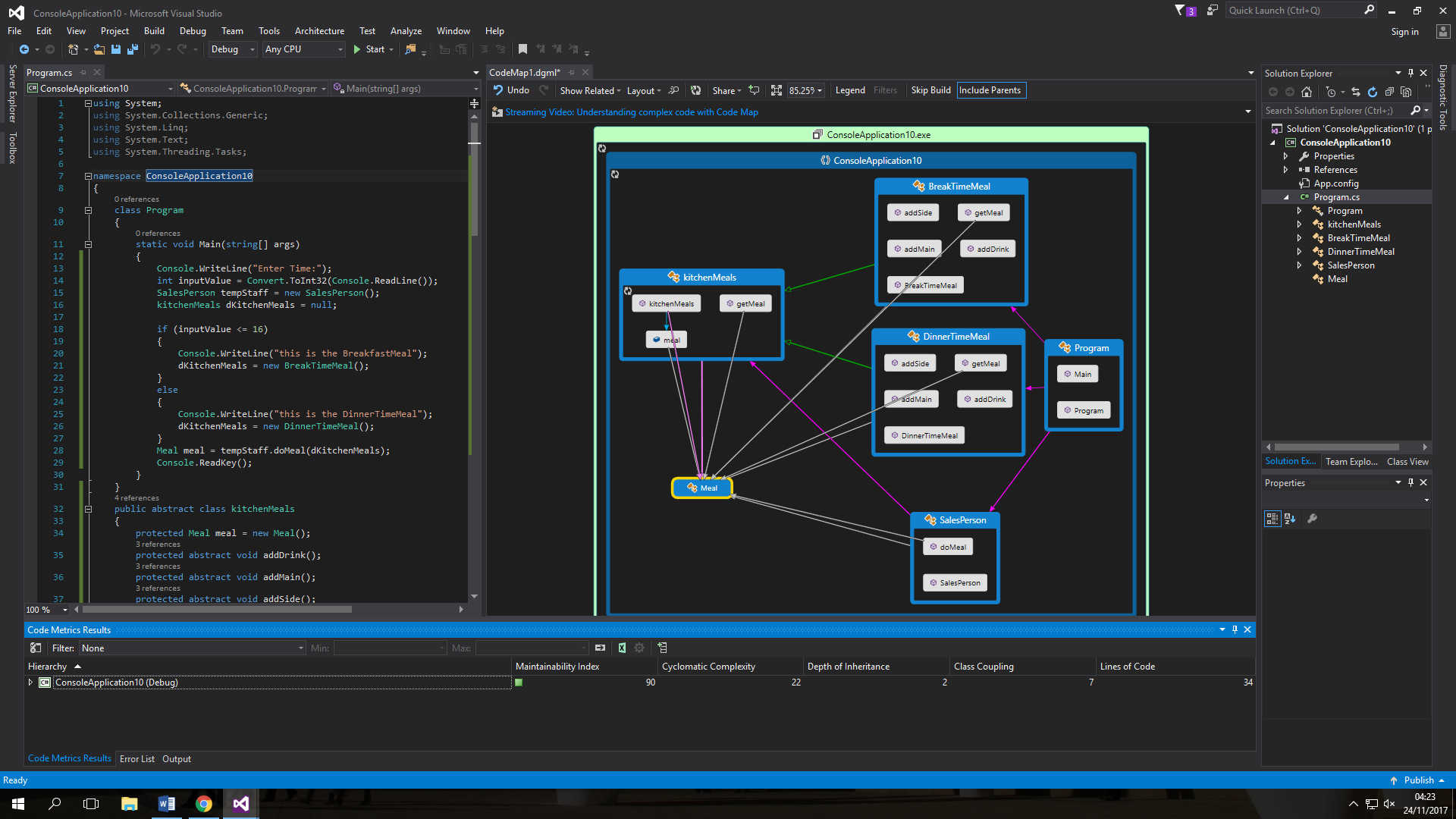


Figure 5.10: Code map of generative construct

As a result of the template method makeMeal(), the director or salesPerson has no concrete idea what goes on in the AbstractBuilder and only serves to prompt the builder to create the products.

The main requirement for this combination to occur is the homogeneity of the build process, as this is the part that the template method encapsulates. In order to vary the build process, the concrete builders would themselves have to be modified.

Having explained the process via which each generative construct is developed, the assessment moves to comparing performance criteria with the aid of code metrics.

#### 5.3.1.2 Dimension 2

The DSR evaluation process stems from prescribed paradigms dependent on the narrative of the research. While there exist numerous ways to characterise such paradigms, the prescriptive and functional nature of design science research stresses a distinction that is less philosophical and more practical argues (Gregor & Hevner, 2013; Venable, Pries-Heje, & Baskerville, 2016). This paves the way for the application of more artificial evaluation methods as opposed to naturalistic ones.

This is because artificial evaluation, being almost entirely positivist and reductionist in nature (Venable, Pries-Heje, & Baskerville, 2016) can be explained by the use of a system of metrics as introduced in the literature review. And since artificial evaluation in this manner stems from the scientific process, by which we mean the application of laboratory experiments, simulations, criteria-based analysis, theoretical arguments, and mathematical proofs (Nola & Sankey, 2014), it benefits of greater scientific reliability in the form of better repeatability and falsifiability (Gummesson, 1995).

**Metrics**

The subject of metrics was introduced in the literature review. There, it was defined as a standard of measure that represents the degree to which a software system or process, possesses some identifiable property that gives an indication of its performance in relation to others (Olsen, Haug & Bergman, 2001). Here individual metric measures are explained to better give an understanding of how these measures relate to code performance.

* Maintainability Index: This value measures the ease with which code can be maintained with a scale of 0 to 100. Scoring higher values are positive and indicate the code is easier to maintain. The generic formulae for calculating MI is given as:

“Maintainability Index = MAX (0,(171 - 5.2 \* log(Halstead Volume) - 0.23 \* (Cyclomatic Complexity) - 16.2 \* log(Lines of Code))\*100 / 171)”

* Cyclomatic Complexity: This evaluates the degree of complexity of the operations performed by objects within the program rather than the operations the objects perform. The lower the complexity the more modular the system and is thus a favourable result.
* Class Coupling: This is an indication of the degree of integration between the objects and classes of a program. Loose coupling is a concept that is given high favouritism in code design as it promotes reusability and maintainability. As such the CBO metric counts the number of inroads within the program as depicted by the number of attribute declarations, method selections, local variables, object and return types. A high degree of integration hence low degree of loose coupling implies that the program is more sensitive to change which adversely affects maintainability.
* Depth of Inheritance: This is defined as “the maximum length from the node to the root of the tree” (Chidamber & Kemerer, 1994). Depth of inheritance is predicated on three fundamental observations. These are:
  + The deeper an object class is located in the hierarchy, the greater the number of methods it is likely to inherit which makes predicting its behaviour more difficult.
  + Deeper tree hierarchies typically involve greater design complexity because more classes and methods are involved.
  + Deeper class hierarchies in a piece of code’s inheritance tree possess a greater potential for the reuse of inherited methods than shallow hierarchies.

Observations 1 and 2 seem to indicate that a higher depth of inheritance value is disadvantageous. However, observation 3 signals that a higher code depth value is advantageous to potential code reuse.

* Lines of code: The “lines of code” metric is a fairly common metric when assessing the performance of a program. It evolves from the belief that computer code as a matter of principle must be short and to the point as this favours the understanding and maintainability of software code. Furthermore, since this metric contributes to the size of the eventual code file, lines of code should be optimized.
* Number of classes: This is a fairly simple metric that stems from the idea that code with fewer classes would be easier to maintain and control. The NOC metric, therefore, counts the number of classes including abstract, interface and virtual classes in a code sample. The NOC metric is not a particularly reliable means of judging the efficacy of a piece of code. This is especially true in object-oriented languages where object-oriented principles typically necessitate the creation of more classes than in functional languages.
* Weighted methods per Class: This indicates the degree of complexity exhibited by the methods implemented in a program. It affords an estimation of the amount of ardour that goes into developing such a program because it measures the number of methods present versus the degree to which they are used and their scope across the system. The philosophy behind this metric is that the more complex the program the more specific it is which in turn diminishes its reusability, adaptability and in some cases maintainability.

Having defined the individual metric measures, this section will go on to detail the values recorded for each artefact and its static counterpart. This allows for a side by side comparison of the apparent performance of the artefacts in an attempt to fulfil the evaluation criteria defined in Section 5.2.

##### Templated factory

|  |  |  |
| --- | --- | --- |
| TEMPLATED FACTORY | | |
|  | M I | C C | Class C | D O I | L O C | N O C | W M P C |
| Static | 93 | 27 | 10 | 3 | 35 | 10 |  |
| Generative (Compound) | 94 | 26 | 8 | 3 | 33 | 8 |  |

Table 5.11: performance metrics for templated factory

The performance metrics as shown by table 5.11, reveal that the templated factory slightly outperforms its static counterpart in the areas of maintainability, lines of code and cyclomatic complexity. Although slight, these values indicate a more streamlined approach to achieving the same results as the static variants. The maintainability values, in particular, show that the system is more decoupled and extensions or changes to the parameters of the scenario would not require extensive modification of the code in multiple places to effect.

##### Prototyped template

|  |  |  |
| --- | --- | --- |
| PROTOTYPED TEMPLATE | | |
|  | M I | C C | Class C | D O I | L O C | N O C | W M P C |
| Static | 88 | 14 | 8 | 2 | 29 | 6 |  |
| Generative (Compound) | 82 | 11 | 6 | 2 | 27 | 4 |  |

Table 5.12: Metrics for the static and generative prototype and template methods

As concerns the prototyped template, it is possible to observe a significant drop in performance especially as relates to the maintainability. All other metrics taken individually show improved performance, and this implies that the efficacy of the generative pattern in this case is dependent on context. The cyclomatic complexity and class coupling in particular record a significant increase indicating the system is more maintainable due to the fact that any future modifications to the code would only need to be performed once in a single location or class. It is possible to surmise that the efficacy of the generative artefact in this case depends on the operation or use case being implemented.

##### Templated builder

|  |  |  |
| --- | --- | --- |
| TEMPLATED BUILDER | | |
|  | M I | C C | Class C | D O I | L O C | N O C | W M P C |
| Static | 90 | 20 | 9 | 2 | 31 | 6 |  |
| Generative (Compound) | 90 | 22 | 9 | 2 | 32 | 6 |  |

Table 5.13: Metrics for the static and generative builder and template methods

In the templated builder, the metrics indicate that any differences between the static and generative variants are negligible in terms of maintainability. This is shown in table (5.13) which shows that both structures score relatively similarly on a majority of the performance measures tested. The difference arises when one considers the sequence diagrams of the two constructs and any new characteristics the combination with a template method evolves. There is a notable increase in cyclomatic complexity probably as a result of an increase in the levels of abstraction present within the structure, although the increased decoupling exhibited by the generative structure implies that the system is more maintainable and possibly more efficient.

## Conclusion

This chapter in accordance with the evaluation demands of the design science research methodology has assessed the efficacy of the developed generative design pattern. Using the FEDS framework, this assessment considered two primary evaluative dimensions, the first, a functional assessment of the developed patterns which was presented in section 5.3.1.1 and a performance or metrics-based assessment which was presented in section 5.3.1.2. The functional assessment revealed that the process of introducing a template method into the abstract class of the creational patterns considered provided these patterns with increased levels of decoupling, security and abstraction. This is because the template method affords the other patterns the ability to manage similar structures in a homogeneous manner.

This is the primary effect of merging the creational methods of the creational patterns with the control method of the template method.

The performance metrics indicate that the generative artefacts perform significantly better than their static counterparts in the area of code maintainability. This means that any future modifications to the code could be implemented with little difficulty and effort.

It was observed that there is an element of context to be considered when applying these generative artefacts, this is exemplified by the prototyped template and signals that these artefacts being patterns themselves are subject to the disadvantages that exist with overt use of patterns.

These disadvantages range from increased complexity to sometimes inefficient implementation due to the violation of some crucial principles of object-oriented programming.

As a result, it can be concluded that generative artefacts although effective performance wise still require in depth analysis and evaluation before deployment within code and as such, their primary benefit would be in achieving rapid prototyping of scenarios or use cases which can then be rigorously tested before final production.

# Chapter 6

CONCLUSION

## Introduction

In this chapter, the main findings with regard to the research questions detailed in chapter 1 of this thesis, as well as the general content of the thesis and research process are summarised. This summary ascribes a number of conclusions based on the research’s findings and uses this to illuminate the strengths, limitations and contributions this research makes to the subject area.

## Thesis Summary

This section details the main content of each chapter of this thesis as it paints a detailed picture of how the research work was undertaken, and help provide context for the conclusions that will be presented later on in this chapter.

Chapter 1 of this thesis introduced the subject of design patterns and some of the limitations they possess. These limitations ranged from their inflexibility to their innate rigidity which is at odds with the demands of complex software development projects that often require the co-operation and sometimes combination of multiple design patterns. A discussion on these limitations invariably necessitated a look at how design patterns functions and from this, the chapter drew parallels between design patterns as a field of study and generative programming. The parallel stems from the fact that both fields of study deal with the application of a given set of rules to the solution of multiple related problems. In the case of design patterns, the application of the rules of object-oriented programming to create structures that solve a family of related problems and in the case of generative programming, the application of the rules of software development to the automated generation of code which is simply the solution of problems in its own right. This allowed the thesis to segue into the subject of generative design patterns and explore the different attempts via which the subject of generativity is explored by different researchers. Attention was drawn to Wilson’s (2008) take on the subject of pattern generativity which posited that generative design patterns could be realised from the combination of two or more classical design patterns. This combination would see the resulting artefact limit the disadvantages of the constituent patterns while also exhibiting resultant characteristics that would make such a structure advantageous to the realisation of specific functionality at reduced cost to the developer or the system’s resources.

Having introduced this, the thesis, through Chapter 2 explored the existing literature on the subject of design patterns, the relationships they maintain with each other, how generativity is defined and how the performance of code artefacts are assessed. The chapter revealed that software design patterns are at their core, an abstracted representation of a concrete form (Zlobin, 2013), in this case, the structure of a particular functional body of code which persists in specific non-arbitrary contexts. These structures are functional in that their presence in a body of code results in the solution of specific code design problems. This realisation paved the way for questions regarding how these abstracted forms interact with one another. The chapter consequently focused on the relationships identified by Noble (1998) and Zimmer (1995) and went on to describe how these relationships factored into Wilson’s (2008) definition of a generative pattern.

The relationships identified from the literature can be broadly categorised into three. These are:

* The uses relationship
* The similar relationship
* The combines relationship.

Chapters 1 and 2, provided enough information on the subject of patterns and pattern combination that it was necessary to start prototyping the combinatory patterns. It was crucial that the process of prototyping these combined artefacts be documented in a manner than was systematic, understandable, reproducible and designed in a manner that delivered on the objectives stated in Chapter 1 in order to lend more credence to the veracity of the developed artefacts.

To that end, a more rigorous definition of the methodology that was being employed to deliver on the thesis’s stated objectives was required. Consequently, Chapter 3 was devoted to defining the methodology and elaborating on the manner in which it was applied. The chapter presents the name of the methodology, given as design science research and the justification for it. This justification lay in the fact that the primary aim of this research was to develop a number of artefacts that conformed with a pre-set definition of what a generative pattern is and aligned with the stated objective of design science research as a methodology. After this, the chapter goes on to detail the various stages involved in realising the generative artefacts as relates to this methodology focusing on the relationship between the chapters of the thesis and the three-cycle view of design science research.

Chapter 4 goes on to define the developed artefacts using the generative design pattern framework developed by Wilson (2008). The artefacts included in this chapter are:

* The Template Factory Pattern
* The Prototyped template
* The Templated Builder.

Having defined these generative constructs, the chapter introduced as one of its findings, a category of design patterns which could not be combined with other design patterns to produce a generative construct. This category of patterns is called “non-combinatory patterns” and the chapter exemplified this category with a discussion of the visitor pattern.

The chapter showed that by virtue of its composition the visitor pattern does not possess a singular definitive method or a primary method that could be combined with the definitive method of another classical design pattern. Consequently, it was difficult, if not impossible to realise a scenario whereby the visitor pattern could be combined with other static design patterns in the manner espoused by the concept of generative compound patterns subscribed to by this thesis.

Having identified the main outputs of the research, and in step with the dictates of design science research methodology, it was crucial that the outputs be evaluated. This investigation was geared towards ensuring that the developed generative compound patterns were addressing the stated objectives of the research effort and met the criteria for definition as generative design patterns.

To achieve this, the artefacts were assessed based on two main criteria.

The first criterion was geared toward examining the functionality the resulting or developed artefacts possessed and how these functionalities made such artefacts unique. This investigation focused on any functional benefits the artefact afforded the developer in particular scenarios as opposed to the static variants and was detailed in the definition of the individual generative constructs.

The second criteria focused on assessing quantifiable performance metrics of the developed artefacts. This was achieved with the aid of code metrics software and metric measuring functions in popular integrated development environments (IDEs). These allowed for certain qualities or characteristics of the realised patterns to be quantified and compared with those same quantities in scenarios where the constituent patterns were operating statically.

The results showed that the developed artefacts not only outperformed their static counterparts but also delivered a number of functional abilities that improved code creation and development.

The combination of the behavioural pattern template method with each of the creational patterns allowed for improved abstraction and decoupling, which introduced finely grained control over the operations executed within the system.

## The Contributions of the Study

The primary contributions of this research work are as follows:

* This primary aim of this thesis has been met via the identification and development of three generative patterns. These generative patterns are the templated factory, prototyped template and the templated builder. These patterns further reinforce Wilson’s (2008) ideas about the relationship between generative programming and generative design and showed that the combination of classical design patterns could indeed result in structures that were more dynamic and possessed advantageous behaviour not afforded by the constituent patterns independently.
* This thesis defined a more stringent process for realising the generative combination of classical design patterns into generative constructs. The process involves the merger of the primary methods of the constituent patterns.
* The identification of the “non-combinatory patterns” category with the definition of the visitor as the first of this category due to its unusual mode of operation

## Critiquing the Study

Although these findings are generally compatible with the findings Wilson (2008) recorded in his work on a framework for a generative design pattern, there exists certain critical areas where the arguments presented here diverge from his.

First and foremost is the manner of combination. Wilson in his experiments passed of some incidents of interaction as combination. This largely resulted from the fact that his work did not consider the existence of compound patterns and as such falsely recorded some compound cases as generative when they are not.

Another point of divergence is in the identification and selection of compatible patterns. Wilson (2008) relied solely on the relationships individual patterns maintained with other patterns in the Gof handbook as was identified by Zimmer (1995). This limits the number of possible combinations as Zimmer (1995) only identified six patterns that maintain a “combines” relationship. This thesis however looks at the functional application of the patterns as well as the relationships they maintain.

This means that in choosing combinatory patterns, the class of pattern as well as the relationships the pattern possesses are crucial to achieving or developing effective generative artefacts. For example, the template method helps hide the process of creation as performed by the factory, builder and prototype patterns. This effectively decouples the system and introduces a level of control over the creation process of the creational patterns. It is able to do this because the template method is a behavioural pattern and so allows for fine grained control over the actions executed within the body of code. That action in the case of the structures presented here is object creation.

This approach reinforces the idea that design patterns are intended to solve specific problems and increases the likelihood that other combinatory structures could be developed.

A major point of critique lies in the fact that this thesis focused on the combination of not more than two classical design patterns. The rationale behind this stemmed from Wilson’s discovery that the structure of the generative construct began to break down as the number of combined patterns exceeded three. This made sense considering the manner with which pattern selection was made. Since Wilson (2008) based his selection entirely on the “combines” relationship as identified by Zimmer (1995), he was limited in how these combinations could be implemented. This research considered the function of the patterns as well as their relationship and from that it might be possible to identify other relationships that could accommodate the combination of three or more classical design patterns.

Another point of critique is the research design. This thesis applies the design science research methodology. A research model that has its roots in engineering and the sciences of the artificial (Simon 1996). This implies that it is at its core, a problem-solving approach that pursues the generation of innovations that are defined by ideas, practices and technical capabilities which produce products via which the design, implementation, and use of information systems can be effectually and efficiently accomplished. This quality of design science research is well suited to this thesis but fails to consider the human element that is as much a part of software development as the implementation of code solutions. Patterns, both classical and generative are tools that would be used by humans and this research doesn’t explore how developers interact with the developed artefacts at both the design stage and the subsequent implementation stage. An examination of how software developers interact or interpret generative code would serve to flesh out the findings of this research and could be an area of future research in this field.

It is important to stress that this study has been primarily concerned with the combination of one design pattern with another. There undoubtedly would exists generative constructs made comprised of more than two design patterns but in light of Wilson’s discoveries of the instability of the artefacts care was taken to limit the identified artefacts to those comprised of two.

Limitations of the Study

## Implications of the findings

The findings from this research indicate that generative design constructs are indeed a viable solution to the problems of rigidity that plague classical design patterns. This is because, generative constructs attempt to introduce additional beneficial; behaviour which might have been difficult to implement with the static approach without significant overhead to the code being written or the functionality being implemented.

The development of the three generative constructs put forward in this thesis furthers the conversation on how design patterns work together within code

A generative design pattern framework was modified and applied in the definition of a number of generative design patterns. These constructs represent a small number of the sum total of combination experiments that were conducted during this research. The breadth of this experimentation revealed a number of pathways or rules that could further define how individual classic design patterns could be combined.

## Conclusion

This chapter concludes the work undertaken to extend Wilson’s generative design pattern catalogue. The point was made in the introductory chapter that the aim of this research was to populate the generative pattern catalogue with a view to catapulting generative design patterns into mainstream software development practice. As such, the aim of this research has been met via the definition of three generative constructs and in particular definition of the rules that define the process via which these constructs come into being.

# Chapter 7

FUTURE WORK

## Introduction

If we were to consider the exploration of the subject of generative design patterns as a field of study, then the process of pushing it into mainstream discussion in the field of software development would be akin to an endless methodology and future work would be commensurate with the rigour phase of the design science research. This is because future work is crucial to testing the rigour of the findings of this research and subsequent ones in the field.

This chapter discusses a number of avenues or paths future research could take. It starts of by recognising the existence of a number of question that were encountered during the course of this research that are not explored in this thesis. It goes on to show that these questions could lead to discussion concerning the identification of patterns that fall into the non-combinable design pattern category introduced in this thesis.

Secondly, it looks at the possibility for the development of more refined formula or set of formula for generating generative design pattern.

After this, the chapter discusses widespread testing of the identified patterns and how these would lend itself to better classification of the identified constructs.

Finally, it discusses the benefits of a CASE tool, dedicated to promoting and implementing generative design pattern in the solving of commonplace design and development problems.

The body of research encountered in the process of conducting this research raised certain questions that remain unanswered both within and beyond the scope of this thesis but would make for alternative avenues of research.

## Investigating non-combinable design patterns

One such question regards a more formal investigation of the non-combinable pattern category of patterns. This thesis identified the difficulty of combining the visitor pattern with other patterns in the manner detailed by the combination formula identified in this thesis. Based on this, Chapter 4 of this thesis introduced the non-combinable pattern category, which it argues, is comprised of those classical design pattern whose operation or code representation makes it difficult to identify a primary method via which they can then be merged with the primary methods of other design patterns. This suggests that there might be other classical design patterns that could fit into this category which haven’t been discovered. A formal investigation of these non-combinatory patterns would go a long way in further fleshing out the reasons why this category of patterns exists or might discover alternative methods which circumvent this classification and make these patterns useful as generative constructs. This would go a long way in limiting the scope of the investigation into the process of combining pattern. This is crucial in light of the difficulty of attempting to combine each pattern with every other pattern in the gang of four. There are 23 individual design patterns in the gang of four catalogue and attempting to experiment with each possible single pair combination would require 529 experiments. This number will no doubt increase as the number of combined patterns increases from two to three or more.

This could lead to the creation of a definitive formula for the creation of a generative construct. This research discovered that the creation of a generative construct is possible via the merger of the primary methods that define each design pattern and used this formula to generate the generative constructs presented in this thesis. This is not to say there might not exist alternative ways or formulae for combining the patterns. Further study and the elimination of non-combinable pattern will largely result in a refinement and possible development of the formulae used in this research.

## Stress Testing the identified generative patterns

Another possible line of research is field and stress testing the identified constructs. Generative patterns are designed to afford software developers more dynamic constructs via which optimal solutions could be afforded to commonly encountered problems. Although the evaluation chapter indicates a positive bent to the findings and its suitability to solving the problems of static design patterns, the focus was on performance testing which is somewhat different from stress testing the constructs.

Stress testing as defined by (Singh, 2012) refers to a non-Functional testing technique that is intended to break the system or the subject of the test. For this, both the use and performance of the generative constructs is observed after subjecting the pattern to an overload of relationships to ensure that the system can sustain the stress. This would mean taking the construct out of acceptable conditions. In the case of a generative pattern, this could mean the application of the generative construct in applications that require its interaction with numerous static or other generative patterns. Although not primarily used to assess design patterns, Wilson’s discovery that any attempts to combine more than 3 patterns leads to unstable constructs suggests that there is a possibility that generative patterns are limited in their ability to work with other patterns or within other constructs. This begs for further investigation and would make for an informative line of enquiry.

Furthermore, the recovery of the generative pattern from such extreme use (after stress) is and would be very significant as it would indicate that the generative pattern is a standalone entity that can maintain any number of crucial relationships that are unavoidable in a development project. Here, the true test of efficacy could involve presenting a large number of developers with the generative constructs as tools and tasking them with solving problems. This would go a long way in testing the flexibility and utility of the constructs in the real world.

## Generative pattern CASE tool development

Wilson (2008), in his work on a framework for generative design patterns suggests the development of a case tool that primarily caters to the application of generative design patterns to software development exercises. Classical design patterns enjoy ample representation amongst such CASE tools and are easily recognised by several integrated development environments (IDE)s. The development of a CASE tool could catapult the concepts surrounding the combination of design patterns into generative artefacts, into the mainstream software design discourse and place a generative design pattern catalogue on a similar footing to the more popular static ones.

## Conclusion

This chapter’s sole focus was to provide direction for future research into the subject of generative design patterns. This is because, although the term has existed for more than a decade, there is no consensus on how the ideas the term connotes can be achieved. Detailed in this thesis is one of several competing approaches to realising what the term means and as such, before a particular approach is declared as prevailing, significant research that reinforces the viability of the concepts each approach lays out is crucial.

This chapter has shown that such research could continue by expanding the repertoire of generative design constructs as achieved by this research, identifying other possible combination formulas, determining the boundaries of generative design patterns by the possible discovery of other non-combinable patterns. These avenues for future research would significantly improve the viability of these constructs especially as they would focus on or require the creation of scenarios for stress testing the identified generative pattern discussed here.

Finally, an important area of future research could be the creation of a case tool that leverages on the concepts put forward here.

# References

1. Alan R. Hevner, Salvatore T. March, Jinsoo Park, & Sudha Ram. (2004). Design science in information systems research. MIS Quarterly, 28(1), 75-105. Retrieved from <http://www.jstor.org/stable/25148625>
2. Alexander, C. (1979). Theœ timeless way of building. New York: Oxford Univ. Press.
3. Alur, D., Crupi, J., & Malks, D. (2003). Core J2EE patterns: Best practices and design strategies (2nd ed.). Upper Saddle River, N.J; London: Prentice Hall PTR. Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwbV3JDoIwEJ24XExMFJeIS8IPqLRAgTPBGM8mHk0LJXpBo_j_TisQt2N7mLSZzvbaeQVw6MpefvkE1IVEb0m4n_gZE45QuidJEGbS5ZSzT2Kmmna7At5Oj7RsrUp4oW421-Is1pgJaLbtpueps00OTo2uqMhImS7DiKNgDpe5Jd1ONcacsoly36LJpg8t1WFgQEPmA-hV_ypYpZkNwYguN2ntaBxbV01_md9HYG7ifbRdoqxjCbkcXyujY-hy9U49L3Q_WzoBC-0lCGSYSBGqX3-J8LjtZ2HKsU5NOfNMMH4FTf9NzqCjH5ZpOGAO7QyPq1zoPT0BYXNijw>
4. Angood, P. B. (2013). Metrics, measurement and reporting. Physician Executive, 39(1), 95. Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwnV1LS8NAEB6sggjFR7VarRAEb6ZmH9lJTlLE4qW3gngK-0o9VW3a_-_uJiVFxYOH5LJ7CDM7s5Nv95sPgNFREn_LCSJDJTOjnf9ThYxTJYymWpZYGkMDA2CrMVNN6vdIQe3tTZIMmdu8aw-a3_vE6sqdlIuHj8_Yy0j549ZGU6MDe0Qgei0HfHlt73zwoDCaZLlbHuj7k_6eh8PmMjkC2yIqMugXjt7WpiF5_eja-O-PPYbDpvqMxvVyOYEdu-jB_rQ5X-9Bt0bxopqcdAo3Uy-4pau7aNpiiZFcmKgu3N2-dwazydPs8TluVBXieZaJmFhhU5PLBI2ixj0uAktOSoWomcmpsq7cTqRgBqXVmHJVWiYFzam1DFXC-tCV_vL9YhVIeuYCIkt4SSTxfW04R02Ucb62QmlCpCWaDeDWm7lodDXdq_LIQzWX66oqxq6EQf-fmQzgPMzzsbVaSl1sjQw3di2aIKuK1qiXfw9fwQENKhYeORnC7mq5ttfQcb78AlJayvI>
5. Basili, V. R., Briand, L. C., & Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering, 22(10), 751-761. 10.1109/32.544352 Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwnV3JTsMwEB2xXJAQZasa2koW4pqSOHESn1AFVHwACHGyHDsRHGhLk0rw93gcpwtIHDhbsRzNeBbPzHsAER0F_g-bkDKqozJKUOix8RgISGlykzyULC8Cyz2xAcwELSqEk3ZrJK3l1jOFj-bXIWcMqXPpzfzDRxYprLY6So1d2A-TNEUqh_T5Zd3ykaasxdBkLOMOaSgM-LU5NCLBMbrln5yVtrwr2yEozowsqw1XNOmAXr-_SMt2OHpdajcS9gvj8b-_dgxHLlQl40a3TmCnmJ5Cp6WBIM4qnMHdmBh1fWvImcisJLMcH3dwSwT81ETbLhHyjuRdqiKyIs0o5xfBgrnCtL86h6fJ_ePtg-_IGXxlYijuU8qVzuLY5CvY52FcLULF0VRJJUsTVqg0KcI8YxoTHqozXga8zHlm6bFCY9W6cCixiX9a22E_3QMSK57xIFEI0mcCDazZySSKjVFiEeUZ8-CylY2YN2AcwiYxARcRFY0APeii1ARe0HohlTAhD6LKcw-GKMfVl4ENkJPYhMw8jArtwaCVhpA5PjSpuhIrWXjQXy27e7612mtUYn0wajQQi5AeXP1YoqKiIrB1_4RiTijqz_riz_37cGD7xW0b4QD26sWyGMKuUbBvVxYBbQ>
6. Belk, R. W. (2006). Handbook of qualitative research methods in marketing. Cheltenham: Edward Elgar. Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwbV3LCsIwEFx8gHjziU_ID6g1aVJ6FsUPEDxK0iS0B_Wg_r-btqlFPfSQQpIWwu7OtDMLwOg6WH3FBEx8KtJhaARLEmWolbGysYkjaYNA560Pa8ZMle22J97Sly6lVYl8ui-bG5WpDWcYacMmAi7uzvb2zCp2BQsZzHys0IzzkDoWuzR48mOXj3DdWjY59KDlFAZ9aJjbADr-1_Mh8CPCelf1krslhdoxt-UmpSNPSop-zw-S3cjVy5VHMD3sT7vjCre5lGzMpXhoOoYWwnszARJJwYTGOoBZjpdCAIQgK5HCCmeVwqbQ_50_-3dzDt0PQbCAtsUDbJb5W74BiD9qog>
7. Bevis, T. (2012). Java design pattern (2. ed. ed.). Essex: Ability First.
8. Bishop, J. (2008). C# 3.0 design patterns (illustrated edition ed.). US: O'Reilly. Retrieved from <http://lib.myilibrary.com?ID=613012>
9. Bonfe, M., Fantuzzi, C., & Secchi, C. (2013). Design patterns for model-based automation software design and implementation. Control Engineering Practice, 21(11), 1608. 10.1016/j.conengprac.2012.03.017
10. Borchers, J. (2001). A pattern approach to interaction design. AI & Society, 15(4), 359-376. 10.1007/BF01206115
11. Budd, T. (2000). Understanding object-oriented programming with java (Updat ed.). Harlow; Reading, Mass: Addison-Wesley. Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwbV3JigIxEC1cLoIw6iiukB9otbtjmz7LiHhWPEplafQwelG_3ypNi9sxCVQSSKUqL3kvAHE0HAdve4JWEyTXykxopVEUYrRyqU7NOHMmmab2VZjpIbudA2-7s_XUKoMnvtkc6b0exbQDh6pIB64Ju2S44U8bKP2hJIaCcurldfIyE1DIzlP0mNegxIyCOhTcoQE_-T8KwrvVL8TrZ4aJOGrGRoIjKxBTPij8E6p_bmPYVCzxgk3ozP9Ws0VAnW09BrO9DzVqQRX54frhdCO42TaIBFHLxCUuMlKiRmTp4MhZ6UKrKIB1oP5pqPutsgeVO0-c8YE-lDNav25wm_QVag5umw>
12. Buschmann, F. (1996). Pattern oriented software architecture. Chichester [u.a.]: Wiley.
13. Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 20(6), 476-493. 10.1109/32.295895
14. Clark, B. (2002). Eight secrets of software measurement. IEEE Software, 19(5), 12-14. 10.1109/MS.2002.1032844 Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwnV07T8MwED7xWJAQ70eBShlAYklxY8epJ4QQFQsTIMRkObYjFl5NKvj53LlOS0FiYE50ce7se9jn7wPgWY-lP3wC1lkDaXhACxNKOu55XmFo8AW3mazYPDATtKgQ0dqtkwye271a2jQ_w1iF1YIQ4vztPSUWKTptjZQai7Dcl0VBVA7Fw-Os5aMfuEcwarK0ELmKUD99ps5ubkO7Qi_gywkxF6Wirw7sK_OJKN0cGdffAtJwHdxsF8YEzsPe09jFi2G_kB7_-4MbsBYT1uRiMsM2YcG_bMF6SwaRRN-wDSdXVOYnNaWhTZ28VkmNHv7DjHzyPNuH3IH74dXd5XUaORhSm-HSTn1eVIyXymGmaAfGlCVW31iG-LLE4RlCFxso1Au110iXGyFs5ZUtlfCuUlnJd2HVUK_-SxPu9Ll9SLx0riDOIlkZITFFxS-Zyri-HFihGOvAaat8_TbB3NChVmFK39wScWamo506sEfG0bQam5Gx1LGmiMYdhXTJXlMBLKTDUmCCbNDt2A4ctVrXpqRtJdvUeqrzDhxOH8dVPff0eGL6qfhM15kOSKuyQP1wrpvPBkf34zWeM0Fn3wd_ij-ElUA6E1rZjmCpGY19FxZxHn0BBxf8cQ>
15. Cleven, A., Gubler, P., & Hüner, K. (May 7, 2009). (May 7, 2009). Design alternatives for the evaluation of design science research artifacts. Paper presented at the 1-8. 10.1145/1555619.1555645 Retrieved from <http://dl.acm.org/citation.cfm?id=1555645>
16. Cooper, J. (2000). Java™ design patterns: A tutorial (First ed.) Addison-Wesley Professional. Retrieved from <http://proquestcombo.safaribooksonline.com/0201485397>
17. Coplien, J. O., Harrison, N., & Vlissides, J.Pattern languages of program design (2. [Dr.] ed.). Reading, Mass. [u.a.]: Addison-Wesley.
18. Crawford, W., & Kaplan, J. (2003). J2EE design patterns. Sebastopol, Calif; Farnham: O'Reilly. Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwbV3JCsIwEB1cLoKgdcEV8gNqm6Qxd2kRz4JHaTsNeimC-v9OYitux8lhSEKSeUzmvQEQfOkvvt6EkJu1yXiqCWD7CjWBWIpcWaC0MTLH4FOY6SW7XSXeTncsqVVZcrM_m6v0nK4khR9LH6-HoYNGB_HKrthIqALtyKmE0W1DCVHK7VQ2Yco6-X2LJnEXGpZh4EEtL3rQqfoqsPKa9aG_41HE0NVWsIsTwCyuAxjH0X6zXZC3Y5l0OT7nxofQTmylenFzjDYcAfONQNo4JXGdScx5GkoyhdWNVypBPQbv19Hk3-AUWq60zCUEZtA0dGDzuVvVA8FqZE4>
19. Czarnecki, K. (2000). Generative programming. methods, tools, and applications. Boston: Addison-Wesley. Retrieved from <https://www.epo.org/index.html>
20. Deck, P., Setiadi, M., Kurniawan, B., Mayle, C., & Books24x7, I. (2014). Spring MVC: A tutorial (First; 1; 1st ed.). Vancouver, Canada: Brainy Software. Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwdV1NT8MwDLVgHBgXYHx1bKI_gI20SZvmPDFx4YbgGCVNIqZNCKkg8fPntClr0ThGluLEkhP7OX4BoOmczP6cCUbZQmiXGO6YSW1KNNdKuFKwknNef0TXIWaC-z6SMX__NqG1qlRfvrL5oFcaHZ57NrJDzLgy_5oveaM7eMXz0CW5z8NEgWGGyDMa-HbaMfuv1QivGVTYDzkr5TB3PYHjar3abCo8IjvX0PIMBr414RwO7McITtsPGeLgnxcwbGC6-Pl1cQnj5ePL4mmGOmTAaGTYSnoFA8z67Q3EBS1d4lzJSeYYV0oUumCWW6W1IYqYCCbdxcvPhpxCUl_rEySCa9sX-BIYxUQ4gtEezRFM293Ket7wNnQ337gxgfTCSvYMGcHdr1kaecp-uNRrT1Ll67L5eK_SWxhi_MEaRGMCRw49zk5r628BiKShQQ>
21. Deissenboeck, F., Hummel, B., & Juergens, E. (May 1, 2010). (May 1, 2010). Code clone detection in practice. Paper presented at the , 2 499-500. 10.1145/1810295.1810449 Retrieved from <http://dl.acm.org/citation.cfm?id=1810449>
22. Diggelen, W. v., & Overdijk, M. (2009). Grounded design: Design patterns as the link between theory and practice. Computers in Human Behavior, 25(5), 1056-1066. 10.1016/j.chb.2009.01.005 Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwpV1LT8MwDI54HSYhBuM1HlIuXJC6dWn64oaAaUck4BylTcLGoZpoJ7F_j52m2xiIA5wqte7WJqn92bE_ExKwnu-t6QQ_kGALeajAICWGGyVVqMHaIJdLHtnYzgoxU10a45IsnSWoNbzV3e5M341tfzqZ9J-QCD6MAuanFvaAF7_NkEsGlvijly6zPkLbbhOlPRRvNjptylc-zhyD5aDnY0O7n02V0907WDcyK1fM0bBNXpcxGGk7HvbGM-XKwr7xPP7j9fbJnkOs9LaWOyAbuuiQdtMNgjrl0CGthS6dH5IRxrQwuE6VTRG5off2SKeWz7MoqSwpYE-KG8jUJYtRW1U5p7JQtKneOiIvw4fnu5HnmjZ4OcOSwyzLjUx0rpkybKAlOJBJqg3LIsU1uD8AKXSa5UxJcHR0FIVqIJVRcRpJDr6fCY7JrsTk_qKyRYDqlFADqEbygEcZwDeTcGzDkGVxbBIAH8YPu-S6mSgxrUk6RJO99iZgVrHZZir8gfBRmDdTKb4MvgAj8tttJzjtAr_7CoYArgWgvDiLu-SqXgmL_2aiZAJT5gD-4k5twET1UcEvrMkhexzAw-jsb490Tlr13hYGhC7IVvU-05dkExbcJ4GKCZo>
23. Doebelin, E. O. (2004). Measurement systems: Application and design (5th ed.). New York; London: McGraw-Hill Higher Education. Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwY2AwNtIz0EUrEwwMk5LMU5JTgbVlWpKBRapBSlqycQqwW2ZuACykLZNQD2aCH7sNG3jLKE2Bbq1KTiwBzWzqJ2Um6ZubAxvzwJ4Ps6kpKG0bhhtDul1GJsYWFmaw43VgfBNg7QE0B6n2cBNkYAHtKBBiYErNE2bgRjoBUIRB2BcxRqcAOVW5WJRB0s01xNlDF2hOPHR4JR7iCiMxBt5E0Jr0vBLw3rUUCdB2aAtLExNz42TQoVdmoHNhUg3SgMotLVJMkwxTEiUZhDANksImKM3ABVkxAur6yzCwpgGTZqos2D8AbKBfug>
24. dofactory. (2017). Factory method. Retrieved from <http://www.dofactory.com/net/factory-method-design-pattern>
25. Downey, A. (2009). Python for software design: How to think like a computer scientist. Cambridge: Cambridge University Press. Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwbV1Lb8IwDLYGu0xCAsYQTyl_gEHSxk3PCMSRAxJHlDaJ4MKBh6b9-zklLWzs6FhyEimxHdufAxCJz9nkj06ItUArjBVkzU2SOFQiJVcaydzEkmv7uzFT1Xa7DLztryZAq3J98ZnNaXbIpkgyUNbowSV9MR_fRlV0xWfXUq4CODUhHx9LQqWKTF7ovVMy5Z0u-ORw1mjSB1OzbEHdww_a8GKP79AsP11g4Q52YLD-9nB_Rs4mO5MO_dIny0xRh_EB_eViM19NSOYuxGV2t-WLLjS0L2Y_XgrQm-kBc5ETqcmtzBTGnGuFRqNTToo4lzPj-tB-FjT4b3AIb7f0h48ZjODV0Zm242JvP5PAa9A>
26. Doyle, C., Sammon, D., & Neville, K. (2016). A design science research (DSR) case study: Building an evaluation framework for social media enabled collaborative learning environments (SMECLEs). Journal of Decision Systems, 25(sup1), 125-144. 10.1080/12460125.2016.1187411 Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwtZ1Ji9RAFICL6TmIF3HFcYF3nCGkra5KZRE8NDMtHtSDPYN4CpVaHGHMiPYI_g7_sK-2pGOLqOAlNNWVhbwv9V493kIIZ3Oa_7QmqMoIwbUsuTSCFl0hONrCVlS0krIrzLQw015q1DiO_VfB4xiK3iXS_oXwh4viAP5GBPCIEODxjzBYZtrHaGQpfydW9zl3puXJ-o3zCijUZKHOrPMPdLFRdoZf_lgLPLMpissHJkZXu887yYzPv3LpcQNTX03qR_F-mkuHd12_Wh2_XH1JPohdy_gk9v2ZlFN3xvbltxD87NIW5RAaspbudcYA_YHT16jxL8L0MeEtujcWpQvDioGuPkFup9PI1mLNCucNCYnTc-PHcD9b5hUVkxU-zghLdJy_ozpCrKW7prukC_or564XexGVwbQqd_rHFWj_qD-ozTPT52frGZnxykWWVm_fJYNAFMy3gR4eNyWS1fTJL283MZEmBXS3TJ_Tm-RGlAwsA123yJ7pb5NrKWXiDvm-hAAZRMggQQaHiNgROMDAA_YUEl4gexjxggEvwOeAgBd4vCDiBRO8IOEF23jBYYTr6C45e746PX6Rx2YfuVqwssltp3nD61qgilbKsoJb1VBlSkWFrUUnrMTNN1roTHTcCo2aqEFTnsuGlXpBFb9H9vvL3twnoKnGXYtFw7PjRWFZoy0tNc5nNS48mh2QeXq_7adQ06VdxFK5SSCtE0gbBXJA6m0ptBsPpQ08_v7UB_9-6kNyffwgHpH9zecr85jMzq_0D8sEqyE>
27. Esposito, D. (2014). Programming microsoft ASP.NET MVC (Third ed.). Redmond: Microsoft. Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwbV1LTwIxEJ4gXJQDsmrkldS7rN1td9seCYFwwZBI8Ej6jB70ov5_Z5cSHuHYHqZt0pnp93UeACxP6fjMJjjtpTIhcyJwl_ucGmG0ClZxK4SoG9EdFWaC51MmI_34czG1yurf6mfzxXwaVHhWovu7QsRVVNF82Ts70CtZWeLboAbqrCglohgV6-3sxxzdCQpuH6zj_BaaVYpBFxr-O4HOvrECiXqWwM1RlcA7eFrtYqi-cESWVQDdD9pOMnlbpa-zNVlupvfQn8_W08UYF9pGQmYb950_QBMhvn8EIpkNWQhW0CJwobWSRnIvvDbGUU1dD5ILAvoXZwdwjU6d72iCIbQCXmM_qo_6D3dBa6M>
28. Fleischmann, A. (1994). Distributed systems: Software design and implementation (1st ed.). Berlin/Heidelberg: Springer.10.1007/978-3-642-78612-9 Retrieved from <http://replace-me/ebraryid=10968238>
29. Freeman, A., & Sanderson, S. E. (2011). Pro ASP.NET MVC 3 framework (3. ed. ed.). New York: Apress.
30. Freeman, E., Freeman, E., Sierra, K., & Bates, B. (2004). Head first design patterns. Sebastopol, Calif; Farnham: O'Reilly. Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwbV3JCsIwEB1cLoKgdUGrQn5AbdNU07MofoDgsWRFL0VQ_99JbMXtmByGZEgyL5N5LwAJXUTzrzMhE4pZZWOdmUilNDVa8pXmGdNcrdjafgozvWS3q8Tb6a5LapUSN_eyuZRniXd5hIdpHeqIEB00Oiav7IqLhHESe3Jq5iTYMYyVcjtV24mMoN23aLLrQsMxDAKomaIHnepfBVJusz6Ee3Q8sWcEZkT7Cgty8TKYxXUA4932sNnP0WZepl7y5wjpENrC1asXN89r0yMgqeSCWwQjDAE9eocLPAHZ2ojI0IhKMYbg11D4r3MCrWc1iUsLTKFpcdmamZ_bAxyRaAA>
31. G, A. (2014). Spring MVC beginner's guide. Olton Birmingham: Packt Publishing. Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwtV1LS8QwEA66XhTBNz6heNCDVNok3SYHD6soIgiKq9eSx1RldRHb9fc7ybbd7noQD15Ck6FN-WY6eTTfDCGMnkbhjE_oqigB4DQ2CdcyFzrVRtMEcgBpRKymAzM1qRwnbf-qeGxD1Tsi7R-U3zwUG_AaTQBLNAIsZ-bHTbWiePgdu5PbJ9SfS77giS1pgTbxahtler5S733k9s_b6_-Yu3NKY66w19idMoOytWc1tT6MU_xgBa5Joom7r39xz4wCzdm82M1RpDxyocff7aspz2AYPj7Mo_NgtEMWeuf3_ZvJNhajLgKgo8w1fVWRtJr6EllSxQB9NvrzsvhB8FrGt_ZdPbs0KT_GRD_Q91fJAjj2xxqZg-E6WalzXgSVC9wge2NcA8Q1qHE9LgKP6iZ5vLrsX1yHVYKJUDH01KEwUlNpmITIqFhyDZDkbkamhYk1B6oZU5KDEBBRrbiNIptqlnBIjIlswrbIsnJEhGHpCYt2mwQC56ypAppDajiCIHPbzXNprAuUB5rvkMMWGNnXmwejyFqIiXSH7Lcxyj7GkUeySjconsKskXddlmIUH9QQZv4x1Tnf-vbdX-R7ZHFiaPukU36O4IDMv4zsN3qGLgM>
32. Gamma, E. (1995). Design patterns: Elements of reusable object-oriented software. Reading, Mass: Addison-Wesley. Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwbV3JCsIwEB1cLoK4i1aFfoBLmxjbnl3wAwSPMmlS9FIE6_87qXGlx-QwJJDJezOZNwHgbO7N_u4ElCg0JgQIga-FUCHFcGolNRPIUDyLLD-NmWD6m8mYn-_KSqtizMzL5kJeJDk80XWP4vUyUUTDjY78nV7xOGf-yvzIR3yIOAcnJLf9dl5jo2kjw19wsmtBxUgM2lDSaQear48VXOtnXWhs8roK95o3v0xvPXB228N6PyNDJ5twOdl1sT7U0ZSpp1kuZ1MDcGOJYUQQueSeJtfhUoUYBlGMhJpJEMRD6BRYcgpnR1B76qtNPmAM1YTOq57ke3oAKqVj9g>
33. Gaudel, V., Singhoff, F., Plantec, A., Rubini, S., Dissaux, P., & Legrand, J. (2011). An ada design pattern recognition tool for AADL performance analysis. ACM SIGAda Ada Letters, 31(3), 61-68. 10.1145/2070336.2070359 Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwhZ1LS8QwEMcH9SSIq6uiq8J8gda2aUr3WHZdPKgn70texcM2LdL9_mbS2PWxIOSUwzCEPIbM_H8DwLI4iX7dCaVRma61pgBE8sRoQVwuyjEpKYRkP8FMIztab5zVxif0VUBWx6puKHma0V7lpNwj5hzFRIuXMX1Qpny8j6krziCIm-cRK_I0AH7SnD94I4wqFbwxeqFU8-19WU3gdfepInwLw_h9q4PO6w-48X9_z-A0RJpYDVvjHA6MncLkq4sDhkN9AcvKYqUFal_LgZ0HblocK4tai33bbtBFt1hVy2fsdloDFIFpcgmz1ePb4ilyjq27gWCxDs6wKzgRVEhvey-409eAbjG1C5lkmYk0rzM5zzUrE1NIXhhVyOQGpnsszfbO3sKx_4Klwe_gqP_Ymns4dAv4CdCmmcg>
34. Gifford, D. K., Sheldon, M. A. & Turbak, F. A. (2008). Design concepts in programming languages.
35. Gilb, T., & Cockburn, A. (2008). Metrics say quality better than words. IEEE Software, 25(2), 64-67. 10.1109/MS.2008.43 Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwtV1NS8NAEF2seBDEb239gFzqpUSS3aS7OXhQUXvx1Ip4CtlJqgUt0qZo_70z2W2SCoIevCxhE0Kyb5i82cy8YUzwc8_95hOiMBFdvDgELjKMQDgEESgRhFEqk6Bo3lATZirbkFZz_wo8ziH0VEj7B_DLm-IEHqMJ4IhGgOOvzOCeOmbBtDNN5raAck65WSSNSJvm1CrGlPouGCpFf50-OuePpMqLvRu9Fv9tBuYJy20CVeVJlaFjZ9EoYpERWvM1khILAyvOnRlfiMyFfhIHdWdpqpStUfCa5zNa5MuC1lf9ng1YlyWtfYleAXmfOiN987d0BPlFNnYf-g3WEJISM-XjU5Wo4xcdY8pHLD-ea1S4M5vW-MBgm21aIu9cmiXfYSvZeJdtLd7dsT5zj7UtAg4i4FgEHIOAQwg4BQL7bHB7M7juubY3hfushO9iGKo4YPDZ1RAqLVOlAAcOyOB0ojM8r6WnhZf5Cnw-JKFGMeRKYXibKKSwB2wjoRKGcV6UOqZN5kgBvgQYdjWPAg1IvZRMtQi7KXhZANBih_TeMRlfPkkgLpexxdpmKeJ3I1ES83jKYy-m4D3yCxIX5595izW_XUd1HEjqpTz68ebHbL2ypxO2mk9m2SlrvMzSL-hUM30>
36. Giovani Guizzo, & Silvia R. Vergilio. (Jan 1, 2016). (Jan 1, 2016). Metaheuristic design pattern: Visitor for genetic operators. Paper presented at the 157.
37. Gomaa, H., & Books24x7, I. (2010). Software modeling and design: UML, use cases, patterns, and software architectures. Cambridge: Cambridge University Press. Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwdV3dT4MwEL_48aBkiW5-4WYkvs9BCxSejYsvPmniY9PSosvmYgbq_nyvHRCYmvDUC-V6lDvuevc7AEpu_fGWTqCZDISM0kQngmkVS5kHxM_QeKW4xZTuAjM1sNt14O3tU1WlVZkozcnmRM7kBE0nYekuOlyRSeYLXmgTXfFDRgljVXEqi0N0BGyfIHTxaULSGnunJiYV-lNNDx1wRDFHPYM6qCz-K0pCg4SsOdBDM7MS72r2ajp9dH5WD4r5bLEoUKe27Nb0GPZMLUMfdvRyAEfVH6dXfc8FDtVNHeqxATiPDZBrcQKjJ5zyW6y0Z1vmIAOeWCpP2byPU3Cn9893D2PkjldxIL4RFzmDnjDJ88vSFtmpC_AyyqRKhY6RHIpIpjm6YTrME6mkYL7vwk1LGPxrYYVRcCsxA7CGBi5xYdiWEf_YoGXwmIVxRFw4193xwB7JmsmHW5RyXbao_d9rcGHUEXhzJyovmlIk1_LnlqUqz7Vh5bp5J9wuhIRrxuXcQGoZfy-5_OuhQzjcxJPNNYL9HNWDvrIb4Aexsdmt>
38. Gregor, S., & Hevner, A. R. (2013). Positioning and presenting design science research for maximum impact. MIS Quarterly: Management Information Systems, 37(2), 337-355.
39. Gummesson, E. (1995). Qualitative methods in management research (Rev. ed., 6. print. ed.). Newbury Park, Calif. <<[u.a.]>>: Sage.
40. Hanmer, R. (2013). Pattern-oriented software architecture for dummies (1st ed.). GB: For Dummies. Retrieved from <http://lib.myilibrary.com?ID=426262>
41. Hevner, A. (2007). A three-cycle view of design science research. Scandinavian Journal of Information Systems, 19(2) Retrieved from: <http://aisel.aisnet.org/sjis/vol19/iss2/4>
42. IEE90. (1990). &nbsp;IEEE standard glossary of software
43. engineering terminology. IEEE Std 610.12-1990, (IEEE Std 610.12-1990) Retrieved from <http://www.mit.jyu.fi/ope/kurssit/TIES462/Materiaalit/IEEE_SoftwareEngGlossary.pdf>
44. Iversen, J., & Ngwenyama, O. (2006). Problems in measuring effectiveness in software process improvement: A longitudinal study of organizational change at danske data. International Journal of Information Management, 26(1), 30-43. 10.1016/j.ijinfomgt.2005.10.006 Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwtV1Nj9MwELWW5YKEEN8bFiQfQAKhrNzEbpKVOFTACsQBJBY4Wm7sbD-2SdWkLBz558zETtK0IBYhLmllN1GSeRq_jmfeEBIGR8zf8gnDVGdaq5hl4BAFqoZpkYk4UNjtfchVX5hpr9F77Mb-q-FhDEyPhbR_Yfz2ojAA3wECcAQQwPFSMPhgG8bUWa-LOh6IsQGbxdE4OpgqwRtfYBLY0hYOYPXkqvjaxQ6fnxfY2Wit6y5aZSNFXWyUcqLSSF2rgAWSgKZybuCjUpv8dzcA6diwK4uq0bibkvMWk0dcGYmaF-M2hH12YfLvalEz4Pczc67y-bQfyOjiak1tTZfIVNaSsDH8u3VJ1sa65zgKfR7ZPrmN_7YV9z2cWmfsNnzssm7FoHYWDBu7mB1NZzi8OKtsmM0m_HVrZJu5COfJJjFuJtuzsJunwAk8C0XbF3qaVi9M7n_6CPwAOCbfJ1dHJ---jNo9rojV_aDa5-xlH_7yfn7HnbbJBMrepsVyXW7p39ac6fQmueHMS0cWlrfInslvE89WhFO3qpT0qZM-f3aH_GjgSqc5beFKe3DFqQau1MGVbsD1mI7oJlhpDVZaZLQPVmrBSlVFLVgpgvUuOT15ffryje-ahPhpGCTcz9RAcWZCkxjge0rj1vKYx0oHGU8iE4PnCccsicU4E0YpZgYZVxkmbagsDXV4j1xXWEuSV3XNqT4gVAslooQl6TBk3LBAJSJSIlYqHPMxixKPsMYMcmk1Yf6ECY8cN-aSjtpayiovASiP3EcDS5yqViqVAxvnGEQ7M4AzTKQQoUcOGzRI545KCTQV6PtgKDzyxAKkvf9AloFkMkQFUB6hiqisvlVw_a3f8ZANExHDO3hskdXOoE79q-nnkQRjyskEHgBe4YN_ee5Dcq0LeD4k-9VqbR6RK5O1_gkrpw_j>
45. Jürgens, D. (2008). Generative programming in scientific computing. Pamm, 8(1), 10973-10974. 10.1002/pamm.200810973
46. Kamoun, A., Kacem, M. H., & Kacem, A. H. (2014). (2014). Feature model for modeling compound SOA design patterns. Paper presented at the 381-388. 10.1109/AICCSA.2014.7073224 Retrieved from <http://ieeexplore.ieee.org/document/7073224>
47. Kanjilal, J. (2017). How to implement the template method design pattern in C#. InfoWorld.Com Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwpV1LS8NAEF6sXgTBioq1FhY8pybZ3WxyEimW_oCCeCqTfWBB01hT_Pud2aa1KnjxPJcws_nm_Q1jIh3G0Q9MMLkQOqd6m5MGshwKDMx14rxIi1gEKs89YqbNcCGtxrTm3qJkgG67MFQ1v0NXhr5fJrm-r98juiNF_db2qEaHHSWY3NGQn3563qVgxPb1C3WDKxmf7o98QLhWOHxZ2Xal6ztH478-rcu6BGA11G7JHzbv44wduOqcFZPFJ28WfP7WDo9zjAI5kVS9YuTJN0eluQ2zHbwOBJwVn1d8dHvBpuPH6WgStScUolohAqKWjVZel-CNMFpCZlPlIUcfbmRi0SoJBWyyiBVocNQDtJjwpRZE7B3-_ZfsBGjSvmrCRp69YrxE-0mXp1Z4kBmYEhMqn5W2BOucAtVjg61OZlBSvcY0H7MvjfRYfydv-yJ70us_pX12nJJbpSqIumGHzXLlBqyDZloDcnS8FA>
48. Klecun, E., & Cornford, T. (2005). A critical approach to evaluation. European Journal of Information Systems, 14(3), 229-243. 10.1057/palgrave.ejis.3000540 Retrieved from <https://search.proquest.com/docview/218765768>
49. Larman, C. (2005). Applying UML and patterns: An introduction to object-oriented analysis and design and iterative development. United States: Retrieved from http://catalog.hathitrust.org/Record/004920517
50. Lockhart, P. (2012). Measurement. US: Harvard University Press. Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwtV07T8MwED5RulAx8CyhrYQYWFBQEoc0GRgAgRCCAVFYI7-iUkqFiIv4-ZydZ1MYGFis2E7s5O50j9j3GYB4J47d0AmnXPvtbhBRNDmSiJB5UlDXYWHIHGHQm2rATOW2rqrtXxmPbch6nUj7B-aXg2IDXqMIYIlCgGXDPy6r-fFM1c-_aploPj2-QwU4LpCN8nBf75uI7Cw12DCoODGomUe4EBaiBTJ-WAaA28CTnsRcKW_ClPN6pOHF38QLV2dyZj89tlBBEFQT7fOLh9Ft-atKQ6ih36Bj2mJgP8cuKifqQIemr6iXUWer9KdF34VsrmL355IFNGZ9tAFtqXM9NmFFzragc19C2KbbWKsIuAPP11ejyxs7PznCpiFGSI4dEjl0mcdFQjiTMpCJoBELHMrF0MeYDb9JUJJEgdTI6TyhHg-oS30RUnRJUe3twjrVKQYzZVIRxR4coCPlaShzgkbCT9DV8TiKr2TDwMXhHWHBYY0E8efUfHkaF3QKhsSPLOhnlInfMzSRpe46vcqbiNbDhFjQlYsdGNHrxfPQgl6jR32pWm_XkDvO3qgmAhYMCgbEZuZ8T3Ax4_6vT_ZgrZLPPqyqj7kcQGs8F99HYzo1>
51. MacDonald, S., Szafron, D., Schaeffer, J., Anvik, J., Bromling, S., & Tan, K. (2002). (2002). Generative design patterns. Paper presented at the 23-34. 10.1109/ASE.2002.1114991 Retrieved from <http://ieeexplore.ieee.org/document/1114991>
52. March, S. T., & Smith, G. F. (1995). Design and natural science research on information technology. Decision Support Systems, 15(4), 251-266. 10.1016/0167-9236(94)00041-2 Retrieved from <http://www.sciencedirect.com/science/article/pii/0167923694000412>
53. McDermid, J. A. (2013). Software engineer's reference book. Kent: Butterworth-Heinemann. Retrieved from <http://lib.myilibrary.com?ID=966482>
54. Menkya, R. (2007). Towards combining aspect-oriented design patterns; Iit.Src 2007, 1-8. Retrieved from <http://www2.fiit.stuba.sk/~vranic/proj/bp/Menkyna/menkyna.pdf>
55. MENKYNA, R. (2017). (2017). Towards combining aspect-oriented design patterns. Paper presented at the Iit. Src 2007, Bratislava, Slovakia. 1-8. Retrieved from <http://www2.fiit.stuba.sk/~vranic/proj/bp/Menkyna/menkyna.pdf>
56. Michael D Myers, & John R Venable. (2014). A set of ethical principles for design science research in information systems. Information & Management, 51(6), 801. 10.1016/j.im.2014.01.002 Retrieved from <https://search.proquest.com/docview/1555638303>
57. Noble, J. (1998). Classifying relationships between object-oriented design patterns. Paper presented at the 98-107. 10.1109/ASWEC.1998.730917 Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwjV1La8MwDBbbToNB9-jYq-DDrknzcBr7WErLdtpggx2LFzu0lyS0zf-fpCQN3WDbLSYgTGJLsqzv-wDiyA-8bz4hw1R_kjurswwzdAxKaRqkkU1MYnK0YQ6JmfasOaR3wq1ozqdHvtm3ZVZT4WyMaxPPGuh9cRWRdsHkua-uEGuKjFImfCTRpRDT8pZupxvL9o4zDPR4-vYxnxFwT_mN1QOtFQ41iwG89PUVw2qG_qq2LeTrB4fjP6Z-DsMe1ide9wHrAo5ccQmDTtdBtNv8CqaslLlmBJTYdN1yq3W1FW1blyg_qX7jlcSSjDmrsNwIIipm6yy2Q3hczN9nTx5Nalk1jBbLgHPFiVT4m2KNYfwazgx12Bc7RuLZGxAm0TpHz6iVUTJyicplHsaO-NOVVVLfwuhXm3d_vL-H0wbvR-WNBzjZbWo3gmP8xF-vIqe8>
58. Nola, R., & Sankey, H. (2007). Theories of scientific method (1. publ. ed.). Stocksfield: Acumen.
59. Nykonenko, A. A. (2012). Creational design patterns in computational linguistics: Factory method, prototype, singleton. Cybernetics and Systems Analysis, 48(1), 138-145. 10.1007/s10559-012-9383-1 Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwlV1ba9RAFB60T4pUWy9dWyEvgkKzTGYyt75JcfGlIFTRFxnmFhUhu2yy0P77njPJbu2uiL5mToaE-ebMucz5DiGcTWm5pRNihINJO628clTAlgrBC6dr7pWqmsTvEjMRtolktL-m6wRl1tu_lb6BLQyOMCsNOFklOkAVnHVY7vv1cqOLAcFDMZykJeVGrvOaf5phVykjfWiYL1bdTpo0nz6zx-TLbcjF5QaH0x-rOFaB7dA6_sffPCH7o0FavBsQdEDupfaQPLzYsLl2h-RgVABd8WZkqX77lHw7H81NeDnmayDFInN1tl3xsy1C7haxFsCa99VACX1WzHKLn-viIjevPi0-Luf9HEPBp8UlyAGQ5u0z8nn2_tP5h3Ls1VB-B4-Ply41XujoWAiY2aFJNM5pKr0whmnvRZQyOq6dkU74VKXAAAYqgsIRNVeRPyePHN7pb_tc-xePCHw8WIMVAzWjXN14Z1yTaEJuCa-lVGFCXuCCWdyK_dIFy7HbKLisbEJO1mtonceYUeg7azAuixRqE3K8GR637J3R18Oq28VA-GGZ7ZilVtdagEFX8aq2_VX_NzlJsQOUMYPc0ZYccvZrLoWCP9gaEth_B0wwnHxEyu30gA-bkWJrbSuLUHn5j3LH5AE-HoJHJ2SvX67SK3If0HoDelUTew>
60. Oodesign (2017) Builder Pattern. Retrieved from <http://www.oodesign.com>
61. Olsen, E. W., Haug, M., & Bergman, L. (2001). Software process improvement: Metrics, measurement, and process modelling. Berlin; London: Springer. Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwbV1NCwIhEB36IAiCvukT9g9Uq7uanaPoXtAxHHOpS0QU_f1Gd1uiAi96GBTG0fecNwJEfBpOvmICJ7yDjEmL0jC1YEYrTbERReKe_rzg-6MwU152-028nR7HTFpl9N29bM7wjLNISTreigS4hPNtto9ydoXuQUrEcSbmiBm55byS8xy-7-oEkt2P02TdgJJTGDShYC8tqL__VQiybdaG0ZZi41PfbHBN0_iDs4f-nsnrQH-92i03E7J6yMiXQzpH3oWadhnrl7tXth17ECDXhsACzgmQxJInaJRmaEIjrJAYsj40fw0N_g0OoZpmRrk2gnJCjmvHfnUvn2Vn2g>
62. Oram, A., & Wilson, G. (2011). Making software (1st ed.). US: O'Reilly. Retrieved from <http://replace-me/ebraryid=10761432>
63. Orlikowski, W. J., & Iacono, C. S. (2001). Research commentary: Desperately seeking the "IT" in IT research--A call to theorizing the IT artifact. Information Systems Research, 12(2), 121-134. 10.1287/isre.12.2.121.9700 Retrieved from <http://isr.journal.informs.org/cgi/content/abstract/12/2/121>
64. Pai, P., & Xavier, S. (2016). NET design patterns (1st ed.). Birmingham: Packt Publishing. Retrieved from [http://ebookcentral.proquest.com/lib/[SITE\_ID]/detail.action?docID=4796407](http://ebookcentral.proquest.com/lib/%5bSITE_ID%5d/detail.action?docID=4796407)
65. Peffers, K. (2012). Design science research in information systems.
66. Pfleeger, S. L. (2008). Software metrics: Progress after 25 years? IEEE Software, 25(6), 32-34. 10.1109/MS.2008.160 Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwnV1LT9wwEB4VuFRCPNoC4SHlQC-VAnYSOwkXVKGiXlaqtFQVJ8uvqCcem6zg5zPjOLsslTj07CR2PPbMN_bMfABFfsayNzrBSuY8WY-WaV8JUxWF9bIwFtE4Y61cLcwEY1WIKO1RSQbN7e4tHZqfo62q0b3Km8uHx4xYpOi2NVJqrMEGl1VFVA7Vn9tlyAcP3CNoNVlWlaKJ-XqcNeeT6RBXyUOtyqWFino6MK-sglDKGpl3r4zR9Ta45QmMDnyHZ3_nLiaF_VPl8X9_bge2IlhNvw-raxc--LtPsD0SQaRRL3yGb1NU5k965tMJEXTZ7iL9RXFfOLQ0kJCnuUhvcSDd5Rf4ff3j5upnFkkYMpsLioCyxtbC85yKcyF4YkWjpW6LthBaloLXXnp0Cul2VAvZWGOsd7ZscWVw41DOe7CpKVj_rg9Jfe4AUgQFrm0NeorelrwptWm58Agtta-xP5PA6SgB9TAU3VDBWWGNmkwH5kwUVAL7JB1FW7GfaasQrdZ0r8gTOCGBLV5mAQtLXGuOFY75BI7HaVfa0JmS7Tu1mPQEjhbNcUuvtH4dZL_4fK66XDElKnS30OdHlab65z6BgzfPIditJPqM-eG7HRzBxxCVEjIej2G9n839CazhUnoBWib81A>
67. Pree, W. (1994). Design patterns for object-oriented software development. Wokingham: Addison-Wesley. Retrieved from http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwbV1NTwIxEJ2IXkw8SIQAStI\_sEJ3O9tyVok\_gMQj6WfgshA-4t9nZlkRkWN7mLTJpPP6Ou8VoMhfx9nFmSDLPBmLqG2h0UvllPRGO5RWRq8m6a8x08l2-4d4W-xDI63ydscvmyO3dKNS4bjQLbpwIee2\_OJPGwj-MK0xUaax1zmNqXpQnLPqMX2EW1YUtOEmVk9g3uueCbGujS2rrSDUKFaO6ZBsxabDBAHFlg7Hb7uJIvz29HSgP\_2YvX1mFH7esC7z4-LyLjxYblWvdrWkLfRAmBALqgkaMbGqE50tVYrKYfCE\_XPbh\_b\_QINrk89wf7T2ZUbgBe4SZWwc1ts8AD1FaOE
68. Prehofer, C. (2001). Feature‐oriented programming: A new way of object composition. Concurrency and Computation: Practice and Experience, 13(6), 465-501. 10.1002/cpe.583 Retrieved from <http://onlinelibrary.wiley.com/doi/10.1002/cpe.583/abstract>
69. Richard L Baskerville, Mala Kaul, & Veda C Storey. (2015). Genres of inquiry in design-science research: Justification and evaluation of knowledge production. MIS Quarterly, 39(3), 541. Retrieved from <https://search.proquest.com/docview/1704380027>
70. Shalloway, A., & Trott, J. (2010). Design patterns explained (2. ed., 9th print. ed.). Boston, Mass. [u.a.]: Addison-Wesley.
71. Shepperd, M., & Ince, D. (1993). Derivation and validation of software metrics. Oxford: Clarendon. Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwbR1JCsIwcHC5CB4UFVfIB2ptmjTp2QUfIHgsSZugBxVc8PtO2ipu5JQEhoSZzJZZAEI6nXlfPEFwQ4UOOYus0Dic5m9mXKowZlpx9VmY6VU15-l4292yMrUqVVf3s-nrvfZR143QWq9y7kg72IZFyRwUQJLRoKyu85wzFB4I5k14rFpQcwkFbaiYYwe8BaK8cIMSNOIJInpftDUiJ0suyBTv6mzIwTW6Si9d6K-Wm_naQ5hJ6WlJ8gPRHjSVi04_XvMstqwPJMYNxQUqO9YyfDVKU5EJmQY6lpGRegCtHzjDP2sjaBQBd84FMIa6RRI1k_xiD0qqY1E>
72. Shirley Gregor, & Alan R Hevner. (2013). Positioning and presenting design science research for maximum impact. MIS Quarterly, 37(2), 337. Retrieved from <https://search.proquest.com/docview/1505325903>
73. Shirley Gregor, & David Jones. (2007). The anatomy of a design theory. Journal of the Association for Information Systems, 8(5), 312. Retrieved from <https://search.proquest.com/docview/198831825>
74. Simon, H. A. (1998). Theoe sciences of the artificial (3. ed. ed.). Cambridge, Mass. [u.a.]: MIT-Press.
75. Singh, Y. (2012). Software testing. Cambridge: Cambridge University Press. Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwbV1LC8IwDA4-LoIw5wOfsD-g1m1u3Vkc3hU8SvqYetlBJv590z1kqMeWElpI8yVp8wXAc1ds-WUTPD8JESOmmNJKcK7DrSDHAyXhuS_y1gQ1YqYP7XaVeLs9VVlaJTEzL5trcRdrbuoqKVxvkpdtXKOz98muEBAGkR_mrYFIqQnUgk1J8FSNDR6R3BqaxD1omQoDGxo67YNV9VVwyms2AOtItvGFD-1khgMjvQ5hEu9Pu8OSJF3KhMul2Jc7gi6aX-ppllezqTE4EglOTePtiNMi5qKXyAglRR0-44h8AvavoOm_yRl0CLrdIhkwh3ZCyqoX-YnetWxi-g>
76. SOA Patterns. (2017). SOA Patterns. Retrieved 22 March 2018, from <http://soapatterns.org/compound_patterns/overview>
77. Stefan Cronholm, & Hannes Göbel. (2016). Evaluation of the information systems research framework: Empirical evidence from a design science research project. Electronic Journal of Information Systems Evaluation, 19(3), 158. Retrieved from <https://search.proquest.com/docview/1860846419>
78. Stefanou, C. J. (2001). A framework for the ex-ante evaluation of ERP software. European Journal of Information Systems, 10(4), 204-215. 10.1057/palgrave.ejis.3000407 Retrieved from <http://dx.doi.org/10.1057/palgrave.ejis.3000407>
79. Stevenson, D., & Phillips, A. (2003). Implementing object equivalence in java using the template method design pattern. Paper presented at the 278-282. 10.1145/611892.611987 Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwhV1bS8MwFA7qkyDM6cQr5A-0NmmSJo8yHCIM9rD3kescbHXK5u_3JO06b-Bj-3AoSXNu-c73IVTSvMh--AThIO_XkdvMQUiPFC1SC-q0Z5Iz4dV3Yqau_eGWYHWVLvRtS1md27CKl6eCxHI5-l5OIpTrYTjueitFFfsaUY6PcBlFS5hsyXa652rHt8n4PZiSiuaNxRil7OpLjBn10HjfWNFJxjB_2bp21usXeeO_33yKBvtpPjzp4lQfHfj6DPV2cg64Pd3naJKYghN8qJ7jVxM7NNi_bRfwMyYjixo_6w-NI1Z-jiFzxJHYagnZKm6EqLFLeBC8TqSd9QBNR4_T4VPWCi5kmnGVBWNNgILVSMgpqHNKeGasLkJQlsJWEV5BMacNUYwbYapKV1DslAZ8Zim9KssLdKIjLr_epPk9d4lwIEFIcBK2oIpZx6TiXtIAWYUOIghxhfqwWrN1Q60xa1bo-q-XN-g4AekKkhFyi44271t_hw5hJz4Bn_KvDA>
80. Stockdale, R., & Standing, C. (2006). An interpretive approach to evaluating information systems: A content, context, process framework. Retrieved from http://hdl.handle.net/1959.3/61988
81. Tichy, W. F. (1997). A catalogue of general-purpose software design patterns. Paper presented at the 330-339. 10.1109/TOOLS.1997.654742 Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwjZ1LS8QwEMcH9SQI62NFVxdy8Npusm2T9CjioiBUcO9Lmoee2rLd4tc3SV-sgnprKQxp086k_8z8BiBahjj45hOWKeGpMiqlSpKIM8NiibFihnBhV_xmH8w0UHNcvxOfiqZDd-h39lUpGyecLVzb3Nh7XxsD7ZtMn0d1BScJt3Y98NEB4e1_WI_b6c9Jt8dJcLpYZ9nLmyvcY2Frda_Xig81qwlko74ifDfD8KNRXcnXD4bjP4Z-CtOxrA-9DgHrDA50cQ6Tvq8D6j7zC2D3aFB1UGnQe4umDio7KWWtUW1d96fYaqR8-geqPKOzqKdwt3pcPzwFbiibquVYbLBfIdKY2zhPuEmiSzgRLq--2Pn6O3UFiAmqec6lprmIDca5pGlib59SxqSi4hrmv9qc_XH9Bo5bIKwTNW7haLdt9BwO7YP9AiAlpRo>
82. Tichy, W. F. (1998). Should computer scientists experiment more? Computer, 31(5), 32-40. 10.1109/2.675631 Retrieved from <http://ieeexplore.ieee.org/document/675631>
83. Tzanakakis, K. (2013). The railway track and its long term behaviour: A handbook for a railway track of high quality (1st ed.). Berlin, Heidelberg: Springer. Retrieved from <http://lib.myilibrary.com?ID=500699>
84. Vaishnavi, V. K., & Kuechler, W. J. (2008). Design science research methods and patterns. Boca Raton: Auerbach Publications.
85. Venable, J. R., Pries-Heje, J., Bunker, D., & Russo, N. L. (2011). Design and diffusion of systems for human benefit. Information Technology & People, 24(3), 208-216. 10.1108/09593841111158347 Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwlV1JSwMxFH5YC6KIS1WstZCLgoep2WY7FtsiePDSozBkxV5qse3_N8mkC62X3pPHTPLy9vc9AEZ7ONmRCco4EakybAlTWOdKGMoLhm0mMpGKkODdAmZaRzJCkWUdnJlM595bDcI7Vq-_BlTdgvtXT9KC8bwBTeoMCsfvzf5w8Pmxzic4O4TUqHtl4nakMb_5L5G93tyNfA5KZ3QJX5tIiwhzDXvfSx2bv_bQHA_9iSu4iMYo6tfccw1HZtqCk1UtfAvOttAKW9CNPQ7oGcUmJn-pKEqHGyCDUA2CxFQjP3dl6QNx6MeiGi16jtweFGYCIukkrJ0sbmE8Go7f3pM4kSExZUaSvEgxVsLZaIZpayUnqnQfToXT-0w5ZajKzKZaSea8Gp2LzArJuSGUlkrnsmR3cC584f50ERr89D0gbZwKdVSoUM64oURiKblkwlJqtchwG57i4VWzGn-j2juxaqZtG1521wX_xg_Z3F3fhs7qXqv4ZOdVUXhnmXDycAClDpyugsyUPcLx4ndputBw3PAHtW_YeQ>
86. Venable, J., Pries-Heje, J., & Baskerville, R. (2016). FEDS: A framework for evaluation in design science research. European Journal of Information Systems, 25(1), 77-89. 10.1057/ejis.2014.36 Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwpV09T8MwED1RWECIjwJqoZW8wJa2Tty4gQEh2ogdEDBFduyIMpRC2__PXWKXwMDCbieKz7kP-957AFHYGwS_fILUFmN5qIXRsU1kKPTQFpirFJZbUcG6a8RMVXMhQWOcub2XLF23ec_p1LzPJbG_Ecbhev4RkI4U3bc6UY0GbGFkSkjMQT69rEswLGZKDT3M2wNMNp5dKzwmLX37NiX6bi56JV1zLUjVMk-CiqwWtQiU7tc7RVQpcth7XRmHBPtJ7fivLzqAPZejsptqUx3Chp01Yd_rPzDnDpqwUyMzbELXQSDYBXMYJ7K5H30EV-lkfH_JFEt9PxjDUWyyphtn0xkblw0lfhLzfYHH8JhOHm7vAifdEOToI-KA-EYN5zkf6Cgh2jKDmU-hDSYHiusIa5jcDLQe6VBINSyEiJTFSk8X8SjiSproBHYVtfjPliUU0LSAYeklCqUxqucknIbTVaKTUKtQ5xh_ZRvOvdGyeUXVkZVX7EOZkXEzMm4WxW3o-NXP3A-7yL6Xvg2tysrrh8RUhmChJ0__nnkG2_gKd0DTgc3l58p2oYG74QvlIOmh>
87. Vlissides, J. M. (1998). Pattern hatching (1. print. ed.). Reading, Mass. [u.a.]: Addison-Wesley.
88. Walls, J. G., Widmeyer, G. R., & El Sawy, O. A. (1992). Building an information system design theory for vigilant EIS. Information Systems Research, 3(1), 36-59. 10.1287/isre.3.1.36 Retrieved from <http://isr.journal.informs.org/cgi/content/abstract/3/1/36>
89. Walter, B., & Alkhaeir, T. (2016). The relationship between design patterns and code smells: An exploratory study. Information and Software Technology, 74, 127-142. 10.1016/j.infsof.2016.02.003 Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwrV1NT9wwEB2VVqqQELQUtEtB8h8I69hex-GGEKinCon2bDmxDVtBWG12xd9n_JFlacWpXOMc7JmR54395hmAs1Na_LUnuHra1EIGeTFhSyNd2TStVaJuFVdJdntDmCm1xmSSZc4EaYePe3f-Msm2ncxns8kNQgWKyRTRDo91zBZ8YkFLBkP8uqjXFwshwJP8Hi3C70M3XaR8oU_7x6jrKZOQJ38rW22i0dA-suo3stLVHty-HMWY-PDh6d3K5u6wf-Qe_2OVX2A3A1dynv77Ch9ctw-fB978N_iJAUcWA7XubjYnmQJGbCSJkHlU8ux6YjpLQic96R_c_X1_Rs474iIVMN74k6h4ewC_ry5_Xfwo8mMNRYsoQBWGSYvQgdGpLZWS3FXO20ZUtUdEILiflpI1WD8J71pWM6e88hX1plGSOmsNP4QdE0j93TI2_9kRECNaKdum5qbkApGRYY5xTn3LS--9qcdQDN7R8yTOoQfW2h-dvKmDNzVlQQR1DJeDC_Urg-vc-xUspNEK-gZz4JNZOP1ywaEroUtWYa3ExnAy-F-bJhxJtctel1UV1PkFp2M4Xo_nLeH18CjFzHrSkiqsJBGOHb3TDL_Ddlh34rQdw8flYuVOYAuD8BmdUhtO>
90. Weiss, C. H. (1972). Evaluation research: Methods for assessing program effectiveness. New Jersey: Prentice-Hall. Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwbZ1LC8IwDICDj4vgYb7wCfsDc3XrdD3Lhj9A8DjapWNedpr_32ROGeqxhYZAH2nS5CtAGOyF93UmCGmlovkWtkCtTlbqIArRohIYoGhSdTpgpg815x14Kx_YllbluuaXTd_cjR8pumv0yd-KGJp_uHU4UjGZYcb2jfgpQdGuDI8tbefd5oo2EtsxJqkDAy4wmEDPVjOYJh_cttuCd8o5LNPker54NDJr4ytZo0awgLHmnPSqbmrXcAluJLUw5M9oVeQyLqRRHPU5obDHQ2w0rsD5kbP-07eBEX8O9nL8tzAsaGHaXaP-ExoWXdU>
91. Welie, V. (2003). Patterns in interaction design. Interact, 2003, Retrieved from <http://www.welie.com/papers/Welie-Interact2003.pdf>
92. Zhu, H. (2005). Software design methodology (1. publ. ed.). Amsterdam u.a: Elsevier Butterworth-Heinemann.
93. Zimmer, W. (1995). Pattern languages of program design. New York: ACM Press/Addison-Wesley Publishing Co.
94. Zlobin, G. (2013). Learning python design patterns. Birmingham, UK: Packt Pub.
95. Zou, C., & Peterson, J. B. (2015). Quantifying the scientific output of new researchers using the zp-index. Scientometrics, 106(3), 901-916. 10.1007/s11192-015-1807-z Retrieved from <http://hud.summon.serialssolutions.com/2.0.0/link/0/eLvHCXMwlV3dS8MwED9UfBDEuamsfkCeRMWOJun68Sji9FVU9K1kTYIgzOHWl_313iVdy5wP-lbKJdBLcve73t0vAFIMovCHTUi4sVbYTOg8N9ygT7LEzKX0WPFYyXKVmAlE8ydj8jFYJiid3W5b3zinukk-DHkWpeECrTBHX0ftvm9PjS3G_etsMXfHWjZ5zd9mWPFM29QUUs3W8qPO7Yw68Nr-a1HuZsPBe6Xr9q81Psd_fMY-7NVIlN34rdOFDTPpQWd5ywOrD30PuvXTjF3UHNWXB3D_WCkqM6ImKYYYkvnGSqo7Yp_VHKdgn5YhZGc1mxDBTEZF9l58MQ0dT-MhvIzunm8fwvpOhrD0pA5ZmScpwhI-RvCSYkAiDAY1mYrKSCoR6zKXkksTcROX1iqbK2nyIeV3eZIqJY9gV1Ht_mTuevx0H5jCIEeWY5NHOo5FbJWwSRZjYDuUpbZaBHC1XJhi6jk4ipZtmRRYoAILUmCxCKDvl64RTQg1SYTFAVwvF6Gdxw0XNL7Wt59nqm0A52viggri_BiMpQvphI__KngCO_Tel7Cdwtb8qzJnsIk75xu3den7>

# Bibliography

# Appendix A: Factory method combined with strategy pattern

|  |
| --- |
| public class StatementExtractorFactory {  private static Map <String,StatementExtractor> extractors =  new HashMap<>();  static {  extractors.put(“BankAccountStatement”,  new BankAccountStatementExtractor());  extractors.put(“CreditCardStatement”,  new CreditCardStatementExtractor());  }  public static StatementExtractor getInstance(String type) {  return extractors.get(type);  }  } |
| public class DocumentExtractor {  public void extractDocument() {  …  StatementExtractor extractor =  StatementExtractorFactory.getInstance(type);  extractor.extract(statement);  …  }  } |
| StatementExtractor extractor = null;  if(type.equals(“BankAccountStatement”)) {  extractor = new BankAccountStatementExtractor();  }  else if(type.equals(“CreditCardStatement”)) {  extractor = new CreditCardStatementExtractor();  }  extractor.extract(statement); |

# Appendix B: Short-listed pattern combinations

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Pattern | Factory Method | | | Template Method |
| Problem | A framework needs to standardize the architectural model for multiple applications but allow each individual application to define its own domain objects and be responsible for its instantiation. | | | Two dissimilar components maintain significant similarities but possess varying implementations. Consequently, any modifications would require expenditure of duplicate efforts. |
| Intent | * There is a need for a universal interface for the creation of individual objects but subclasses are permitted to decide which class to instantiate. This in effect means deferment of class instantiation to subclasses. * Defining a "virtual" constructor. * The new operator considered harmful. | | | * Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. * Base class declares algorithm “placeholders”, and derived classes implement the placeholders. |
| Commonality | -Maintains a common interface  -Defers behaviour to subclasses | | | |
|  |  | | | |
| Pattern | Prototype Pattern | | Template Method | |
| Problem | A system "hard wires" the class of object to be created with each "new" expression | | Two different components have significant similarities but demonstrate no reuse of common interface or implementation. If a change common to both components becomes necessary, duplicate effort must be expended. | |
| Intent | * Specify the kinds of objects to create using a prototypical instance and create new objects by copying this prototype. * Co-opt one instance of a class for use as a breeder of all future instances. * The new operator considered harmful. | | * Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. * Base class declares algorithm “placeholders”, and derived classes implement the placeholders. | |
| Commonality | -Makes use of a base object or class, in one instance for breeding cloned objects and in the other for breeding cloned actions. | | | |
|  |  | | | |
| Pattern | Builder | | Template Method | |
| Problem | An application needs to create the elements of a complex aggregate. The specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage. | | Two different components have significant similarities but demonstrate no reuse of common interface or implementation. If a change common to both components becomes necessary, duplicate effort must be expended. | |
| Intent | * Separate the construction of a complex object from its representation so that the same construction process can create different representations. * Parse a complex representation, create one of several targets. | | * Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. * Base class declares algorithm “placeholders”, and derived classes implement the placeholders. | |
| Commonality | -Deferment to subclasses | | | |
|  |  | | | |
| Pattern | Singleton | | Template Method | |
| Problem | A system requires one and only one instance of an object coupled with a singular access point to that object. | | Two components have significant similarities but do not require the reuse of common interface or maintain very different implementations. If a change common to both components becomes necessary, duplicate effort must be expended. | |
| Intent | * Guarantees that a class possesses only one instance and provides a global point of access to that instance. * Encapsulates "just-in-time initialization" or "initialization on first use". | | * Defines the frame of an algorithm but opts to defer the execution of some steps of the algorithm to client subclasses. Template Method lets subclasses redefine particular steps of an algorithm without altering an algorithm's construction. * Base class declares algorithm “placeholders”, and derived classes implement the placeholders. | |
|  |  | | | |
| Pattern | State Pattern | Chain of Responsibility | | |
| Problem | A monolithic object's behaviour is a function of its state, and it must change its behaviour at run-time depending on that state | Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings | | |
| Intent | * Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class. * An object-oriented state machine * wrapper + polymorphic wrappee + collaboration | * Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it. * Launch-and-leave requests with a single processing pipeline that contains many possible handlers. * An object-oriented linked list with recursive traversal. | | |
| Commonality | -Both Patterns are easily extended  -Details algorithmic pipeline | | | |

# Appendix C: Templated Factory Scenarios (A \* B)

## C.01 Scenario 1

A game environment spawns two main enemy types for a user to interact with. These “enemytypes” possess a similar set of action which achieve varying results.

The template method is used here to model the creation of the enemy objects and the actions attributed to them.

|  |  |
| --- | --- |
| class Program  {  static void Main(string[] args)  {  EnemyTypeFactory[] factory = new EnemyTypeFactory[2];  factory[0] = new humanEnemyFactory();  factory[1] = new robotEnemyFactory();  foreach (var enemytype in factory)  {  EnemyType enemy = enemytype.spawnEnemyType();  Console.WriteLine("spawned {0}", enemy.GetType().Name);  enemy.shoots();  }  Console.ReadKey();  }  } | CLIENT |
| abstract class EnemyTypeFactory  {  public abstract EnemyType spawnEnemyType();  } | FACTORY |
| class humanEnemyFactory : EnemyTypeFactory  {  public override EnemyType spawnEnemyType()  {  return new humanEnemy();  }  } | CONCRETE FACTORYA |
| class robotEnemyFactory : EnemyTypeFactory  {  public override EnemyType spawnEnemyType()  {  return new robotEnemy();  }  } | CONCRETE FACTORYB |
| abstract class EnemyType  {  public String name;  public String amtDamage;  public String getName()  {  return name;  }  public void setName(String newName)  {  name = newName;  }  public String getDamage()  {  return amtDamage;  }  public void setDamage(String newDamage)  {  amtDamage = newDamage;  }  public void shoots()  {  pickWeapon();  reloadWeapon();  Console.WriteLine(getName() + " shoots playercharacter and does " + getDamage());  }  public abstract void pickWeapon();  public abstract void reloadWeapon();  } | PRODUCT TEMPLATE |
| class humanEnemy : EnemyType  {  public humanEnemy()  {  setName("humanPatrol");  setDamage("15.0");  }  public override void pickWeapon()  {  Console.WriteLine("human enemy picked weapon");  }  public override void reloadWeapon()  {  Console.WriteLine("human enemy reloaded weapon");  }  } | CONCRETE PRODUCTA |
| class robotEnemy : EnemyType  {  public robotEnemy()  {  setName("DoomBot");  setDamage("25.0");  }  public override void pickWeapon()  {  Console.WriteLine("robot enemy picked weapon");  }  public override void reloadWeapon()  {  Console.WriteLine("robot enemy reloaded weapon");  }  } | CONCRETE PRODUCTB |

# Appendix D: Template and Factory Scenarios (A + B)

## D.01 Scenario 1

The code here implements the enemy creation process in a game environment enemy object are spawned and attack a player dealing varied damage amounts depending on the enemy type. The factory pattern is responsible for the creation of enemy objects while the template method is used to assign enemy behaviour to the enemy objects. The code depicts pattern interaction.

|  |  |
| --- | --- |
| class Program  {  static void Main(string[] args)  {  EnemyTypeFactory[] factory = new EnemyTypeFactory[2];  factory[0] = new humanEnemyFactory();  factory[1] = new robotEnemyFactory();  foreach (var enemytype in factory)  {  EnemyType enemy = enemytype.spawnEnemyType();  Console.WriteLine("spawned {0}", enemy.GetType().Name);  enemy.shoots();  }  Console.ReadKey();  }  } | CLIENT |
| abstract class EnemyTypeFactory  {  public abstract EnemyType spawnEnemyType();  } | FACTORY |
| class humanEnemyFactory : EnemyTypeFactory  {  public override EnemyType spawnEnemyType()  {  return new humanEnemy();  }  } | CONCRETE FACTORYA |
| class robotEnemyFactory : EnemyTypeFactory  {  public override EnemyType spawnEnemyType()  {  return new robotEnemy();  }  } | CONCRETE FACTORYB |
| abstract class EnemyBehaviour  {  public void shoots()  {  pickWeapon();  reloadWeapon();  pullTrigger();  }  public void pickWeapon()  {  Console.WriteLine("Enemy picked up weapon");  }  public void reloadWeapon()  {  Console.WriteLine("Enemy reloaded weapon");  }  public abstract void pullTrigger();  } | ABSTRACT TEMPLATE CLASS |
| abstract class EnemyType : EnemyBehaviour  {  public String name;  public String amtDamage;  public String getName()  {  return name;  }  public void setName(String newName)  {  name = newName;  }  public String getDamage()  {  return amtDamage;  }  public void setDamage(String newDamage)  {  amtDamage = newDamage;  }  } | PRODUCT |
| class humanEnemy : EnemyType  {  public humanEnemy()  {  setName("humanPatrol");  setDamage("15.0");  }  public override void pullTrigger()  {  Console.WriteLine(getName() + " shoots playercharacter and does " + getDamage());  }  } | CONCRETE PRODUCT |
| class robotEnemy : EnemyType  {  public robotEnemy()  {  setName("DoomBot");  setDamage("25.0");  }  public override void pullTrigger()  {  Console.WriteLine(getName() + " shoots playercharacter and does " + getDamage());  }  } | CONCRETE PRODUCT |

## D.02 Scenario 2

A factory produces both cars and bikes. The objects are then acted on by a driver that boards the object, starts it and stops it after some time. The code shown below models this system by pairing the template method with the factory method.

|  |  |
| --- | --- |
| class Program  {  static void Main(string[] args)  {  vehicleFactory[] vFactory = new vehicleFactory[2];  vFactory[0] = new carFactory();  vFactory[1] = new bikeFactory();  foreach (var item in vFactory)  {  vehicle v = item.MakeVehicle();  Console.WriteLine("made {0}", v.GetType().Name);  v.move();  }  Console.ReadKey();  }  } | CLIENT |
| abstract class vehicleFactory  {  public abstract vehicle MakeVehicle();  } | FACTORY |
| class carFactory : vehicleFactory  {  public override vehicle MakeVehicle()  {  return new car();  }  } | CONCRETE FACTORYA |
| class bikeFactory : vehicleFactory  {  public override vehicle MakeVehicle()  {  return new bike();  }  } | CONCRETE FACTORYB |
| public abstract class vehicularBehaviour  {  public void move()  {  receiveDriver();  start();  stop();  }  private void stop()  {  Console.WriteLine("vehicle stopped moving");  }  private void start()  {  Console.WriteLine("vehicle started moving");  }  public abstract void receiveDriver();  } | ABSTRACT TEMPLATE CLASS |
| public abstract class vehicle : vehicularBehaviour  {  } | PRODUCT |
| class car : vehicle  {  public override void receiveDriver()  {  Console.WriteLine("Driver boards car");  }  } | CONCRETE PRODUCTA |
| class bike : vehicle  {  public override void receiveDriver()  {  Console.WriteLine("Rider climbs bike");  }  } | CONCRETE PRODUCTB |

# Appendix E: Prototyped Template Scenarios (A \* B)

## E.01 Scenario 1

Here we consider the character generator for a military style game. The game has a troop manager that supports the creation of three different soldier types. For optimization reasons, the developers have been asked to limit the creation of objects to the minimum, but a core requirement is the ability to modify the abilities of the instantiated objects by adding hook methods which are likely to change over time as development progresses. Using the Prototyped Template, this can be achieved as shown. The code demonstrates the Prototyped Template pattern in which new Soldier objects are created by copying pre-existing, user-defined Soldier types. Appendix F.E.01 depicts a way of achieving this without the generative pattern

|  |  |
| --- | --- |
| class Program  {  static void Main(string[] args)  {  ArmyManager troopManager = new ArmyManager();  // Initialize with standard soldier types  troopManager["Commander"] = new Soldier("Genaral", "David", "Wilson");  troopManager["SquadLeader"] = new Soldier("Genaral", "Gary", "Allen");  troopManager["GruntUnitA-soldier"] = new Soldier("Corporal", "Toyin", "Obikoya");  // User adds custom Soldier type  troopManager["GruntUnitB-soldier"] = new Soldier("Corporal", "John", "Alamina");  // User clones selected soldier types  Soldier soldier1 = troopManager["Commander"].Clone() as Soldier;  soldier1.doAction();  Soldier soldier2 = troopManager["GruntUnitB-soldier"].Clone() as Soldier;  soldier2.doAction();  Soldier soldier3 = troopManager["GruntUnitA-soldier"].Clone() as Soldier;  soldier3.doAction();  Soldier soldier4 = troopManager["SquadLeader"].Clone() as Soldier;  soldier4.doAction();  // Wait for user  Console.ReadKey();  }  } | CLIENT |
| abstract class SoldierPrototype  {  public abstract SoldierPrototype Clone();  public abstract void March();  public abstract void Fight();  public void doAction()  {  March();  Fight();  }  } | PROTOTYPE TEMPLATE |
| class Soldier : SoldierPrototype  {  private string \_rank;  private string \_fName;  private string \_lName;  public string firstName  {  get { return \_fName; }  set { \_fName = value; }  }  public string LastName  {  get { return \_lName; }  set { \_lName = value; }  }  public string rank  {  get { return \_rank; }  set { \_rank = value; }  }  // Constructor  public Soldier(string rank, string firstName, string lastName)  {  this.\_rank = rank;  this.\_fName = firstName;  this.\_lName = lastName;  }  // Create a shallow copy  public override SoldierPrototype Clone()  {  Console.WriteLine("Cloning soldier : {0} {1} {2}",\_rank, \_fName, \_lName);  return this.MemberwiseClone() as SoldierPrototype;  }  public override void March()  {  Console.WriteLine("{0} {1} is marching",\_fName, \_lName);  }  public override void Fight()  {  Console.WriteLine("{0} {1} is fighting", \_fName, \_lName);  }  } | CONCRETE PROTOTYPE |
| class ArmyManager  {  private Dictionary<string, SoldierPrototype> \_soldiers = new Dictionary<string, SoldierPrototype>();  public SoldierPrototype this[string key]  {  get { return \_soldiers[key]; }  set { \_soldiers.Add(key, value); }  }  } | PROTOTYPE MANAGER |

## E.02 Scenario 2

Here we consider an application that allows one add colour filters to a subject (picture). The application supports a colour palette that makes this possible. The developers have to conserve system resources in populating the individual filters. Using the prototype pattern, it is possible to achieve this and combining with the template method makes the application more secure. Furthermore, any modification which is typical of application development would be easier to implement. Appendix F.02 depicts a way of achieving this without the generative pattern

|  |  |
| --- | --- |
| class Program  {  static void Main(string[] args)  {  ColourPalette colourPalette = new ColourPalette();  // Initialize with standard filter types  colourPalette["Sunny-filter"] = new Colour("Yellow");  colourPalette["Dawn-filter"] = new Colour("Acqua");  // User adds custom filter type  colourPalette["sunset-filter"] = new Colour("Maroon");  // User clones selected filter types  Colour colour1 = colourPalette["Sunny-filter"].Clone() as Colour;  colour1.doAction();  Colour colour2 = colourPalette["sunset-filter"].Clone() as Colour;  colour2.doAction();  // Wait for user  Console.ReadKey();  }  } | CLIENT |
| abstract class colourPrototype  {  public abstract colourPrototype Clone();  public abstract void desaturate();  public abstract void sharpen();  public void doAction()  {  desaturate ();  sharpen();  }  } | PROTOTYPE TEMPLATE |
| class Colour : colourPrototype  {  private string \_name;  public string colourName  {  get { return \_name; }  set { \_name = value; }  }  // Constructor  public Colour(string colourName)  {  this.\_name = colourName;  }  // Create a shallow copy  public override colourPrototype Clone()  {  Console.WriteLine("Cloning soldier : {0} {1} {2}",\_rank, \_fName, \_lName);  return this.MemberwiseClone() as colourPrototype;  }  public override void desaturate();  {  Console.WriteLine("{0} applied",\_name);  }  public override void sharpen()  {  Console.WriteLine("{0} {1} is fighting", \_name);  }  } | CONCRETE PROTOTYPE |
| class ColourPalette  {  private Dictionary<string, colourPrototype> \_colours = new Dictionary<string, colourPrototype>();  public colourPrototype this[string key]  {  get { return \_colours[key]; }  set { \_colours.Add(key, value); }  }  } | PROTOTYPE MANAGER |

# Appendix F: Prototype and Template Scenarios (A + B)

## F.01 Scenario 1

|  |  |
| --- | --- |
| class Program  {  static void Main(string[] args)  {  ArmyManager troopManager = new ArmyManager();  // Initialize with standard soldier types  troopManager["Commander"] = new Soldier("Genaral", "David", "Wilson");  troopManager["SquadLeader"] = new Soldier("Genaral", "Gary", "Allen");  troopManager["GruntUnitA-soldier"] = new Soldier("Corporal", "Toyin", "Obikoya");  // User adds custom Soldier type  troopManager["GruntUnitB-soldier"] = new Soldier("Corporal", "John", "Alamina");  // User clones selected soldier types  Soldier soldier1 = troopManager["Commander"].Clone() as Soldier;  Soldier soldier2 = troopManager["GruntUnitB-soldier"].Clone() as Soldier;  Soldier soldier3 = troopManager["GruntUnitA-soldier"].Clone() as Soldier;  Soldier soldier4 = troopManager["SquadLeader"].Clone() as Soldier;  troopManager.doAction();  // Wait for user  Console.ReadKey();  }  } | CLIENT |
| abstract class SoldierPrototype  {  public abstract SoldierPrototype Clone();  } | PROTOTYPE |
| class Soldier : SoldierPrototype  {  private string \_rank;  private string \_fName;  private string \_lName;  public string firstName  {  get { return \_fName; }  set { \_fName = value; }  }  public string LastName  {  get { return \_lName; }  set { \_lName = value; }  }  public string rank  {  get { return \_rank; }  set { \_rank = value; }  }  // Constructor  public Soldier(string rank, string firstName, string lastName)  {  this.\_rank = rank;  this.\_fName = firstName;  this.\_lName = lastName;  }  // Create a shallow copy  public override SoldierPrototype Clone()  {  Console.WriteLine("Cloning soldier : {0} {1} {2}",\_rank, \_fName, \_lName);  return this.MemberwiseClone() as SoldierPrototype;  }  } | CONCRETE PROTOTYPE |
| class ArmyManager: SoldierAction  {  private Dictionary<string, SoldierPrototype> \_soldiers = new Dictionary<string, SoldierPrototype>();  public SoldierPrototype this[string key]  {  get { return \_soldiers[key]; }  set { \_soldiers.Add(key, value); }  }  public override void Fight()  {  Console.WriteLine("{0} is fighting", \_soldiers);  }  public override void March()  {  Console.WriteLine("{0} is marching", \_soldiers);  }  } | PROTOTYPE MANAGER |
| abstract class SoldierAction  {  public abstract void March();  public abstract void Fight();  public void doAction()  {  March();  Fight();  }  } | TEMPLATE ABSTRACT CLASS |

## F.02 Scenario 2

|  |  |
| --- | --- |
| class Program  {  static void Main(string[] args)  {  ArmyManager troopManager = new ArmyManager();  // Initialize with standard soldier types  colourPalette["Sunny-filter"] = new Colour("Yellow");  colourPalette["Dawn-filter"] = new Colour("Acqua");  // User adds custom Soldier type  colourPalette["sunset-filter"] = new Colour("Maroon");  // User clones selected soldier types  Colour colour1 = colourPalette["Sunny-filter"].Clone() as Colour;  Colour colour2 = colourPalette["Sunset-filter"].Clone() as Colour;  colourPalette.doAction();  // Wait for user  Console.ReadKey();  }  } | CLIENT |
| abstract class ColourPrototype  {  public abstract ColourPrototype Clone();  } | PROTOTYPE |
| class Colour : ColourPrototype  {  private string \_Name;  public string colourName  {  get { return \_Name; }  set { \_Name = value; }  }  // Constructor  public Colour(string colourName)  {  this.\_Name = colourName;  }  // Create a shallow copy  public override ColourPrototype Clone()  {  Console.WriteLine("Cloning colour : {0}", \_Name);  return this.MemberwiseClone() as ColourPrototype;  }  } | CONCRETE PROTOTYPE |
| class ArmyManager: filterAction  {  private Dictionary<string, ColourPrototype> \_soldiers = new Dictionary<string, ColourPrototype>();  public ColourPrototype this[string key]  {  get { return \_ colours[key]; }  set { \_ colours.Add(key, value); }  }  public override void desaturate()  {  Console.WriteLine("{0} applied", \_ colours);  }  public override void sharpen()  {  Console.WriteLine("{0} applied", \_colours);  }  } | PROTOTYPE MANAGER |
| abstract class filterAction  {  public abstract void desaturate();  public abstract void sharpen();  public void doAction()  {  desaturate();  sharpen();  }  } | ABSTRACT TEMPLATE CLASS |

# Appendix G: Templated Builder Scenarios (A \* B)

## G.01 Scenario 1

The following scenario illustrates how a PersonDayPlanner schedules activities for two type of individuals. One a college student and the other is a child.

Using the templated builder, the code implements this day planner allowing for a template method to control the process of building the plan. Appendix H.H.01 depicts a way of achieving this without the generative pattern

|  |  |
| --- | --- |
| class Program  {  static void Main(string[] args)  {  DayPlanBuilder builder = new DayPlanBuilder();  PersonDayPlan[] factories = new PersonDayPlan[2];  factories[0] = new CollegeStudentSaturday();  factories[1] = new LittleKidSaturday();  foreach (var item in factories)  {  item.Schedule();  }  Console.ReadKey();  }  } | CLIENT |
| abstract class PersonDayPlan  {  public abstract void Pending();  public abstract void Complete();  public abstract void outputTime();  public abstract void MorningPlan();  public abstract void AfternoonPlan();  public abstract void NightPlan();  public void Schedule()  {  Pending();  MorningPlan();  AfternoonPlan();  NightPlan();  Complete();  outputTime();  }  } | ABSTRACT TEMPLATE |
| class DayPlanBuilder  {  private PersonDayPlan DayPlan { get; set; }  public DayPlanBuilder() { } // default constructor  public DayPlanBuilder(PersonDayPlan dayPlan)  {  this.DayPlan = dayPlan;  }  public void SetPersonDayPlan(PersonDayPlan dayPlan)  {  this.DayPlan = dayPlan;  }  public void BuildPlan()  {  if (this.DayPlan != null)  {  this.DayPlan.MorningPlan();  this.DayPlan.AfternoonPlan();  this.DayPlan.NightPlan();  // You can control the steps or add new logic inside the builder  if (this.DayPlan is CollegeStudentSaturday)  {  Console.WriteLine("Play games no later than 12PM.");  }  }  }  } | BUILDER |
| class CollegeStudentSaturday : PersonDayPlan  {  public override void Pending()  {  Console.WriteLine("task Pending");  }  public override void outputTime()  {  Console.WriteLine("Day ends at 10pm");  }  public override void MorningPlan()  {  Console.WriteLine("Have a simple breakfast and do cleanup.");  }  public override void AfternoonPlan()  {  Console.WriteLine("Study for 2 hours then go to gym for exercise.");  }  public override void NightPlan()  {  Console.WriteLine("Eat out with friends and watch movie after");  }  public override void Complete()  {  Console.WriteLine("tasks completed");  }  } | CONCRETE BUILDER |
| class LittleKidSaturday : PersonDayPlan  {  public override void Pending()  {  Console.WriteLine("tasks pending");  }  public override void outputTime()  {  Console.WriteLine("day ending 6pm");  }  public override void MorningPlan()  {  Console.WriteLine("McDonald's happy meal for breakfast, then play at park.");  }  public override void AfternoonPlan()  {  Console.WriteLine("Friend's birthday party.");  }  public override void NightPlan()  {  Console.WriteLine("Play LEGO and watch TV.");  }  public override void Complete()  {  Console.WriteLine("tasks completed");  }  } | CONCRETE BUILDER |

## G.02 Scenario 2

The following scenario illustrates how a PersonDayPlanner schedules activities for two type of individuals. One a college student and the other is a child.

Using the templated builder, the code implements this day planner allowing for a template method to control the process of building the plan. Appendix H.H.02 depicts a way of achieving this without the generative pattern

|  |  |
| --- | --- |
| class Program  {  static void Main(string[] args)  {  DayPlanBuilder builder = new DayPlanBuilder();  PersonDayPlan[] factories = new PersonDayPlan[2];  factories[0] = new CollegeStudentSaturday();  factories[1] = new LittleKidSaturday();  foreach (var item in factories)  {  item.Schedule();  }  Console.ReadKey();  }  } |  |
| abstract class PersonDayPlan  {  public abstract void Pending();  public abstract void Complete();  public abstract void outputTime();  public abstract void MorningPlan();  public abstract void AfternoonPlan();  public abstract void NightPlan();  public void Schedule()  {  Pending();  MorningPlan();  AfternoonPlan();  NightPlan();  Complete();  outputTime();  }  } |  |
| class DayPlanBuilder  {  private PersonDayPlan DayPlan { get; set; }  public DayPlanBuilder() { } // default constructor  public DayPlanBuilder(PersonDayPlan dayPlan)  {  this.DayPlan = dayPlan;  }  public void SetPersonDayPlan(PersonDayPlan dayPlan)  {  this.DayPlan = dayPlan;  }  public void BuildPlan()  {  if (this.DayPlan != null)  {  this.DayPlan.MorningPlan();  this.DayPlan.AfternoonPlan();  this.DayPlan.NightPlan();  // You can control the steps or add new logic inside the builder  if (this.DayPlan is CollegeStudentSaturday)  {  Console.WriteLine("Play games no later than 12PM.");  }  }  }  } |  |
| class CollegeStudentSaturday : PersonDayPlan  {  public override void Pending()  {  Console.WriteLine("task Pending");  }  public override void outputTime()  {  Console.WriteLine("Day ends at 10pm");  }  public override void MorningPlan()  {  Console.WriteLine("Have a simple breakfast and do cleanup.");  }  public override void AfternoonPlan()  {  Console.WriteLine("Study for 2 hours then go to gym for exercise.");  }  public override void NightPlan()  {  Console.WriteLine("Eat out with friends and watch movie after");  }  public override void Complete()  {  Console.WriteLine("tasks completed");  }  } |  |
| class LittleKidSaturday : PersonDayPlan  {  public override void Pending()  {  Console.WriteLine("tasks pending");  }  public override void outputTime()  {  Console.WriteLine("day ending 6pm");  }  public override void MorningPlan()  {  Console.WriteLine("McDonald's happy meal for breakfast, then play at park.");  }  public override void AfternoonPlan()  {  Console.WriteLine("Friend's birthday party.");  }  public override void NightPlan()  {  Console.WriteLine("Play LEGO and watch TV.");  }  public override void Complete()  {  Console.WriteLine("tasks completed");  }  } |  |

# Appendix H: Template and Builder Scenarios (A + B)

## H.01 Scenario 1

|  |  |
| --- | --- |
| class Program  {  static void Main(string[] args)  {  DayPlanBuilder planBuilder = new DayPlanBuilder(new LittleKidSaturday());  planBuilder.BuildPlan();  planBuilder.Schedule();  // changing to student's plan  planBuilder.SetPersonDayPlan(new CollegeStudentSaturday());  planBuilder.BuildPlan();  planBuilder.Schedule();  Console.ReadKey();  }  } | CLIENT |
| interface IPersonDayPlan  {  void MorningPlan();  void AfternoonPlan();  void NightPlan();  } | BUILDER |
| class DayPlanBuilder: Scheduler  {  private IPersonDayPlan DayPlan { get; set; }  public DayPlanBuilder() { } // default constructor  public DayPlanBuilder(IPersonDayPlan dayPlan)  {  this.DayPlan = dayPlan;  }  public void SetPersonDayPlan(IPersonDayPlan dayPlan)  {  this.DayPlan = dayPlan;  }  public void BuildPlan()  {  if (this.DayPlan != null)  {  this.DayPlan.MorningPlan();  this.DayPlan.AfternoonPlan();  this.DayPlan.NightPlan();  // You can control the steps or add new logic inside the builder  if (this.DayPlan is CollegeStudentSaturday)  {  Console.WriteLine("Play games no later than 12PM.");  }  }  }  public override void Pending()  {  Console.WriteLine("tasks pending");  }  public override void Complete()  {  Console.WriteLine("tasks completed");  }  public override void outputTime()  {  Console.WriteLine("day over");  }  } | DIRECTOR |
| class CollegeStudentSaturday : IPersonDayPlan  {  public void MorningPlan()  {  Console.WriteLine("Have a simple breakfast and do cleanup.");  }  public void AfternoonPlan()  {  Console.WriteLine("Study for 2 hours then go to gym for exercise.");  }  public void NightPlan()  {  Console.WriteLine("Eat out with friends and watch movie after");  }  } | CONCRETE BUILDER |
| class LittleKidSaturday : IPersonDayPlan  {  public void MorningPlan()  {  Console.WriteLine("McDonald's happy meal for breakfast, then play at park.");  }  public void AfternoonPlan()  {  Console.WriteLine("Friend's birthday party.");  }  public void NightPlan()  {  Console.WriteLine("Play LEGO and watch TV.");  }  } | CONCRETE BUILDER |
| abstract class Scheduler  {  public abstract void Pending();  public abstract void Complete();  public abstract void outputTime();  public void Schedule()  {  Pending();  Complete();  outputTime();  }  } | ABSTRACT TEMPLATE CLASS |

## H.02 Scenario 2

|  |  |
| --- | --- |
| class Program  {  static void Main(string[] args)  {  documentBuilder builder;  // Create shop with vehicle builders  documentDirector director = new documentDirector();  // Construct and display vehicles  builder = new invoiceBuilder();  director.Construct(builder);  director.directorSign();  director.ManagerSign();  builder.document.Show();  // Wait for user  Console.ReadKey();  }  } | CLIENT |
| class documentDirector: Signage  {  // Builder uses a complex series of steps  public void Construct(documentBuilder docBuilder)  {  docBuilder.BuildHeader();  docBuilder.BuildContent();  docBuilder.BuildFooter();  }  public override void directorSign()  {  Console.WriteLine("director signature applied");  }  public override void ManagerSign()  {  Console.WriteLine("manager signature applied");  }  } | DIRECTOR |
| abstract class documentBuilder  {  protected Document doc;  // Gets document instance  public Document document  {  get { return doc; }  }  // Abstract build methods  public abstract void BuildHeader();  public abstract void BuildContent();  public abstract void BuildFooter();  } | BUILDER |
| class invoiceBuilder : documentBuilder  {  public invoiceBuilder()  {  doc = new Document("Invoice");  }  public override void BuildHeader()  {  doc["Header"] = "document title is: Expense Bill";  }  public override void BuildContent()  {  doc["Content"] = "500 cc";  }  public override void BuildFooter()  {  doc["Footer"] = "Thank you for you patronage";  }  } | CONCRETE BUILDER |
| class Document  {  private string \_documentType;  private Dictionary<string, string> \_sections =  new Dictionary<string, string>();  // Constructor  public Document(string docType)  {  this.\_documentType = docType;  }  // Indexer  public string this[string key]  {  get { return \_sections[key]; }  set { \_sections[key] = value; }  }  public void Show()  {  Console.WriteLine("\n---------------------------");  Console.WriteLine("document Type: {0}", \_documentType);  Console.WriteLine(" Header : {0}", \_sections["Header"]);  Console.WriteLine(" Content : {0}", \_sections["Content"]);  Console.WriteLine(" Footer: {0}", \_sections["Footer"]);  }  } | PRODUCT |
| abstract class Signage  {  public abstract void directorSign();  public abstract void ManagerSign();  public void sign()  {  directorSign();  ManagerSign();  }  } | ABSTRACT TEMPLATE CLASS |